# UNIDAD Nº 4
# EL PARADIGMA LOGICO

## 1. INTRODUCCION A LA PROGRAMACION LOGICA

Una forma de razonar para resolver problemas en matemáticas se fundamenta en la lógica de primer orden. El conocimiento básico de las matemáticas se puede representar en la lógica en forma de axiomas, a los cuales se añaden reglas formales para deducir cosas verdaderas (teoremas) a partir de los axiomas. Gracias al trabajo de algunos matemáticos de finales del siglo pasado y principios de éste, se encontró la manera de automatizar computacionalmente el razonamiento lógico -particularmente para un conjunto significativo de la lógica de primer orden- que permitió que la lógica matemática diera origen a otros tipos de lenguajes de programación, conocidos como lenguajes lógicos. También se conoce a estos lenguajes como lenguajes declarativos, porque todo lo que el programador tiene que hacer para solucionar un problema es describirlo vía axiomas y reglas de deducción.

En los lenguajes lógicos se utiliza el formalismo de la lógica de primer orden para representar el conocimiento sobre un problema y para hacer preguntas que, si se demuestra que se pueden deducir a partir del conocimiento dado en forma de axiomas y de las reglas de deducción estipuladas, se vuelven teoremas. Así se encuentran soluciones a problemas formulados como preguntas. Con base en la información expresada dentro de la lógica de primer orden, se formulan las preguntas sobre el dominio del problema y el intérprete del lenguaje lógico trata de encontrar la respuesta automáticamente. El conocimiento sobre el problema se expresa en forma de predicados (axiomas) que establecen relaciones sobre los símbolos que representan los datos del dominio del problema.

Este concepto de programación lógica está ligado históricamente a un lenguaje llamado Prolog, que proviene de *PROgrammation en LOGique* (programación en lógica), que fue el primer lenguaje de programación lógico y el más conocido y utilizado. Este lenguaje fue desarrollado por el Grupo de Inteligencia Artificial de la Universidad de Marseille, dirigido por Alain Colmerauer, en 1972. Prolog es utilizado para el desarrollo de aplicaciones de inteligencia artificial debido a su forma de representar el conocimiento, facilitando las búsquedas en bases de datos, la escritura de compiladores, la construcción de sistemas expertos, el procesamiento de lenguaje natural y la programación automática. También es muy adecuado para las aplicaciones que implican búsqueda de patrones, búsqueda con rastreo inverso o información incompleta. Asimismo es el lenguaje escogido por Japón como piedra angular de los sistemas de computación de quinta generación.

Dice Alain Colmerauer en [1]: "Se suele preguntar cuál es la diferencia entre Prolog y los otros lenguajes de programación y de dónde viene su supuesto poder. Prolog nació de un desafío: crear un lenguaje de muy alto nivel, aun cuando fuera ineficiente para los informáticos de la época. La eficiencia consistía entonces en que una máquina ejecutara muy rápidamente programas laboriosamente escritos. El reto consistía en poder escribir rápidamente los programas dejando a la máquina su laboriosa ejecución.

Una vez liberados de esa obsesión por la eficiencia, se pudo recurrir a los formalismos y mecanismos de inferencia de la lógica matemática; mecanismos ciertamente ineficientes, pero poderosos sin lugar a dudas. De ahí viene el nombre de Prolog “Programación en lógica”. Ahora bien, si se considera lo que sucede a nivel de la máquina, resulta que ésta deberá ser mucho más inteligente de lo normal: debe ser no-determinista, es decir, capaz de explorar muchas posibilidades y debe poder resolver miles y miles de pequeñas ecuaciones.

Estas ecuaciones introducen incógnitas, que no son otra cosa que las variables del programa, pero distan mucho de ser las variables habituales que designan localidades de memoria. El programador que llega a Prolog desde un lenguaje clásico, experimenta una revelación semejante a la del escolar que pasa de la aritmética a los primeros rudimentos del álgebra. Puede representar como incógnitas aquellas entidades cuyos valores busca,

establecer ciertas relaciones entre esas incógnitas y, sin tener que detallarlos, dejar que la máquina considere todos los casos posibles y aportar todas las posibles soluciones...

¿Qué más se puede pedir?"

# 2. LOGICA PROPOSICIONAL

La programación lógica tiene sus orígenes en los trabajos de prueba automática de teoremas. Para esto se utiliza una única regla de inferencia llamada *principio de resolución*[1], mediante la cual la prueba de un teorema puede ser llevada a cabo en forma automática. La resolución es una regla que se aplica sobre las fórmulas surgidas de la **lógica de primer orden** y la demostración de teoremas mediante esta regla de inferencia se lleva a cabo por reducción al absurdo.

La lógica de primer orden o lógica proposicional es uno de los formalismos más utilizados para representar conocimiento en Inteligencia Artificial. Esta lógica es la que utiliza proposiciones y nexos entre éstas para expresar sus verdades. Las proposiciones equivalen a frases u oraciones del lenguaje hablado, mientras que los nexos a través de los cuales puede relacionar estas proposiciones son la conjunción (y), la disyunción (o) y la implicación (si). Cuenta con un lenguaje formal mediante el cual es posible representar fórmulas llamadas axiomas o predicados, que permiten describir fragmentos del conocimiento, y además consta de un conjunto de reglas de inferencia que aplicadas a los axiomas, permiten derivar nuevo conocimiento.

El lenguaje formal de la lógica proposicional o lógica de primer orden es el Lenguaje de Primer Orden (LPO). En realidad éste no es un lenguaje simple, sino que es una familia de lenguajes, donde todos sus miembros tienen una gramática similar y comparten ciertos ítems importantes de su vocabulario. De todos modos nuestro estudio se centrará en un lenguaje genérico de primer orden, que es el que luego podremos aplicar en los programas de Prolog.

Los enunciados más básicos de LPO son los **enunciados atómicos**. Estos se corresponden a los enunciados más simples del español, enunciados que consisten en algunos nombres conectados por algún predicado. Ejemplos de este tipo son *Juan corrió*, *Juan vio a Ana* y *Ana regaló flores a Juan*. En LPO, los enunciados atómicos son formados también combinando nombres (o constantes individuales, tal como suelen llamarse) y predicados, aunque como posteriormente veremos difieren un poco en el modo en que son combinados.

A continuación desarrollaremos algunos conceptos elementales de la lógica.

## 2.1 CONSTANTES INDIVIDUALES

Las constantes individuales son simplemente **símbolos** (nombres) que se usan para referir a algún objeto individual fijo. Por ejemplo, podríamos usar Juan como una constante individual para denotar una persona particular, o 1 como una constante individual para denotar un número particular. En ambos casos, funcionan exactamente como los nombres funcionan en español.

Las constantes individuales hacen referencia exactamente a un objeto en particular, por ejemplo el nombre de Juan en español puede ser usado para hacer referencia a personas diferentes, y podría ser usado dos veces en un enunciado para hacer referencia a dos personas diferentes, pero en LPO esto no es posible, el nombre Juan hace referencia exactamente a un objeto.

[1] Estos trabajos fueron realizados por J. A. Robinson en 1965 y publicados bajo el título de "A machine oriented logic based on the resolution principle" en el Journal of the ACM n1 12. pág. 23-41. Sin embargo, la aplicación del método desarrollado por Robinson genera un gran número de combinaciones posibles para llevar a cabo las resoluciones, por lo que Prolog utiliza en realidad un método más refinado llamado Resolución-SLD.

Resumiendo, en LPO :

- Todo nombre debe referir a un objeto.
- Ningún nombre puede referir a más de un objeto.
- Un objeto puede tener más de un nombre.

## 2.2 SIMBOLOS DE PREDICADO

Los símbolos de predicado son utilizados para denotar alguna propiedad de objetos o alguna relación entre objetos. Como en español, son expresiones que combinadas con nombres, forman enunciados atómicos. Pero no corresponden exactamente a los predicados de la gramática española.

Consideremos en español el siguiente enunciado :

*Juan es padre de Ana*.

En la gramática española esto es analizado como una oración sujeto-predicado. Consiste del sujeto *Juan* seguido del predicado *es padre de Ana*. En el lenguaje de primer orden, por contraste, vemos a esto como una afirmación que involucra dos sujetos lógicos :

los nombres *Juan* y *Ana* (que son constantes individuales)
y un predicado, *es padre de*,

que expresa una relación entre los referentes de los nombres.

Los enunciados del LPO tienen a veces dos o más sujetos lógicos, y el predicado es, por así decirlo, lo demás. Los sujetos lógicos son llamados los **argumentos** del predicado. En este ejemplo se dice que el predicado es binario, puesto que toma dos argumentos.

En español, algunos predicados tienen argumentos opcionales. En este sentido podemos decir *Ana regaló*, *Ana regaló flores*, o *Ana regaló flores a Juan*. Aquí el predicado *regaló* toma uno, dos y tres argumentos respectivamente. Pero en LPO, cada predicado tiene un número fijo de argumentos, una aridad fija. La **aridad** es un número que indica cuántas constantes individuales necesita el símbolo de predicado para formar una oración. Si la aridad de un símbolo de predicado es 1, entonces ese predicado se usará para denotar algunas propiedades de los objetos, y requerirá por consiguiente exactamente un argumento (un nombre) para hacer una afirmación. Por ejemplo, podríamos utilizar el siguiente símbolo de predicado unario:

*Mujer*

para denotar la propiedad de ser mujer. Podríamos posteriormente combinar esto con el nombre

*Ana*

para lograr la expresión

*Mujer(Ana)*,

que expresa la afirmación que Ana es mujer.

Si la aridad de un predicado es 2 o más, entonces este predicado será utilizado para representar una relación entre sus argumentos. De este modo, podríamos usar una expresión de aridad 2 como :

*Mayor(Juan, Ana)*

para expresar una afirmación acerca de Juan y Ana, por ejemplo la afirmación de que Juan es más viejo que Ana. En LPO podemos tener símbolos de predicado con cualquier aridad.

En síntesis, en LPO,

- Todo símbolo de predicado viene con una **aridad** simple fija, un número que le dice cuántos nombres (sujetos) necesita para formar un enunciado atómico.
- Todo predicado es interpretado por una propiedad o una relación determinada de la misma aridad que el predicado.

## 2.3 ENUNCIADOS ATOMICOS

En LPO, las clases más simples de afirmaciones son aquellas que son realizadas con un predicado simple y el número apropiado de constantes individuales. Un enunciado formado por un predicado seguido por el número correcto de nombres es llamado un **enunciado atómico**. Por ejemplo :

*Mujer(Ana)*
*Mayor(Juan, Ana)*

son enunciados atómicos, siempre que los nombres y símbolos de predicados en cuestión sean parte del vocabulario de nuestro lenguaje. Como vemos, en los predicados utilizamos notación prefija: el predicado precede a los argumentos.

El **orden** de los nombres en un enunciado atómico es importante. Así como *María es mayor que Juan* significa algo diferente de *Juan es mayor que María*, en LPO *Mayor(María, Juan)* también tiene un significado diferente que *Mayor(Juan, María)*.

Predicados y nombres hacen referencia respectivamente a propiedades o relaciones y a objetos. Lo que hace a un enunciado especial es que hace afirmaciones (o expresa proposiciones). Una afirmación es algo que es verdadero o falso, al que sea de estos dos casos lo denominamos su **valor de verdad**. En este sentido, si se quiere afirmar que *María* es *mayor* que *Juan,* entonces el enunciado *Mayor(María, Juan)* expresa una verdad cuyo valor de verdad es VERDADERO, mientras que *Mayor(Juan, María)* expresa una verdad cuyo valor de verdad es FALSO. Dada nuestra suposición que los predicados expresan propiedades determinadas y que los nombres denotan individuos definidos, se sigue que cada enunciado atómico del lenguaje de primer orden debe expresar una afirmación que es verdadera o falsa.

Entonces, en el lenguaje de primer orden,

- Los enunciados atómicos se forman colocando un predicado de aridad **n** al frente de **n** nombres (encerrados entre paréntesis y separados por comas).
- El orden de los nombres es crucial cuando se forman enunciados atómicos.

## 2.4 ENUNCIADOS ATOMICOS COMBINADOS

Los predicados constituyen funciones que transforman los argumentos de objeto (constantes individuales) en valores verdaderos o falsos.

Por ejemplo, con la forma normal de interpretar el objeto Albatros y los predicados Plumas y Pájaro se puede decir, de manera informal, que los siguientes son expresiones verdaderas :

Plumas (Albatros)
Pájaro (Albatros)

Suponga, ahora, que usted dice que la siguiente expresión es verdadera :

Plumas (Paloma)

Evidentemente, Paloma es un símbolo que denota algo que tiene plumas, lo que restringe las posibilidades de lo que éste puede ser, ya que Paloma satisface el predicado Plumas.

Se puede expresar otras restricciones con otros predicados, como Vuela y Pone_Huevos. De hecho, es posible limitar los objetos que Paloma puede designar a aquellos objetos que satisfacen ambos predicados al mismo tiempo, al afirmar que las dos expresiones siguientes son verdaderas :

Vuela (Paloma)
Pone_Huevos (Paloma)

Sin embargo, existe una forma más tradicional de expresar esta idea. Simplemente se combinan la primera expresión y la segunda y se dice que la combinación es verdadera.

Vuela (Paloma) y Pone_Huevos (Paloma)

Por supuesto, puede insistir en que Paloma designa algo que satisface uno de los dos predicados. Esta restricción se especifica de la siguiente manera :

Vuela (Paloma) o Pone_Huevos (Paloma)

Sin embargo, en lógica se utiliza una notación diferente. Se escribe *y* como & (ó $\wedge$ ) y *o* como $\vee$. Entonces en lógica las expresiones quedarían como :

Vuela (Paloma) & Pone_Huevos (Paloma)
Vuela (Paloma) $\vee$ Pone_Huevos (Paloma)

Cuando se unen las expresiones con &, forman una conjunción y cada parte se conoce como conjuntante. De manera parecida, cuando las expresiones están unidas por $\vee$, forman una disyunción y cada parte que la forma es un disyundante.

Observe que & y $\vee$ se conocen como **conectivos lógicos** porque transforman combinaciones de falso y verdadero.

Además de los símbolos & y $\vee$, existen otros dos conectivos fundamentales: uno de ellos es el conectivo no, escrito como ~, y el otro es implica, escrito como $\Rightarrow$. Considere lo siguiente:

~Plumas(Toto)

para que esta expresión sea verdadera, Toto debe representar algo para lo que Plumas(Toto) no es verdad. Esto es, Toto debe ser algo para lo cual el predicado Plumas no se satisface.

A continuación, mediante el conectivo $\Rightarrow$, tenemos una expresión que se asemeja mucho a una regla de consecuente-antecedente:

Plumas(Toto) $\Rightarrow$ Pájaro(Toto)

Decir que el valor de esta expresión es verdadero restringe lo que Toto puede representar. Una posibilidad permitida es que Toto es algo para lo cual el Plumas(Toto) y Pájaro(Toto) son verdaderas. Naturalmente, la definición de $\Rightarrow$ permite también que Plumas(Toto) y Pájaro(Toto) sean falsas. Otra posibilidad permitida por la definición de . es que Plumas(Toto) sea falsa y Pájaro(Toto), verdadera. Sin embargo, si Plumas(Toto) es verdadera y Pájaro(Toto) falsa, entonces la implicación es falsa.

## 2.5 LENGUAJES GENERALES DE PRIMER ORDEN

En general, un lenguaje de primer orden es determinado fijando los nombres y predicados que contiene. Cada predicado viene con una aridad especificada. Un lenguaje de primer orden debe tener al menos un símbolo de predicado.

Cuando traduzca un enunciado de español a LPO, tendrá un lenguaje predefinido de primer orden que querrá usar. Su objetivo será lograr una traducción que capture el significado del enunciado original en español tan fielmente como sea posible, dados los nombres y predicados disponibles en el lenguaje de primer orden.

Otras veces, sin embargo usted no tendrá un lenguaje predefinido para usar en su traducción. Si no lo tiene, la primera cosa que tiene que hacer es decidir qué nombres y predicados necesita para su traducción. En efecto, estará diseñando al vuelo, un nuevo lenguaje de primer orden capaz de expresar el enunciado español que quiere traducir. Estuvimos haciendo esto por ejemplo, cuando introdujimos *Mujer(Ana)* como la traducción de Ana es mujer y *Mayor(María, Juan)* como la traducción de María es mayor que Juan.

Cuando toma estas decisiones, existen a veces formas alternativas de proceder. Cada alternativa puede generar como resultado predicados igualmente útiles, pero con distinto nivel de flexibilidad. En general, cuando diseñamos un lenguaje de primer orden

tratamos de economizar predicados introduciendo los predicados más flexibles que se pueda. La resultante de esto es un lenguaje más expresivo, que hace a las relaciones lógicas entre varias afirmaciones más claras.

En cuanto a los nombres, estos pueden ser introducidos en un lenguaje de primer orden para referir a cualquier cosa que pueda ser considerada un objeto. La noción de un objeto es construida más flexiblemente, cubriendo todo aquello que puede hacer afirmaciones.

Diseñar un lenguaje de primer orden con los nombres y predicados correctos requiere alguna habilidad. Usualmente, el objetivo general es lograr un lenguaje que pueda decir todo lo que desee, pero que use el vocabulario más pequeño posible. Seleccionar de manera adecuada los nombres y predicados es la clave para hacerlo.

## 2.6 PREDICADOS CON CONSECUENTE

Una de las mayores preocupaciones de la lógica es el concepto de consecuencia lógica. ¿Cuándo una oración, enunciado o afirmación se sigue lógicamente de otras?

En realidad, una de las principales motivaciones en el LPO fue hacer la relación de consecuencia lógica tan clara como sea posible. Evitando la complejidad y la ambigüedad del lenguaje ordinario, esperamos que las consecuencias de nuestras afirmaciones sean más fácilmente reconocibles.

¿Qué queremos decir con consecuencia lógica? Unos pocos ejemplos ayudarán. Primero, permítasenos decir que un **argumento** es cualquier serie de enunciados en el que uno (llamado conclusión) se sigue o es apoyado por otros (llamados premisas o consecuentes). He aquí un ejemplo:

*Todos los hombres son mortales. Sócrates es un hombre. Por consiguiente Sócrates es mortal.*

Otro ejemplo, esta vez expresado en LPO, podría ser:

*Mujer(A)* y *A=B*.

Entonces ciertamente se sigue que *Mujer(B)*.

Esto también podría escribirse como *Mujer(B) si Mujer(A) y A=B*. ¿Por qué? Porque no hay modo de que las premisas sean verdaderas -que *A* sea *Mujer* y que *B* sea el mismo objeto que *A*- sin que la conclusión sea también verdadera. Nótese que podemos reconocer que este último enunciado es una consecuencia de los dos primeros sin saber que las premisas son en realidad, como cuestión de hecho, verdaderas. Pues la observación crucial es que **si** las premisas son verdaderas **entonces** la conclusión debe ser también verdadera.

Esta descripción de la relación de consecuencia lógica es correcta, tal como van las cosas. Pero no hace todo lo que nos gustaría que haga. En particular, no nos dice cómo **mostrar** que una conclusión dada *S* se sigue, o no, de sus premisas *P, Q, R,...* ¿Qué métodos están disponibles para mostrarnos que una afirmación dada es consecuencia lógica de sus premisas? Aquí la noción clave es la de **prueba**. Una prueba es una demostración paso a paso que una conclusión dada (digamos *S*) se sigue de algunas premisas (digamos *P*,*Q*,*R*).

Consideremos un ejemplo concreto y simple. Supongamos que queremos mostrar que *Sócrates algunas veces está preocupado por la muerte* es una consecuencia lógicas de las cuatro premisas

*Sócrates es un hombre*
*todos los hombres son mortales*
*ningún ser mortal vive para siempre*
*cualquiera que eventualmente muera, algunas veces se preocupa por ello*.

Una prueba de esta conclusión podría pasar a través de los siguientes pasos intermedios. Primero notamos que desde las primeras dos premisas se sigue que *Sócrates es mortal*. A partir de esta conclusión intermedia y la tercera premisa (que *ningún ser*

*mortal vive para siempre*), se sigue que *Sócrates eventualmente morirá*. Pero esto, junto con la cuarta premisa nos da la conclusión deseada: *Sócrates se preocupa por la muerte*.

Entonces el modo en que una prueba obra, es estableciendo una serie de conclusiones intermedias, en donde cada una es una consecuencia obvia de las premisas originales y de las conclusiones intermedias previamente establecidas. La prueba termina cuando finalmente establecemos la conclusión *S* como una consecuencia obvia de las premisas originales y las conclusiones intermedias. Si nuestros pasos individuales son correctos, entonces la prueba muestra que *S* es ciertamente una consecuencia de *P*,*Q*,*R*. Después de todo, si las premisas son todas verdaderas, entonces nuestras conclusiones deben ser también verdaderas. Y en ese caso, nuestra conclusión final debe ser también verdadera.

En pocas palabras podemos decir que

- Una prueba de un enunciado (conclusión) *S* a partir de las premisas *P*, *Q*, *R*,... es una demostración paso a paso que muestra que *S* **debe** ser verdadera en cualquier circunstancia en que las premisas *P*, *Q*, *R*,... son **todas** verdaderas.

# 3. INTRODUCCION A PROLOG

Prolog es un lenguaje de programación declarativo basado en la lógica de primer orden, particularmente en una restricción de la forma clausal de la lógica. Fue desarrollado por Alain Colmerauer en 1972 en la Universidad de Marseille, Francia. Usa como regla de inferencia el "principio de resolución" propuesto por Robinson en 1965. La representación del dominio se realiza a través de hechos y reglas.

Decimos que es declarativo porque no es imperativo. Es decir, cada "línea de programa" Prolog es una declaración, no una orden. Se tiene así un conjunto de aseveraciones expresadas simbólicamente, que expresan conocimientos de una situación real o ficticia. Para esto se usa la lógica de predicados de primer orden que se expuso anteriormente.

Prolog es un lenguaje de programación hecho para *representar y utilizar el conocimiento* que se tiene sobre un determinado dominio. Más exactamente, el dominio es un conjunto de objetos y el conocimiento se representa por un conjunto de relaciones que describen las propiedades de los objetos y sus interrelaciones.

En Prolog, el programa (las reglas que definen las propiedades y relaciones entre los objetos) está muy alejado del modelo Von Newman que posee la máquina en la que tienen que ser interpretados. Debido a esto, la eficiencia en la ejecución no puede ser comparable con la de un programa equivalente escrito en algún lenguaje imperativo o procedural. El beneficio es que aquí ya no es necesario definir el algoritmo de solución, como en la programación imperativa, sino que lo fundamental es expresar bien el conocimiento que se tenga sobre el problema que se esté enfrentando.

Prolog forma su lenguaje a partir de un alfabeto que contiene sólo dos tipos de símbolos:

1. **símbolos lógicos**, entre los que se encuentran los símbolos de constantes proposicionales *true* y *false* (verdadero y falso); los símbolos para la negación, la conjunción, la disyunción y la implicación (que en Prolog se denota con los caracteres :-); los símbolos de cuantificadores; y los símbolos auxiliares de escritura como corchetes [,], paréntesis (,) y coma.
2. **símbolos no lógicos**, agrupados en el conjunto de símbolos constantes; el conjunto de símbolos de variables individuales (identificadores); el conjunto de símbolos de relaciones n-arias; y el conjunto de símbolos de funciones n-arias.

A partir de estos símbolos se construyen las expresiones válidas en el LPO de Prolog: los términos (nombres) y las fórmulas (predicados). Este LPO posee un amplio poder de expresión, ya que los términos permiten hacer referencia (nombrar) todos los objetos del

universo, mientras que las fórmulas (predicados) permiten afirmar o negar propiedades de estos o bien establecer relaciones entre los objetos del universo.

Existen muchas versiones de Prolog, algunas de las más conocidas para PC se muestran en la tabla 1. En Prolog de Edimburgo, versión de Prolog diseñada por David Warren que se ha convertido en estándar, un término es cualquiera de las tres expresiones siguientes: una constante, como el número *"100"*; la palabra *"antonio"* y la letra *"c"*; o una variable, por ejemplo *"X"* (notar que los identificadores que comienzan con mayúscula representan, siempre, variables). En cambio, un predicado atómico o elemental es una expresión de la forma *"R($a_1$, $a_2$,..., $a_n$)"* donde *R* es un símbolo de predicado n-ario que como vimos, denota alguna relación entre objetos o alguna propiedad de un objeto, y $a_1$, $a_2$,..., $a_n$ son términos funcionando como argumentos.

Los conceptos generales de este apunte los explicaremos con Turbo Prolog, basado éste en Prolog de Edimburgo.

**TABLA 1. ALGUNAS VERSIONES DE LENGUAJE PROLOG PARA PC**

| SISTEMA | EMPRESA | ENTORNO |
|---|---|---|
| ALS Prolog | (Applied Logic Systems Inc.) | DOS / Unix / Xenix |
| Arity Prolog | (Arity Corp) | DOS / Windows / OS 2 |
| Cogent Prolog | (Amziod) | DOS |
| IF Prolog | (InterFace Computer Inc.) | UNIX |
| Quintus Prolog | (Quintus Corp.) | DOS / Windows / OS 2 |
| Turbo Prolog | (Borland International) | DOS |
| PIE Prolog | (versión shareware) | DOS |

# 4. CALCULO DE RELACIONES

La programación lógica trabaja más con relaciones que con funciones. Se basa en la premisa de que programar con relaciones es más flexible que programar con funciones, debido a que las relaciones tratan de forma uniforme a los argumentos y a los resultados. De manera informal, las relaciones no tienen sentido de dirección ni prejuicio alguno acerca de qué se calcula a partir de qué.

En Prolog se utiliza sólo un tipo determinado de reglas para definir relaciones, llamadas **cláusulas de Horn**[2], llamadas así en honor al lógico Alfred Horn, quien las estudió. Estas reglas están compuestas por dos partes: el **consecuente** y el **antecedente**. El consecuente, que es la primera parte de la cláusula, es lo que se quiere probar, la conclusión de la regla. El antecedente es la condición que determinará en qué casos el consecuente es verdadero o falso. Esta estructura de cláusula es de la forma:

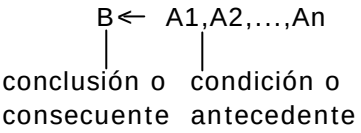
**conclusión *si* condición**

La resolución de una de estas cláusulas podría considerarse como la invocación a un procedimiento, que sería el encargado de comprobar qué valor de verdad posee la condición (o condiciones), para luego asignar el resultado lógico de esa evaluación al objeto que se halla a la izquierda de la implicación *si* (el consecuente o conclusión). Las reglas que gobiernan esta asignación, en el caso de existir más de una condición, son las que rigen la Lógica de Primer Orden, con idénticos operadores y precedencias entre ellos.

Es posible que, para verificar una condición, deban verificarse primeramente una o varias sub-condiciones. Esto equivaldría a la llamada de varios sub-procedimientos para completar la ejecución de un procedimiento padre.

[2] Las cláusulas de Horn son fórmulas (predicados) bien formadas, que se encuentran en forma clausal.

Las cláusulas de Horn permiten crear un lenguaje de primer orden con una sintaxis rígida (se debe respetar siempre la estructura de *antecedente - consecuente*) pero con un gran poder de expresión, ya que nos deja representar todo el conocimiento necesario como cualquier otro LPO.




Estas cláusulas pueden adoptar las siguientes formas:

| | | | |
|---|---|---|---|
| conclusión | B—— | Afirmación | Hecho |
| conclusión si condición | B—— A1,A2,...,An | Implicación | Regla |
| si condición | —— A1,A2,...,An | Negación | Objetivo |

**AFIRMACION**

B —— Afirmación incondicional

Cuando escribimos sólo la conclusión, estamos afirmando algo que no necesita ser probado. Esto significa que estamos diciendo una verdad que no debe ser comprobada, que no tiene condición de valides. Por ejemplo :

mujer (ana)

con lo que decimos que ana posee la propiedad de ser mujer, sin que se deba satisfacer ninguna condición.

Estos casos particulares de cláusulas sin condición son llamados hechos, porque son verdades por sí mismas.

**IMPLICACION**

B—— A1,A2,...,An Afirmación condicional, donde el predicado B es verdadero si A1,A2,...,An son verdaderos conjuntamente.

Cuando escribimos la cláusula completa, estamos escribiendo un hecho condicional, llamado predicado con consecuente en la lógica de primer orden. Con este tipo de estructuras manifestamos que un objeto puede poseer cierta propiedad o que puede existir cierta relación entre objetos si se cumple la condición. Entonces, para expresar la idea de que “Si pedro es hombre entonces pedro es racional”, deberíamos escribir una cláusula como la siguiente :

racional (pedro) si hombre (pedro)

lo que indica que pedro es racional sólo si es hombre.

**NEGACION**

—— A1,A2,...,An Cláusula objetivo o goal

Cada uno de los predicados A1,Aa2,...,An de la cláusula son denominados subobjetivos. Esta es en realidad la cláusula que queremos probar del conjunto de sentencias del programa.

Se deduce que en la prueba del objetivo se esta empleando el principio de resolución de Robinson. Se niega la cláusula que se desea probar y se la agrega a las cláusulas del programa. Luego se aplican las reglas de resolución y, al derivarse una cláusula nula, queda demostrado que la cláusula original es verdadera.

## 4.1 RELACIONES

Resumiendo, el conjunto de reglas que define las relaciones y las propiedades de los objetos es el programa en Prolog. Por ejemplo, según vimos cuando decimos *"Juan es padre de Pablo"* no estamos haciendo más que afirmar que una relación (ser padre) liga dos objetos (designados por sus nombres: Juan y Pablo), y esta relación se puede escribir como:

es_padre_de (pablo, juan)

Igualmente, la respuesta a una pregunta del tipo *"¿Quién es el padre de Pablo?"* se reduce a comprobar si la relación **es_padre_de** liga a *Pablo* con otro objeto, que será la respuesta a la pregunta.

No debemos olvidar que cuando se define una relación entre objetos, el orden que se da a estos es relevante: **es_padre_de (pablo, juan)** puede significar que *"Pablo es padre de Juan"* o que *"Juan es padre de Pablo"*, ya que la relación tiene un solo sentido. Es el programador el que ha de decidir cuál de las dos es la interpretación a usar. Ahora bien, una vez elegida la interpretación, ésta es definitiva.

En el ejemplo anterior se han utilizado identificadores para nombrar a los objetos y a la relación que los liga. En Prolog, el nombre de la relación (**es_padre_de**) se denomina **predicado**, y los nombres que vincula (*Juan* y *Pablo*), **argumentos**.

## 4.2 RELACIONES EN PROLOG

Antes de estudiar la forma en que se construyen las cláusulas en Prolog, veremos cómo se escriben en este lenguaje los operadores lógicos para la conjunción, la disyunción y la implicación:

| Operador | Sintaxis en Prolog |
|---|---|
| AND (conjunción *y*) | , (coma) |
| OR (disyunción *o*) | ; (punto y coma) |
| IF (implicación *si*) | :- (dos puntos y guión) |

Para conocer la forma de trabajar de Prolog, vamos a comenzar viendo un primer ejemplo que describe la carta de un restaurante. Los objetos que interesan aquí son los platos que se pueden consumir y una primer serie de relaciones que clasifica estos platos en entradas, platos a base de carne o de pescado (plato fuerte) y postres. Este menú es un pequeño banco de datos que se expresa de la siguiente manera:

```
entrada(antipasto).
entrada(sopa).
entrada(quesos).

carne(milanesa).
carne(bife_de_chorizo).
carne(pollo_asado).

pescado(congrio).
pescado(pejerrey).

postre(flan).
postre(helado).
postre(fruta).
```

Se podría decir que esto ya es un programa en Prolog (sólo falta cierta estructuración que será vista más adelante). Este programa está compuesto por un conjunto de predicados unarios que definen características de los objetos que poseen como argumentos y los clasifican. Por ejemplo:

```
entrada(antipasto).
```

Indica que antipasto es una entrada y nada más. En realidad este primer tipo de regla se reduce a enunciar los hechos o declaraciones simples. Cuando se tienen estas cláusulas, se pueden hacer preguntas sobre ellas. Una pregunta como:

es una entrada antipasto?

en Prolog se haría de la siguiente manera:

```
:- entrada(antipasto).
```

Nótese que, al contrario que las reglas, una pregunta comienza con :-, que es en Prolog el símbolo de la implicación. Esto se debe a que ésta es una cláusula incompleta que posee sólo el consecuente *si antipasto es entrada*, que es lo que se desea averiguar. A las cláusulas de este tipo se las llama *goals* o *cláusulas objetivo*, ya que son el objetivo, el problema para el que fue escrito el programa.

Se busca entonces si esta afirmación forma parte de las conocidas, la respuesta es *True* (Verdadero), respuesta que indica que sí es verdadera esa afirmación. Sin embargo,

```
:- entrada(ensalada).
```

recibirá una respuesta negativa (*False* o Falso), ya que esta entrada no se encuentra entre las declaraciones de la base.

Estas dos preguntas que realizamos también son llamadas **consultas de existencia**, porque sólo verifican la existencia de hechos y/o relaciones que satisfagan la pregunta formulada.

Si ahora queremos saber cuáles son las entradas que se pueden consumir y no se desea preguntar por separado por las infinitas entradas posibles, esperando una respuesta afirmativa (*True*) o negativa (*False*) en cada caso, entonces podemos preguntar:

¿Cuáles son las entradas?

o lo que es mejor

¿Cuáles son los objetos X que son entradas?

sin conocer al efectuar la pregunta qué objetos representa la X. El símbolo X no designa un objeto en particular, sino todo objeto perteneciente al conjunto (posiblemente vacío) de los que poseen la propiedad de ser una entrada y que se pide sea definido por el programa. En este caso se dice que la X es una **variable**. La pregunta del ejemplo se haría de la siguiente manera:

```
:- entrada(X).
```

Ante esta pregunta Prolog responderá en la pantalla:

```
X=antipasto
X=sopa
X=quesos
```

que es el conjunto de objetos que designa la variable X para que la declaración pregunta se verifique.

Desde el punto de vista sintáctico, una pregunta es una secuencia de uno o más términos (**metas** u **objetivos**) que representa una conjunción de relaciones a satisfacer. Las respuestas a esta pregunta son las restricciones sobre las variables que aparecen en la secuencia de metas y que satisfacen las relaciones consideradas.

Se debe tener en cuenta que en Prolog la variables *siempre* comienzan con una letra mayúscula, mientras que los predicados y los literales deben comenzar con minúsculas. Entonces, las variables son un conjunto de una o más letras, números y guiones bajos (_), que comienzan con una letra mayúscula. En cambio, los predicados y cadenas de literales deben comenzar con una letra minúscula, pudiendo contener, además de letras, números y guiones bajos (_). (En realidad los predicados, y no las constantes literales, también pueden comenzar con mayúsculas, pero en aquí los vamos a escribir siempre en minúsculas.)

A partir de las relaciones que constituyen la base de datos inicial, se pueden construir relaciones más complejas y generales. Por ejemplo, a partir de las relaciones carne y pescado, que indican que su argumento es un objeto que es un plato de carne o un

plato de pescado, se puede definir la relación plato_principal, que indicará que "un plato_principal es un plato de carne o un plato de pescado" y que se formula:

```
plato_principal(P) :- carne(P).
plato_principal(P) :- pescado(P).
```

Aquí aparecen por primera vez las cláusulas de Horn completas, con antecedente y consecuente, que son llamadas *reglas*. Estas reglas se interpretan así:

```
P es un plato_principal si P es un plato de carne.
P es un plato_principal si P es un plato de pescado.
```

lo que indica que un objeto es plato_principal, si es un plato de carne o si es un plato de pescado. Debido a esto, el par de reglas se podría escribir de la siguiente manera

```
plato_principal(P) :- carne(P); pescado(P).
```

aunque continuaremos utilizando la definición inicial en pos de una mayor sencillez.

En estas cláusulas se utiliza la variable P para designar el conjunto de todos los platos de carne, en la primera regla, y todos los platos de pescado, en la segunda.

El ámbito de una variable se limita a la regla en la que está definida, por lo que la variable P de la primera regla de la definición de plato_principal no tiene nada que ver con la variable P de la segunda. En este ejemplo, la pregunta " ¿Cuáles son los platos principales?" formulada como:

```
:- plato_principal(P).
```

produce las respuestas

```
P = milanesa
P = bife_de_chorizo
P = pollo_asado
P = congrio
P = pejerrey
```

Tratamos a continuación la composición de una comida completa. Como es habitual, una comida consta de una entrada, de un plato fuerte (carne o pescado) y de un postre. Una comida es, por tanto, una terna: E,P,D, donde E es una entrada, P un plato y D un postre. Esto se expresa de forma natural por la regla:

```
comida(E,P,D) :- entrada(E), plato_principal(P), postre(D).
```

que se interpreta así: E,P,D satisfacen la relación comida si E satisface la relación entrada, si P satisface la relación plato_principal y D satisface la relación postre.

Por tanto, se ha definido una nueva regla como la conjunción de otras tres relaciones.

A la pregunta "¿Qué comidas hay?", formulada como:

```
:- comida(E,P,D).
```

Prolog responde:

```
E=antipasto, P=milanesa, D=flan
E=antipasto, P=milanesa, D=helado
.....
E=sopa, P=milanesa, D=fruta
E=sopa, P=bife_de_chorizo, D=flan
.....
E=quesos, P=congrio, D=flan
E=quesos, P=congrio, D=helado
.....
E=quesos, P=pejerrey, D=fruta
```

que es la lista de las 54 combinaciones posibles.

Teniendo el mismo conjunto de relaciones definidas, se propone una pregunta un poco más precisa: se desea conocer las comidas que contienen un plato de pescado. Esta pregunta se formula así:

```
:- comida(E,P,D), pescado(P).
```

que expresa la conjunción de las dos condiciones que se quieren verificar. Prolog calculará los valores de las variables E,P,D para los que la primera condición comida(E,P,D) se verifica. Se observa que antes de la evaluación de la relación comida, las variables E,P,D no han recibido todavía ningún valor, lo que no ocurre después de la evaluación. En el momento en el que E,P,D han recibido ciertos valores, por ejemplo:

```
E = antipasto
P = milanesa
D = flan
```

**se procede a evaluar la segunda parte de la pregunta, pescado(P) con el valor asignado a la variable P, entonces, esta selección se ha convertido en:**

```
pescado(milanesa)
```

Al no contener la base ninguna declaración como ésta, el juego de valores propuesto para E,P,D no satisface la pregunta: es un fallo y se intenta con la solución siguiente para comida.

Finalmente, el programa imprime las 18 soluciones posibles:

```
E=antipasto, P=congrio, D=flan
E=antipasto, P=congrio, D=helado
E=antipasto, P=congrio, D=fruta
E=antipasto, P=pejerrey, D=flan
.....
E=sopa, P=congrio, D=flan
E=sopa, P=congrio, D=helado
.....
E=quesos, P=pejerrey, D=fruta
```

Para tener en cuenta:

- Para satisfacer una conjunción de relaciones, se examinan de izquierda a derecha.
- Durante la ejecución, ciertas variables pueden recibir un valor. Si una variable recibe un valor, todas sus apariciones (o más precisamente sus ocurrencias) toman el mismo valor.
- Las variables son locales a la regla en la que aparecen. Cada vez que se utiliza una regla, se tiene una nueva **instancia** de sus variables, exactamente igual como se hace para las variables locales en los lenguajes clásicos de programación. En este caso, se dice que las variables de la regla han sido **renombradas**.
- En una relación no hay distinción entre argumentos de entrada y argumentos de salida. Por tanto, una misma relación puede tratarse de diversas formas: si todos sus argumentos son conocidos sólo hay que verificar si la relación se satisface o no; si algunos de sus argumentos son desconocidos (representados por variables), se calcula el conjunto de valores que se puede asignar a estos argumentos para satisfacer la relación.
- La ejecución es no determinista: se calculan todos los juegos de valores de argumentos que satisfacen la relación.

Este menú podría completarse con la información calórica de cada ración de cada plato, con un predicado que podría tener la siguiente forma:

calorias(<*comida*>, <*valor calórico*>).

que se interpreta como: *"la ración de <comida> aporta <valor calórico> calorías"*. Así, deberá escribirse un hecho para todos y cada uno de los distintos platos que posee el menú (antipasto, sopa, milanesa, pejerrey, flan, etc.), indicando cuántas calorías contiene cada uno, lo que queda de ejercicio para el lector. Con esto se podrá averiguar el valor calórico total de una comida completa. Para esto último se puede definir la relación:

```
valor(E,P,D,V) :- calorias(E,X), calorias(P,Y), calorias(D,Z), V = X+Y+Z.
```

donde V es la suma de las calorías de los componentes de una comida. Para conocer estos valores se pregunta:

```
:- comida(E,P,D), valor(E,P,D,V).
```

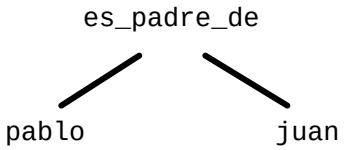
lo que buscaría todas las combinaciones posibles de comidas y calcularía su valor calórico total.

Ahora sería natural definir el concepto de comida equilibrada, cuando el valor calórico de una comida no excede las 800 calorías. La definición de esta relación queda como ejercicio para el alumno.

# 5. BUSCANDO LAS SOLUCIONES

Hasta ahora hemos venido manejando sólo objetos constantes, representados por sus nombres (pejerrey, pollo_asado, etc.). Sin embargo, la estructura que Prolog maneja más naturalmente es el árbol. Las relaciones y sus argumentos se tratan como árboles, debido a que es una estructura suficientemente poderosa para representar informaciones complejas, organizadas jerárquicamente, y de manejo sencillo, tanto desde un punto de vista algebraico como desde un punto de vista informático.

Si representamos en forma de árbol uno de nuestros primeros ejemplos, es_padre_de(pablo, juan) que indica que *Pablo es padre de Juan*, tendríamos algo como:




Esta estructura, como veremos, es también utilizada a la hora de buscar las soluciones a los *goals* de un programa.

## 5.1. EL CONTROL EN PROLOG

Por *control* se entiende la forma en que el lenguaje busca las respuestas a las cláusulas objetivos. En Prolog, el control respeta dos normas:

1. *Orden de metas.* Escoger la meta del extremo izquierdo.
2. *Orden de reglas.* Seleccionar la primera regla aplicable.

¿Qué significa esto? La primera de las normas nos indica que, cuando la cláusula objetivo o *goal* posee más de una regla unidas por conjunciones o disyunciones, éstas se toman de izquierda a derecha para ser resueltas de a una, mediante la aplicación de reglas. Entonces, si volvemos al primer ejemplo del menú de un restaurante, para responder a la pregunta

```
:- comida(E,P,D), pescado(P).
```

Prolog toma en primer lugar la meta del extremo izquierdo (que es comida(E,P,D)) y la resuelve, para tomar luego la segunda de izquierda a derecha (que es pescado(P)) y resolverla, y así sucesivamente hasta terminar con todas las submetas de la cláusula objetivo.

La segunda de las normas enunciadas más arriba quiere decir que, para resolver cada una de las submetas, éstas son reemplazadas por todas y cada una de las reglas de la base de datos que las satisfagan, teniendo en cuenta el orden en que están escritas. Así, por ejemplo, la primera de las submetas que Prolog toma (comida(E,P,D)), es reemplazada por el consecuente de la primer regla que la satisface, que es

```
comida(E,P,D) :- entrada(E), plato_principal(P), postre(D).
```

y sobre ésta se vuelven a aplicar ambas normas, o sea que se toma entrada(E) y luego se la reemplaza por la primer regla que la satisfaga (si la hay), que en este caso sería

```
entrada(antipasto).
```

con lo que E asume un primer valor tentativo (antipasto) que puede ser o no válido, dependiendo de todas las otras partes del *goal*. A la forma en que se van realizando los reemplazos la veremos con mayor detalle a continuación, y es la técnica conocida como *backtracking* o 'vuelta atrás'.

Concluyendo, podemos decir que si se tienen en cuenta las dos normas con que se maneja Prolog para encontrar las soluciones, se advierte que la respuesta a una pregunta se ve afectada por el orden de las metas dentro de la consulta y por el orden de las cláusulas dentro de la base de datos de hechos y reglas. Es muy importante tener esto en cuenta ya que, debido a un orden erróneo en la declaración de las reglas o a una pregunta mal formulada, se pueden producir ciclos infinitos.

## 5.2. EL BACKTRACKING

Para obtener las soluciones a las cláusulas objetivo solicitadas, Prolog utiliza una técnica de borrado que consiste en reemplazar cada submeta por los consecuentes de todas las reglas de la base que la satisfacen. Para poder hacer esto utiliza el **backtracking**, que es el mecanismo para la búsqueda de datos de Prolog, totalmente invisible para el usuario. De esta forma se verifica si existe cierto hecho determinado o si se cumple algún *goal*. Este mecanismo consiste en recorrer reiteradamente las cláusulas, tratando de establecer si el objetivo actual puede considerarse como verdadero o falso.

Entonces, si poseemos tres reglas P, Q y R, que son de la siguiente manera:

```
P :- p1, p2,..., pm.        (m>=0)
P :- a1, a2,..., an.        (n>=0)
Q :- q1, q2,..., qs.        (s>=0)
R :- r1, r2,..., rt.        (t>=0)
```

donde P está definida de dos formas (recordar la definición de plato_principal), y realizamos la pregunta

```
:- P, Q, R.
```

Prolog busca la solución, aplicando el borrado de las submetas, respetando las dos normas del control indicadas más arriba. Así, toma la primera submeta de la izquierda (P) y la borra, esto quiere decir que la reemplaza por la primera regla de la base que la satisfaga, con lo que nuestra pregunta original será ahora:

```
:- p1, p2,..., pm, Q, R.
```

Con esto se borra P, y se continúa realizando lo mismo con las submetas del nuevo objetivo, hasta que no quede nada para borrar o hasta que se llegue a algo que no puede ser borrado. Esto ocurre cuando se llega a un hecho o cuando se alcanza una contradicción. De este modo, se borraría $p_1$, luego $p_2$, y así hasta terminar.

Sin embargo, todavía quedan posibilidades abiertas que no fueron tenidas en cuenta durante el proceso, como ser la segunda definición de P. Entonces se *vuelve atrás*, se reconsidera la última elección y se intenta borrar el término en cuestión de otra manera para tratar de obtener otra respuesta a la pregunta. Entonces, se toma la segunda definición de la regla P de la base de reglas (P :- $a_1$, $a_2$,..., $a_n$.) obteniendo, en este caso,

```
:- a1, a2,..., an, Q, R.
```

para borrar luego nuevamente todas las submetas del nuevo objetivo, incluyendo Q y R. Se continúa así hasta que no quede ningún punto de elección sin tener en cuenta.

Es importante observar que en el momento en que se efectúa una nueva elección para intentar borrar una submeta, se deshacen todas las asignaciones y modificaciones realizadas a las demás submetas, entre la elección anterior y el momento actual. Esta vuelta atrás o retroceso es lo que se denomina **backtracking**.

Para comprender más claramente este funcionamiento, veremos un ejemplo. Retomaremos el caso del menú del restaurante donde poseemos, entre otras, las siguientes reglas:

```
R1    carne(milanesa).
R2    carne(bife_de_chorizo).
R3    carne(pollo_asado).

R4    pescado(congrio).
R5    pescado(pejerrey).

R6    plato_principal(P) :- carne(P).
R7    plato_principal(P) :- pescado(P).
```

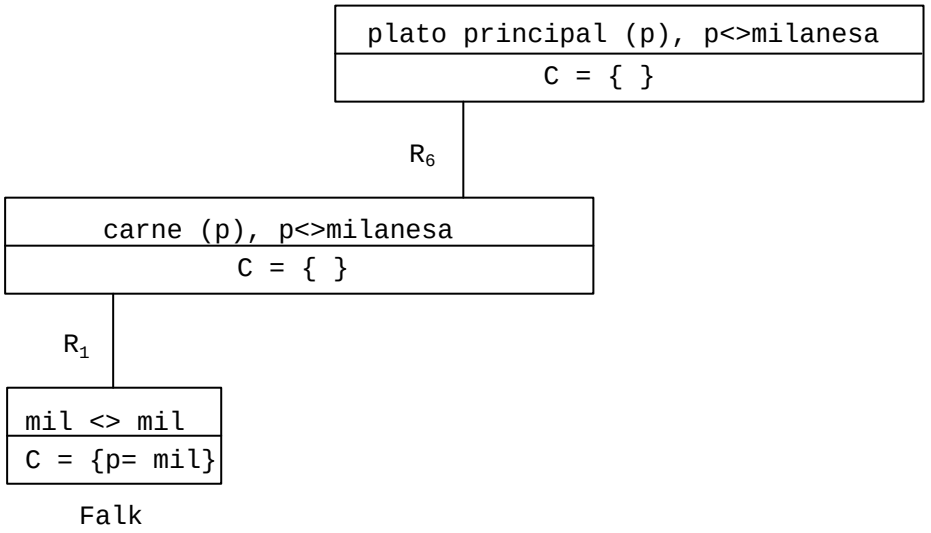
donde $R_1$,..., $R_7$ sólo se agregan para numerar las reglas, y realizaremos la siguiente pregunta (conjunto inicial de metas):

```
:- plato_principal(P), P<>milanesa.
```

que equivale a decir cuáles son los *platos_principales* que no son *milanesa*. Se debe tener en cuenta que Prolog trata a todos los operadores (incluyendo a <> -distinto a-) como reglas comunes, por lo que este *goal* posee dos reglas unidas por una conjunción (*y*).

El proceso de resolución será representado mediante un árbol, en el cual:

- A cada nodo se le asocia el conjunto actual de las metas a borrar y el conjunto de restricciones actuales *C*;
- A cada rama se le asocia la regla elegida para borrar la primer submeta del nodo superior;
- Los sucesores de un nodo son los nuevos conjuntos de objetivos generados por el borrado del primero de ellos de la lista asociada al nodo considerado.




Esta rama es la que queda formada apenas Prolog comienza el análisis de la pregunta. Lo primero que se hace es, partiendo desde el nodo de más arriba hacia abajo, borrar plato_principal(P), reemplazándolo por el consecuente de la primera regla que lo satisface, que es la regla $R_6$. O sea que se permuta el antecedente plato_principal(P) por su definición relacional:

```
carne(P)
```

Este borrado de la primer submeta cambia nuestra pregunta inicial, que es lo que vemos en el segundo nodo, siendo ahora el objetivo:

```
:- carne(P), P<>milanesa.
```

Ahora debe borrarse nuevamente la primer submeta de este nuevo *goal*. Entonces se reemplaza carne(P) por la regla $R_1$, que es la primera que la satisface. Llegamos así al tercer nodo donde se presenta una restricción, ya que la cláusula $R_1$ asigna a la variable P el valor milanesa. En este nodo nos queda la nueva cláusula objetivo

```
:- P<>milanesa.
```

En esta nueva meta vemos que ya no aparece nada de la primer submeta que hemos borrado, como sucedió al pasar del nodo inicial al segundo nodo por la rama $R_6$. Esto

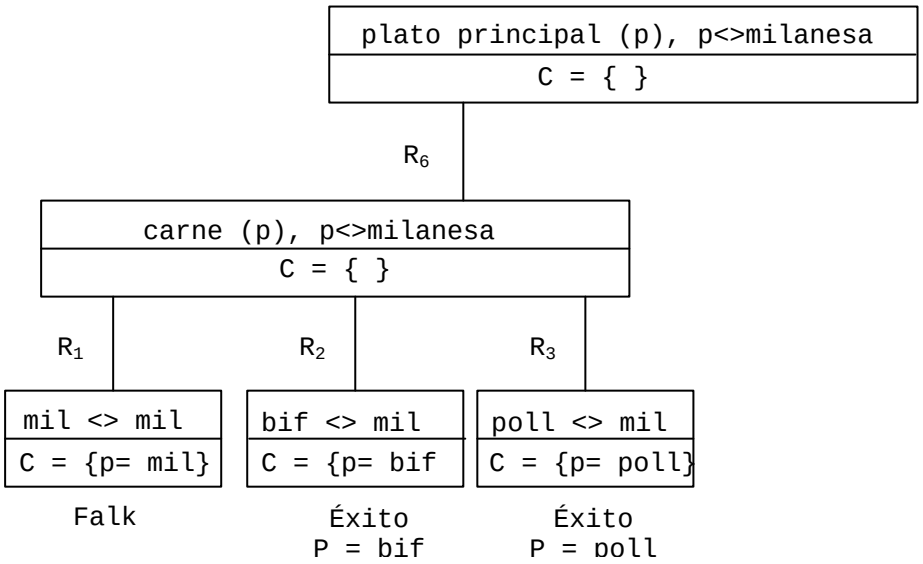
se debe a que llegamos a un hecho ($R_1$) que se borra dejando únicamente la restricción ya indicada. Entonces, respetando dicha restricción, P toma el valor milanesa y obtenemos:

```
:- milanesa<>milanesa.
```

abreviado en el gráfico como mil<>mil, que al borrarse (solucionarse) produce un fallo.

Esto es lo que ocurre al realizarse los borrados de las submetas. Sin embargo, todavía no se consideraron otras reglas que podían reemplazar a las submetas borradas, ya que la base posee dos reglas que definen el *plato principal* y tres que definen el plato de *carne*. Aquí es donde aparece el **backtracking** (retroceso).

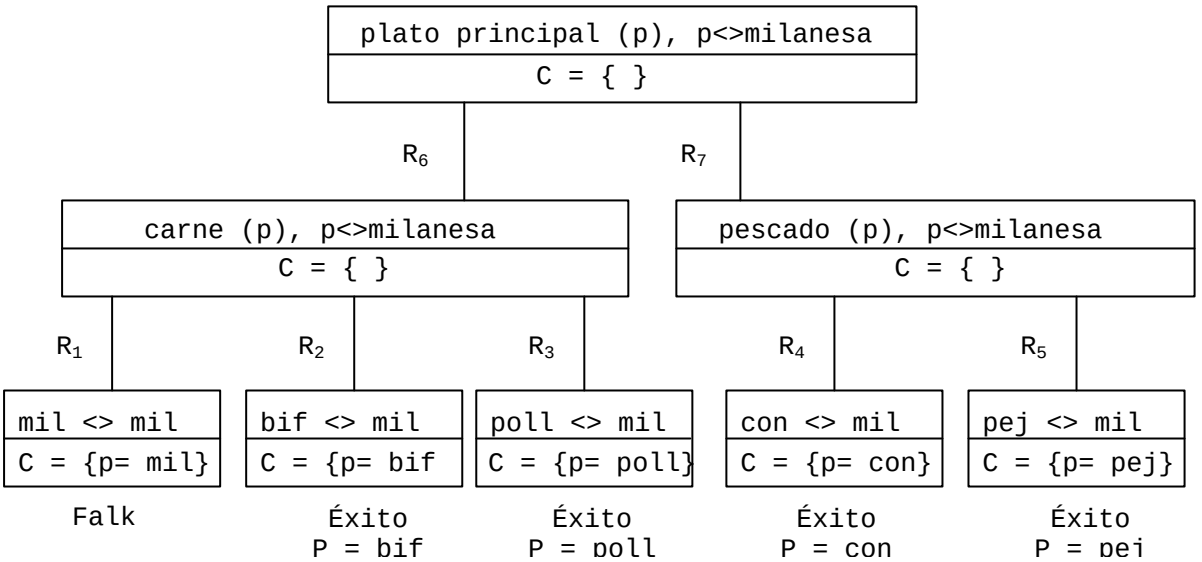
Como la última regla que se borró, teniendo otras posibilidades, fue carne(P), se realiza el *retroceso* subiendo un nodo desde el tercero y se vuelve a borrar esta submeta utilizando las otras opciones. En primera instancia se toma $R_2$ que es la regla que sigue en la base, para luego repetir el proceso tomando $R_3$. Estos borrados nos determinan el siguiente árbol:




Este segundo árbol es el que queda luego de hacer el backtracking para todas las diferentes definiciones del predicado carne( ). Aquí observamos que se presentaron dos éxitos, dos valores de P que son válidos. Estos valores serán los que Prolog nos irá mostrando en la ventana de resultados, de la siguiente manera:

```
P = bife_de_chorizo
P = pollo_asado
```

Después de este retroceso, Prolog intenta borrar de una manera diferente todas las submetas borradas en pasos anteriores. En este ejemplo, se puede utilizar la segunda definición de plato_principal( )para realizar el backtracking. Así se amplía el árbol para llegar finalmente a :




En esta nueva rama, se utiliza $R_7$ para borrar la primer submeta de la cláusula objetivo original, y se borra la nueva submeta de las dos maneras posibles ($R_4$ y $R_5$).

Hemos encontrado así, a través de esta técnica, las cuatro soluciones posibles que se verán como

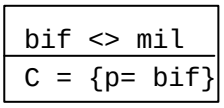
```
P = bife_de_chorizo
P = pollo_asado
P = congrio
P = pejerrey
```

Para finalizar, se debe decir que cada rama del árbol se sigue extendiendo hacia abajo hasta que ocurre alguna de las siguientes cosas:

- La cláusula objetivo del nodo actual está vacía. En este caso nos encontramos frente a un éxito y la respuesta es la restricción actual *C*.
- Se llega a alguna situación donde una submeta no puede ser borrada (solucionada), por no existir una regla que lo permita. Esto es un fracaso.

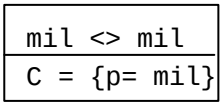
Ante ninguna de estas situaciones se pararía el proceso ya que éste se detiene únicamente cuando se han revisado todas las opciones posibles de borrado. Lo único que provocan es el abandono de la rama donde se presentó dicha circunstancia, con un éxito o un fallo, y el intento de abrir otra rama mediante el backtracking.

Un ejemplo del primero de estos casos es el nodo




donde, al borrar la meta bife_de_chorizo<>milanesa, nos queda el *goal* vacío, siendo esto un éxito cuya solución es la restricción *C*.

Como ejemplo para el segundo caso podemos tomar el nodo




que tiene una meta que no se puede borrar, ya que el operador <> sólo se borra cuando sus argumentos son distintos. Entonces, como tenemos un objetivo que no podemos borrar, estamos frente a un fallo y se debe abandonar esta rama del árbol e intentar por otra utilizando el backtracking.

## 5.3. EL CORTE (!)

Hemos visto que el borrado de un término del *goal* se hace en forma no determinista. Esto significa que se tienen en reserva en todo momento los diversos puntos de elección que se superaron para un posible retroceso posterior. La introducción del corte (representado en Turbo Prolog por un *!*) en una regla permite suprimir algunos de estos puntos de elección, e incluso, hacer un programa enteramente determinista.

Para comprender esto se deben tener en cuenta dos puntos

- Cada vez que se borra una meta, ésta se coloca en la secuencia de puntos de elección, para poder realizar los retrocesos.
- El borrado de la meta del corte "*!*' " devuelve verdadero y suprime todos los puntos de elección que estaban almacenados, impidiendo el retroceso de las submetas anteriores a su aparición.

Por ejemplo, si encerramos entre corchetes los puntos de elección que vamos superando y al lado de éstos colocamos la lista de metas actual, al caso ya visto lo podemos modificar agregándole un corte como última meta

```
:- plato_principal(P), P<>milanesa, !.
```

cuya secuencia de elecciones y metas a borrar será

```
[] plato_principal(P), P<>milanesa, !.
```

borrando la primer submeta por medio de $R_6$ obtenemos un punto de elección y otra submeta

```
[plato_principal(P)] carne(P), P<>milanesa, !.
```

por medio de $R_1$ borramos carne(P) y obtenemos

```
[plato_principal(P) carne(P)] milanesa<>milanesa, !.
```

lo que provoca un fallo y se retrocede al punto de elección precedente, quitándoselo de la lista

```
[plato_principal(P)] carne(P), P<>milanesa, !.
```

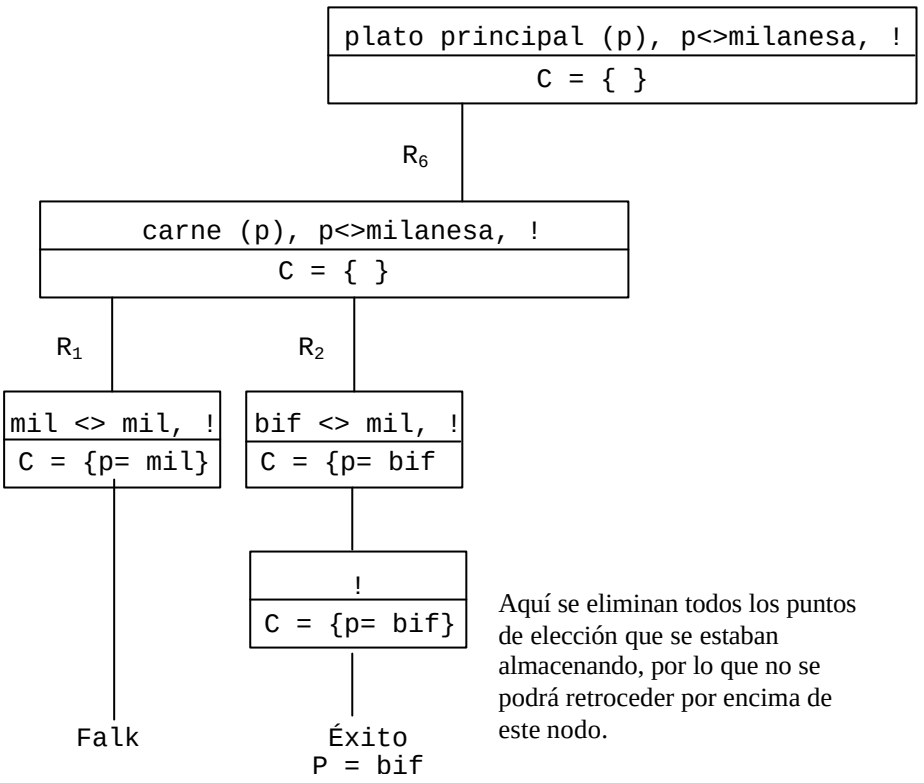
se vuelve a borrar carne(P), esta vez por medio de $R_2$, obteniendo

```
[plato_principal(P) carne(P)] bife_de_chorizo<>milanesa, !.
```

ahora se borra el operador < > sin introducir nuevas elecciones, quedando sólo

```
[plato_principal(P), carne(P)] !.
```

El *!* se borra suprimiendo todas las elecciones en espera, no queda nada para borrar y, por haberse eliminado todos los puntos de elección que había, no es posible realizar ningún retroceso. Por consiguiente, el árbol anterior se convierte en :

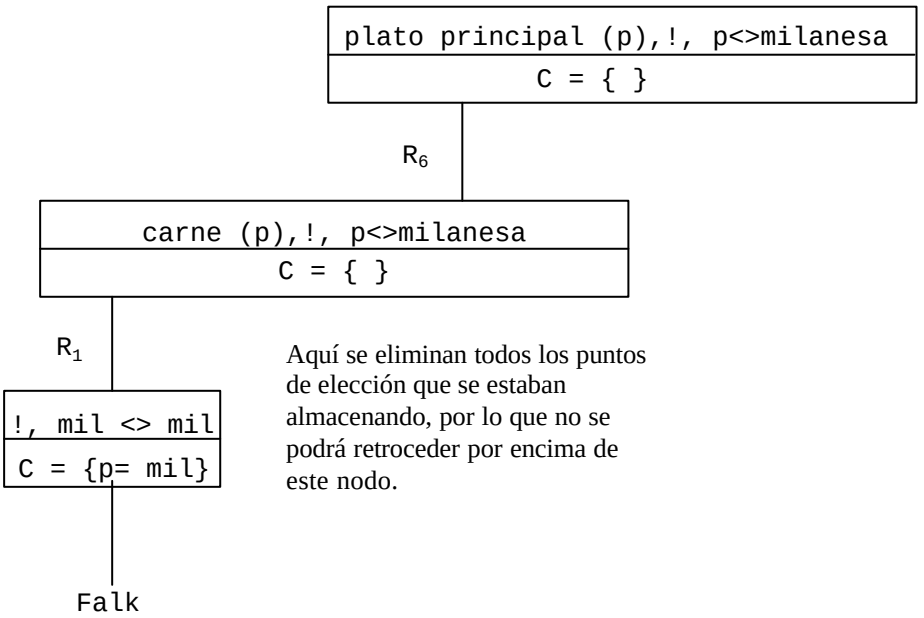



El borrado del *!* tiene el efecto de suprimir todas las elecciones en espera, por lo que no se produce más de una respuesta a la pregunta, que es la primera obtenida.

Si el corte en vez de ponerse al final se hubiera ubicado en otro lugar, por ejemplo

```
:- plato_principal(P), !, P<>milanesa.
```

el árbol tendría la siguiente forma:




Cuando se llega al *!* se eliminan todos los puntos de elección que se almacenaban para poder hacer los retrocesos, por lo que no se pueden buscar más soluciones y no se encuentra ninguna respuesta que satisfaga la pregunta. El resultado devuelto por esta consulta será *False* (Falso).

## 5.4. LA NEGACIÓN COMO FRACASO

Para comprender cómo trabaja la negación en Prolog, se debe tener en cuenta que las reglas y el borrado pueden ser vistos de la siguiente manera:

- Las reglas de un programa son un conjunto de definiciones.
- Un objetivo que se borra es una fórmula verdadera en relación con estas definiciones.

En efecto, borrar un objetivo equivale a demostrar que, para determinados valores de las variables, este objetivo se verifica. Por lo tanto, una consulta como not(P), donde not() es el operador de negación del Prolog de Edimburgo, se trata como verdadera si el sistema fracasa en borrar P, o sea que equivale a decir *'Si no puedo probarlo debe ser falso'*. Esto significa que no existe la definición de P para el conjunto de valores que posee como argumentos y por lo tanto no puede ser borrado.

Debe tenerse en cuenta que not() puede utilizarse únicamente para verificar valores conocidos o que se conocían antes de aplicar el not().

## 5.5. REALIZANDO EL SEGUIMIENTO DE UN PROGRAMA

Turbo Prolog posee cuatro ventanas, una de las cuales está titulada como **Trace**. Esta ventana sirve para realizar el seguimiento de los pasos que realiza Prolog durante la resolución de las cláusulas objetivos planteadas.

Si deseamos seguir paso a paso o realizar la traza de un programa, agregamos la instrucción Trace como primer instrucción al comienzo del código. De esta forma, el programa se parará después de cada paso realizado y nos mostrará en esta ventana lo que se ha realizado, indicando las llamadas a las distintas reglas, los valores retornados y los puntos donde se realiza el retroceso. Para probar esto, realice el seguimiento de los ejemplos utilizados para explicar el backtracking y el corte.

# 6. ESTRUCTURA DE UN PROGRAMA EN PROLOG

Un programa en turbo-Prolog consta de cuatro secciones: “domains”, “predicates”, ”goal” y ”clauses”. Cabe aclarar que todas estas son palabras reservadas, y que toda otra palabra menos las variables deben escribirse en minúsculas.

**DOMAINS (dominio)** : Aquí se definen los objetos y los tipos de datos correspondientes que usaremos en las definiciones posteriores:

**domains**
**objeto** = tipo de dato

**Tipos de datos** : Existen cinco tipos predefinidos :

- **symbol** : Hay dos tipos de símbolos :

  1. Una secuencia de letras, números o caracteres de subrayado en la cual la primera letra es minúscula. Ej. : tiene_lindas_piernas.

     Una secuencia de caracteres encerrados por comillas dobles (“) usada en el caso que el símbolo contenga espacios o no comience con minúsculas. Ej. : “Una persona es trabajadora”.

- **char** : Acepta cualquier carácter, se representa encerrado entre comillas simples (‘) y consta de un solo carácter, por ejemplo ‘A’, ‘2’, ‘/’.
- **integer** : Acepta números enteros en el rango de -32768 al 32767.

- **real** : Acepta números reales, pueden contener signo, punto decimal y varios dígitos decimales. También pueden tener una parte exponencial, pudiendo abarcar números desde +1e-307 a +1e+308, por ejemplo : 427054, -25000, 86.25, -8.525e-203 o - 8411.25658545.
- **string** : Acepta una secuencia de caracteres encerrados entre comillas dobles (“), por ejemplo : “esto también es un string”. La diferencia entre el tipo symbol en 2, y los strings es la forma de representación interna de cada uno. El almacenamiento de los símbolos está implementado de tal manera que su búsqueda en las tablas de memoria es más rápida. Además, los símbolos, tienen problemas para hacer inserciones en tiempo de ejecución. Su uso estará determinado por el tipo de aplicación que se realice.

## Por ejemplo :

```
domains
      persona = symbol     “persona” es un objeto de tipo symbol.
```

**PREDICATES (predicados)** : En esta sección se definen como serán las relaciones entre los objetos del dominio (domains) y el valor que se les asignará en las cláusulas (clauses). En una relación no se puede poner nada que no sea del tipo definido para ella, pues el compilador Turbo Prolog dará error al comprobar tipos distintos. Por ejemplo :

```
predicates

      sabe (persona)       Aquí especificamos que el predicado “sabe” tiene un
                           argumento: “persona”, que a su vez en el dominio está
                           declarado como “symbol”.
      inteligente (persona)
      trabaja (persona)
```

En este ejemplo hemos definido un objeto como símbolo y luego lo relacionamos con el predicado “sabe”. También podemos definir un predicado solo con relacionar tipos predefinidos, como por ejemplo :

```
predicates

      factorial (integer, real)  Esto hace que la relación “factorial” se
                                 establezca entre 2 objetos, uno de tipo entero
                                 y otro de tipo real.
```

**GOAL (meta u objetivo a buscar)** : En esta sección es donde se indica explícitamente cuál es el objetivo o propósito del programa. Hay dos formas básicas de trabajar en Prolog :

- **Goal interno** : Especificando en el programa en su sección “Goal” un hecho a verificar. En este caso el mecanismo de búsqueda se detiene al encontrar el primer valor que lo cumpla, necesitando una indicación explícita de impresión en pantalla para su representación o visualización, si no está esa indicación, responde sólo con un mensaje de “True” o “False”, o no imprime mensajes.
- **Goal externo** : Trabajando a través de la ventana de diálogo sin la sección “Goal” en el programa, verificando y consultando hechos en forma interactiva. En este caso el mecanismo entrega todos los valores que lo verifiquen, y la búsqueda se detendrá sólo al agotarse las cláusulas pertinentes. No necesita de órdenes especiales para visualizar los valores de variables hallados.

Trabajando de cualquiera de las dos formas los objetivos podrán ser tan complicados como información se tenga en las cláusulas.

Siguiendo con el ejemplo, en nuestro caso el objetivo será encontrar una persona que trabaja, da acuerdo al criterio de determinación para ello que se establecerá en la sección Clauses.

```
      trabaja (X) and
      write (“La persona que trabaja es”, X) and n1
```

Aquí vemos que hay algunos predicados que nosotros no definimos, como ser "write" y "n1". Ambos pertenecen a los llamados **predicados predefinidos** y son propios del lenguaje Prolog.

**write** hace lo mismo que idéntica instrucción en Pascal o el "print" de BASIC : imprime un texto en la pantalla.

**n1** imprime solamente una secuencia de fin de línea.

El propósito de todo el conjunto es evaluar "trabaja (X)" e imprimir el resultado de ello con el mensaje entre comillas.

Las respuestas se darán a través de la ventana de diálogos o se deberá armar una interfaz diferente.

**CLAUSES (Cláusulas o definiciones de reglas y hechos)** : Aquí se definen las reglas y hechos que evaluará Turbo Prolog para encontrar las soluciones que se piden en Goal o por la ventana de Diálogos. Poniendo en castellano qué valores asignamos a los predicados de nuestro ejemplo :

José sabe Ana sabe trabaja el que sabe

y en turbo prolog :

```
clauses
       sabe (José)                      hecho
       sabe (Ana)                       hecho
       trabaja (X) if sabe (X)          regla
```

trabaja (X) if sabe (X) se trata de una regla. Las reglas pueden también tener más de una condición para verificarse. Si por ejemplo nosotros ahora decidiéramos que una persona para trabajar debe ser inteligente además de saber, reemplazaríamos la anterior por la siguiente regla :

```
trabaja (X) if sabe (X) and inteligente (X)
```

Veamos el ejemplo completo, escribiendo un programa en Prolog en el que determinaremos como tiene que ser una persona para trabajar en una oficina :

```
domains
       persona = symbol
predicates
       sabe (persona)
       trabaja (persona)
       inteligente (persona)
       tiene_lindas_piernas (persona)
goal
       trabaja (X) and
       write ("La persona que trabaja es ",X) and n1
clauses
       sabe (José)
       sabe (Ana)
       inteligente (Ana)
       inteligente (Carlos)
       inteligente (Silvia)
       tiene_lindas_piernas (Silvia)
       trabaja (X) if sabe (X)
```

Si cargamos en el editor el programa anterior y lo compilamos, inmediatamente aparecerá en la ventana de diálogos el siguiente mensaje :

```
La persona que trabaja es José
Press the space bar
```

Presionando la barra espaciadora volverá al menú principal.

Si en nuestro ejemplo no existiera la sección "Goals", tras compilar el programa se activaría la ventana de diálogos en que pondremos los hechos que queremos comprobar o consultar. Nos aparecerá en esa ventana lo siguiente :

```
Goal :_
```

Si nosotros ponemos este objetivo para nuestro ejemplo :

```
inteligente (Carlos)
```

y presionamos ENTER es como si hubiéramos preguntado “es Carlos inteligente”, o para ser mas exactos con la terminología de la programación lógica “el objeto Carlos cumple la relación inteligente”. Como existe en la parte de las cláusulas un hecho idéntico, inmediatamente veremos como respuesta la palabra “TRUE” y volverá a pedirnos otro Goal.

Si ponemos ahora : inteligente (José) la respuesta será “False” pues buscará en las cláusulas esa relación y no la encontrará.

Podemos poner además por ejemplo preguntas como :

```
sabe (Ana)
tiene_lindas_piernas (Ana)
inteligente (Silvia)
```

Estas preguntas pueden efectuarse siempre y cuando exista esa relación definida en la sección de predicados y tenga también al menos una cláusula con ese nombre y cantidad de argumentos. Todas las preguntas de este tipo se llaman “***objetivos simples de verificación de hechos***” pues su respuesta es “true” o “false”.

## Recordando que en Prolog toda palabra que tiene la primera letra en mayúscula es una variable, veamos que ocurre si preguntamos lo siguiente :

```
trabaja (Quien)
```

La respuesta será entonces :

```
Quien = José
Quien = Ana
2 solutions.
```

La diferencia en la respuesta se debe a la diferencia entre los objetivos internos y externos, como ya vimos: Si el objetivo está incluido en el programa (en la sección Goal), el mecanismo de búsqueda se detiene al encontrar el primer valor que lo cumpla. Si el objetivo es exterior al programa (ingresando por ventana Dialogo), la búsqueda se detendrá sólo al agotarse las cláusulas pertinentes.

En el caso del objetivo “sabe (Quien)” la mecánica es la siguiente : Prolog busca si la relación está declarada en los predicados, luego ve que tenga igual cantidad de argumentos y asume que “Quien” es un símbolo por definición de la relación ; va entonces a las cláusulas buscando aquellas de mismo nombre de relación. Entonces Prolog instancia “Quien” al valor de la cláusula. En otras palabras : la variable “Quien” asume por primera vez el valor del argumento de la cláusula (llamado **vinculación de un valor a una variable o “binding”**), e informa a través de la ventana de diálogos que ha encontrado una instancia de la pregunta, informa de ella y el valor temporal que ha asumido la variable y continua recorriendo las cláusulas hasta el final. Prolog devolverá así uno tras otro todos los casos que hagan verdad al objetivo, y su número. En el caso que ninguna instancia cumpla con los hechos de las cláusulas, Prolog devolverá “False”.

Vemos así la importancia del orden en que se registran las cláusulas en un programa Prolog. Si hubiéramos invertido el orden de los hechos en el programa original, la respuesta hubiera sido :

```
La persona que trabaja es Ana
```

Hagamos ahora la siguiente pregunta :

```
inteligente (Ana) and sabe (Ana)
```

Aquí tenemos una pregunta de dos términos, inteligente (Ana) y sabe (Ana), unidos por una operación lógica de multiplicación , “and”. La pregunta en lógica es : “¿ cumple el objeto Ana las relaciones inteligente y sabe ?”. Prolog primero ve si se cumple la primera parte de la pregunta, si da verdadero (True), ve si se cumple la segunda y devuelve el

resultado de la multiplicación lógica (and). En nuestro caso, al cumplirse las dos da “True”. Si uno de los dos términos da falso la respuesta es “False”

Otros operadores lógicos son el “or” que nos da la suma lógica, y el “not”, negación lógica. Todas las preguntas de este tipo las llamaremos “Objetivos compuestos de verificación de hechos”.

Ejemplos de objetivos de verificación compuestos que dan “True” para nuestro caso en estudio son, entre otros :

```
inteligente (Carlos) and inteligente (Silvia)
trabaja (José) o trabaja (Carlos)
tiene_lindas_piernas (Silvia) and (not sabe (Silvia))
```

## Formulemos a continuación la siguiente pregunta :

```
inteligente (X) and not (sabe (X))
```

Prolog tomará el primer término y buscará el primer valor del primer objeto que cumpla esta condición y lo reemplazará en el segundo término. Si comprueba que también se cumple devolverá “True”.

En otras palabras : Prolog busca la primera cláusula con nombre “inteligente” y reemplaza X con el valor del argumento de esta cláusula, que en este caso es “Carlos”. Luego trata de comprobar la pregunta “not (sabe (Carlos))” ; comprueba “sabe (Carlos)” , que por no existir da “False”, y tras aplicar “not” la respuesta será “True”. Este valor, junto con la respuesta del primer término que da “True”, verifica la regla y devuelve la instancia X con el valor asignado temporalmente “Carlos”.

Vemos que la diferencia entre reglas y hechos es que mientras los hechos describen una situación clara, que no necesita comprobarse, las reglas por lo general son objetivos compuestos que deben verificarse. Por ejemplo las siguientes reglas :

```
jefe (X) if inteligente (X) and not (trabaja (X))
jefe (X) if inteligente (X) and not (sabe (X))
```

Ambas describen la misma situación : “X es jefe si es inteligente y no trabaja o no sabe”, ya que si no sabe no trabaja.

Tomemos un segundo ejemplo para otra serie de explicaciones.

```
domains
       marca, color = symbol
       modelo       = integer
       precio,
       kilometraje  = real
predicates
       auto (marca, kilometraje, modelo, color, precio)
clauses
       auto (peugeot_404,90000,1980,azul,15000)
       auto (ford_taunus,13800,1986,rojo,30000)
       auto (renault_12,90000,1984,gris,25000)
       auto (dodge_1500,13000,1983,rojo,12000)
       auto (volkswagen,39000,1985,rojo,20000)
```

Podemos ver que hemos relacionado mas de un elemento en un predicado. Así un auto tiene marca, kilometraje, modelo, color y precio. Con este ejemplo veremos mejor el tratamiento de variables y objetivos compuestos.

Carguemos en memoria y compilemos este programa, y luego formulemos la siguiente pregunta :

¿ Cuales son los datos de los autos cuyo precio es menor a 20000 ?

En Prolog puede escribirse así :

```
auto (Marca, Kilómetros, Modelos, Color, Precio) and Precio < 20000
```

Como se ve, Marca, Kilómetros, Modelo, Color y Precio son variables por comenzar con mayúsculas. ¿ Cómo funciona Prolog aquí ?. Toma la primera cláusula del predicado “auto” e instancia cada una de las variables a su correspondiente valor. De no ser

compuesta la pregunta nos devolvería los valores hallados, pero en este caso comprobará si el Precio en esta cláusula es menor a 20000. De cumplirse emitirá la lista de datos, en caso contrario no ; pero de cualquier modo seguirá con la cláusula siguiente con la misma operación hasta el fin de las cláusulas.

La respuesta será :

```
Marca = peugeot_404 Kilómetros = 90000  Modelo = 1980Color = azul
Marca = dodge_1500  Kilómetros = 13000  Modelo = 1983Color = rojo
```

Analicemos la forma en que expresamos el objetivo. Aquí utilizamos un objetivo compuesto que consta de dos condiciones : un predicado que debe verificarse comprobando los hechos de la sección cláusulas, y una variable que está limitada en los valores posibles que puede tomar. Esta es una de las formas más comunes y útiles de utilizar las variables para especificar condiciones adicionales para un objetivo, sin apelar a otros predicados o a formular reglas innecesariamente complejas.

Ahora bien, ¿ qué puedo hacer si lo único que quiero saber es la Marca de los autos cuyo precio es mayor a 215000 ? Aquí aparecen las llamadas **variables anónimas** representadas con el signo de subrayado ("_") y significa que el argumento con la variable anónima no interesa y puede contener cualquier tipo de dato. Entonces la pregunta será la siguiente :

```
auto (Marca,_,_,_,Precio) and Precio > 21500
```

La respuesta de Prolog será :

```
Marca = ford_taunus Precio = 30000
Marca = renault_12  Precio = 25000
```

**Donde nos devuelve sólo los valores que nos interesan, obviando los demás parámetros y sus valores.**

Debemos aquí hacer una advertencia sobre el uso de las llamadas "**variables nombradas**". Estas variables son posicionales, es decir que su valor dependerá del lugar que ocupan entre los argumentos del predicado. Si preguntásemos :

```
auto (Marca,Precio,_,_,_) and Precio > 21500
```

La respuesta que podría tomarse equivocadamente como correcta sería :

```
Marca = renault_12  Precio = 90000
Marca = ford_taunus Precio = 13800
```

Otro error frecuente en el uso de estas variables es el siguiente. Supongamos que deseamos exhibir juntos aquellos autos que tengan el mismo color. Supongamos también que debemos exhibirlos de a dos a los de color rojo. Escribimos el Goal que genere todos los pares posibles que cumplan con esta condición :

```
auto (Marca_1,_,_,rojo,_) and auto (Marca_2,_,_,rojo)
```

**La respuesta de Prolog será :**

```
Marca_1 = ford_taunus          Marca_2 = ford_taunus
Marca_1 = ford_taunus          Marca_2 = dodge_1500
Marca_1 = ford_taunus          Marca_2 = volkswagen
Marca_1 = dodge_1500           Marca_2 = ford_taunus
Marca_1 = dodge_1500           Marca_2 = dodge_1500
Marca_1 = dodge_1500           Marca_2 = volkswagen
Marca_1 = volkswagen           Marca_2 = ford_taunus
Marca_1 = volkswagen           Marca_2 = dodge_1500
Marca_1 = volkswagen           Marca_2 = volkswagen
```

Sin dudas esta no es la solución que estábamos buscando. En lugar de una única solución hemos obtenido nueve

Intentemos ahora colocar el siguiente objetivo externo :

```
auto (Marca_1,_,_,rojo,_) and auto (Marca_2,_,_,rojo,_)
and Marca_1 <> Marca_2
```

**Obteniendo la respuesta :**

```
Marca_1 = ford_taunus          Marca_2 = dodge_1500
Marca_1 = ford_taunus          Marca_2 = volkswagen
Marca_1 = dodge_1500           Marca_2 = ford_taunus
Marca_1 = dodge_1500           Marca_2 = volkswagen
Marca_1 = volkswagen           Marca_2 = ford_taunus
Marca_1 = volkswagen           Marca_2 = dodge_1500
```

# 7. OBJETOS COMPUESTOS

El Turbo Prolog, además de trabajar con objetos simples, permite trabajar con argumentos más complejos, como vimos e el ejemplo anterior. La representación del conocimiento se puede simplificar utilizando objetos más complejos. Un hecho simple es de la forma :

predicado (arg_1, arg_2, ..., arg_N)

Ejemplo.

```
Trabaja (juan, panadería)
Trabaja (luis, taller)
```

Hay veces que necesitamos poner más detalles, por ejemplo, podríamos poner el nombre del lugar donde trabajan, el número y la categoría, para esto es conveniente utilizar los “objetos compuestos”, el ejemplo quedaría :

```
trabaja (juan, panadería (sucursal, 12))
trabaja (luis, taller (sección, 97, c))
```

Si observamos detenidamente el ejemplo notaremos que hay argumentos, “panadería” y “taller”, que están actuando como predicados. A estos los denominamos “functores” y a sus argumentos “componentes”.

Los hechos con objetos compuestos son de la forma :

predicado (argumento, functor (componente, componente))

**Declaraciones de predicados y dominios para objetos compuestos** : Si se decide utilizar objetos compuestos, hay que poner especial atención en la declaración de predicados y dominios. Es necesario declarar cada functor y los dominios de todos sus componentes.

Programa ejemplo, con la declaración apropiada para objetos compuestos.

```
domains
        lugar = panadería (nombre_area, número)
                taller (nombre_area, número, categoría)
        nombre, nombre_area, categoría = symbol
        número = integer
predicates
        trabaja (nombre, lugar)
clauses
        trabaja (juan, panadería (sucursal, 12))
        trabaja (luis, taller (sección, 97, c))
```

- Ahora veremos que obtenemos con algunos objetivos externos

```
GOAL : trabaja (Quien, Donde)
       Quien = juan, Donde = panadería (“sucursal”, 12)
       Quien = luis, Donde = taller (“seccion”, 97, “c”)

GOAL : trabaje (Quien, taller (En, Nro, Categoría))
       Quien = luis, En = sección, Nro = 97, Categoria = c
```

En estos dos objetivos externos podemos visualizar cómo, utilizando variables, se puede profundizar en los distintos niveles de los functores.

# 8. LISTAS

L**istas :** es un conjunto de elementos encerrados entre corchetes y separados por comas. Una lista de símbolos sería de la forma [elem_1, elem_2, ..., elem_N]. La lista vacía se representa con dos corchetes, [ ].

La manera más sencilla de escribir listas es enumerar sus elementos. La lista que consta de tres átomos a, b y c puede escribirse como :

```
[ a, b, c ]
```

**También podemos especificar una secuencia inicial de elementos y una lista restante, separadas por |. La lista [a,b,c] también puede escribirse como :**

```
[a, b, c | [ ] ]
[a, b | [c] ]
[a, | [b, c] ]
```

**Un caso especial de esta notación es una lista con cabeza H y cola T, representada como [H | T]. La cabeza puede usarse para extraer los componentes de una lista, de manera que no se requieren operadores específicos para extraer la cabeza y la cola. La solución a la consulta :**

```
? - [H | T] = [a, b, c]
       T = [b, c]
       H = a
```

asocia la variable H con la cabeza y la variable T con la cola de la lista [a,b,c]. La consulta :

```
? - [a | T] = [H, b, c]
       T = [b, c]
       H = a
```

ilustra la capacidad de Prolog para manejar términos especificados parcialmente. El término [a | T] es la especificación parcial de una lista con cabeza **a** y cola desconocida, denotada por la variable T. En forma similar, [H, b, c] es la especificación parcial de una lista con cabeza H desconocida y cola [b, c]. Para unificar estas dos especificaciones, H debe denotar a **a** y **T** debe denotar a [b, c].

## 8.1 LISTAS EN TURBO PROLOG

En Turbo Prolog es necesario declarar el dominio de toda lista que se maneje en un programa. No es complicado, solo se agrega un asterisco junto a la definición del tipo de elementos de la lista. Quedaría así :

```
domains
        lista_enteros = integer *
```

El manejo de listas en Turbo Prolog es muy simple, teniendo en cuenta que divide la lista en solo dos partes : “cabeza” y “cola”.

- Cabeza : es el primer elemento de la izquierda

- Cola : es el resto de la lista (es también una lista)

[lista_enteros] ➔ [cabeza **OR** cola]

Veamos cómo es esta división en la lista antes declarada.

Enteros (lista_enteros [1, 2, 3, 4, 5])

Donde : Cabeza = 1

Cola = [ 2, 3, 4, 5 ]

Hay que tener en cuenta que toda lista es un objeto, y se comporta como tal. Es decir, cuando está como argumento debe estar encerrada entre paréntesis, pueden ser más de una las listas que están como argumentos. De la forma :

predicado ([lista_1], [lista_2], ..., [lista_N])

**- Identificación de la cabeza y la cola :** En un primer lugar el sistema busca una cláusula que identifica el objetivo. Debe coincidir el predicado, el número de argumentos tiene que ser el mismo y los propios argumentos deben coincidir directamente o a través de instanciaciones correctas. El sistema llama a la cláusula, devuelve la cláusula con los valores instanciados y luego parte la lista en una cabeza y una cola.

La cola es todo lo que queda a la derecha de **"OR"**, ya que podemos direccionar a los dos primeros elementos poniendo dos variables y luego **"OR"**, y una tercera variable que sería la cola.

Ej : En la lista [ A, B **OR** C ]. A y B son los dos primeros elementos y C es la cola. Cabe aclarar que la cola sigue siendo una lista, inclusive una lista vacía, por lo que de la misma manera que obtuvimos los dos primeros elementos podríamos obtener elementos subsiguientes, utilizando variables posicionales.

Si la lista original está vacía, la cabeza y la cola están indefinidas. Habíamos dicho que los argumentos deben coincidir en el mismo orden, por lo tanto si la cabeza de la lista objetivo no coincide con la cabeza de la lista de la cláusula, no hay solución al objetivo.

- Como vimos anteriormente, en Turbo Prolog se pueden hacer estructuras complejas. Se pueden poner objetos compuestos dentro de objetos compuestos. De la misma manera se pueden poner listas dentro de listas, es decir que cada elemento de la lista sea a su vez una lista.

Ej : Tenemos una lista que indica los niveles de una empresa

```
empresa ( [ gerentes, contadores, secretarios ] )
```

Pero nosotros podríamos sentir la necesidad de detallar más nuestros conocimientos, y tal vez sería bueno poner el apellido de esos gerentes, contadores y secretarios. Para ello transformaremos a cada elemento en una lista :

```
empresa ([ruiz, alvarez, gomez], [ramírez, rodriguez], [garcía, salinas, perez])
```

Pero observamos que queda como tres listas de nombres, y no es muy representativa, por lo tanto utilizaremos functores :

```
empresa (     gerentes ([ruiz, alvarez, gomez]),
              contadores ([ramírez, rodriguez]),
              secretarios ([garcía, salinas, perez]) )
```

Para manejar listas, es conveniente utilizar funciones RECURSIVAS. La utilización de la recursión suele dar como resultado procedimientos más compactos y simples.

Una manera sencilla de mostrar una operación recursiva, es comprobar si un objeto dado pertenece a una lista.

## Supongamos tener una base de datos que incluye una lista de materias el ciclo básico y del ciclo superior.

| Ciclo Básico | Ciclo Superior |
|---|---|
| geografía | diseño |
| anatomía | cálculo |
| historia | literatura |
| geometría | taller |
| botánica | tecnología |

Para ver si una materia pertenece a uno u otro ciclo superior, se podría hacer un listado de hechos simples que los describan

Ejemplo :

```
domains
       materia = symbol
predicates
       ciclo_basico (materia)
       ciclo_superior (materia)
```

```
clauses
        ciclo_basico (geografía)
        ciclo_basico (anatomía)
        ciclo_basico (historia)
        ciclo_basico (geometría)
        ciclo_basico (botánica)

        ciclo_superior (diseño)
        ciclo_superior (cálculo)
        ciclo_superior (literatura)
        ciclo_superior (taller)
        ciclo_superior (tecnología)
```

Ahora, si nosotros quisiéramos agregar más materias, tendríamos que agregar una cláusula por cada materia que se agregue.

Trabajando con listas esto no es necesario, además las definiciones recursivas de “miembro” hacen lo mismo y en menos espacio.

El programa mostrado anteriormente tenía en la sección de cláusulas hechos simples. En el próximo ejemplo veremos esta sección con la definición recursiva de “miembro” usando listas.

Ejemplo :

```
domains
        lista_materia = materia *
        materia = symbol
predicates
        ciclo_basico (materia)
        ciclo_superior (materia)
        miembro (materia, lista_materia)
clauses
        ciclo_basico (materia) if
                miembro (materia, [geografía, anatomía, historia, geometría,
        botánica])

        ciclo_superior (materia) if
                miembro (materia, [diseño, cálculo, literatura, taller,
        tecnología])
        miembro (materia, [materia OR _])
        miembro (materia, [_ OR materia]) if
        miembro (materia, Cola)
```

En este ejemplo se ve claramente como las listas reemplazan a los predicados.

En las dos primeras cláusulas se define una materia como perteneciente a un ciclo, si pertenece a la respectiva lista.

Las otras dos cláusulas son la definición recursiva de “miembro”

Introduciendo el objetivo “ciclo_superior (tecnología)” obtendremos como resultado “TRUE”.

# 9. RECUSIVIDAD

Si del lado derecho de una cláusula aparece en algún punto el mismo predicado que figura del lado izquierdo, se dice que la cláusula tiene llamado recursivo, es decir “se llama a sí misma” para verificar que se cumple esa misma propiedad como parte de la condición que define a la regla, y sobre algún posible valor de sus variables.

El uso de recursividad es muy común y suele ser imprescindible para de alguna manera simular esquemas de iteración implícitos.

Por ejemplo, si tenemos :

```
progenitor(X,Y) :- padre(X,Y)
progenitor(X,Y) :- madre(X,Y)
```

que expresa que tanto X es padre de Y como X es madre de Y son condiciones suficientes para afirmar que X es progenitor de Y. Podemos definir :

```
antepasado(X,Y) :- progenitor (X,Y)
antepasado(X,Y) :- progenitor (X,Z), antepasado (Z,Y)
```

que dice que X es antepasado de Y si es progenitor de Y, o bien, si es progenitor de algún otro antepasado de Y. La palabra antepasado figura de ambos lados de una de las reglas.

Supongamos ahora que contamos con una base de datos (“hechos”) que son ciudades, y deseamos hacer una lista de todos los caminos existentes de una ciudad a otra. Contamos para eso con rutas simples entre pares de ciudades. Por ejemplo :

```
ruta (BsAs, MardelPlata)
ruta (BsAs, Miramar)
ruta (MardelPlata, Miramar)
ruta (LaPlata, Miramar)
ruta (BsAs, LaPlata)
ruta (BsAs, Jujuy)
ruta (Miramar, BahíaBlanca)
```

**Digamos ahora que toda ruta es reversible, es decir si puedo ir de C1 a C2 (de la ciudad 1 a la 2), podremos volver por la misma ruta de C2 a C1. Eso se escribe así en Prolog :**

```
ruta (C1, C2) :- ruta (C2, C1)
```

Por último, para definir cuándo un camino existe, podemos atravesar ciudades intermedias. Escribimos entonces :

```
camino (C1, C2) :- ruta (C1, C2)
camino (C1, C2) :- ruta (C1, C3), camino (C3, C2)
```

que dice que existe un camino de la ciudad C1 a la ciudad C2 si hay una ruta (predefinida) entre ambas, o bien si hay una ruta desde la ciudad C1 hacia una ciudad C3, y desde ésta hasta la ciudad C2. Vale decir, el llamado recursivo significa resolver el problema original mediante dar un paso hacia una posición auxiliar y resolver el mismo problema desde esa posición (se supone que el problema se simplifica).

Con estas definiciones, la respuesta será afirmativa si consultamos :

```
? camino (BahíaBlanca, Jujuy)
```

Otro ejemplo, definamos ahora que es un número par (positivo). Sabemos que el 0 es par, y que los demás pares se obtienen simplemente al ir sumando 2 a partir de un par conocido. Bien, prolog cuenta con tipos numéricos y operaciones elementales predefinidas. Por ello, esto se puede lograr simplemente con :

```
par (0)
par (N) :- par (M), N = M + 2
```

que afirma que el 0 es par, y que si tenemos que M es par, entonces definiendo N = M + 2 será también un número par.

No siempre la recursividad es buena o garantiza éxito en una búsqueda de solución. Por ejemplo si ponemos:

```
lindo (X) :- lindo (X)
```

muchas versiones de prolog entraran en un ciclo infinito, ya que para buscar algo que sea lindo, la solución es buscar algo que sea lindo, es decir el mismo problema sin cambio.

# PARTE PRACTICA

1. A partir de los siguientes predicados :

   PADRE (padre, hijo) MADRE (madre, hijo)
   CASADO (hombre, mujer) LINDO (X)

   y considerando los siguientes hechos :

   juan es padre de marcelo y maría. mirian es la madre de ricardo.
   raul es el padre de sergio. marcelo es el padre de raul y rita.
   rita es la madre de victor y veronica. mirian y rita son lindas.

   Se desean extraer las siguientes respuestas :

   a. Quien es el abuelo de victor. b. Quien es el nieto de raul.
   c. Quien es la hermana de victor. d. Quien es la hermana de veronica.
   e. De quien es hija veronica. f. Quien está casado con alguien linda.
   g. Que relación familiar tiene las personas lindas.
   h. Quienes son tíos y cuales son sus sobrinos.

2. A partir de los siguientes predicados :

   PERSONA (nombre, sexo)
   AMISTAD (nombre, nombre)
   SOSPECHOSO (nombre)

   y considerando :

   barbara es amiga de juan. barbara es amiga de roberto.
   barbara es amiga de maría. susana es amiga de juan .
   susana es amiga de pedro.

   Se debe escribir un programa que permita individualizar a los sospechosos del asesinato de susana. Para ello considerar como sospechosos a :

   a. Los hombres que tuvieron amistad con susana.
   b. Las mujeres que tuvieron relación con hombres que susana conocía.
   c. Los amigos de mujeres que tuvieron relación con hombres que susana conocía.

3. Considerar los siguientes predicados

   PERSONA (nombre, sexo, edad)
   MAYOR_EDAD (nombre, nombre)

   y sabiendo que :

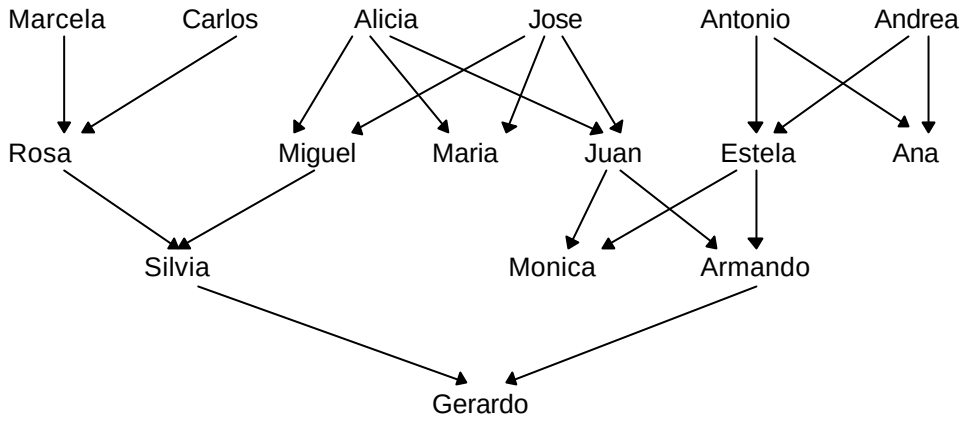
   juan tiene 20 años pedro tiene 30 años
   luis tiene 28 años ana tiene 10 años
   alejandra tiene 43 años jorge tiene 60 años

   escribir un programa que permita identificar a las personas mayores de cierta edad, identificándolas como hombres o mujeres.

4. En base a los siguientes predicados :

   padres ()
   masc (nombre)
   fem (nombre)

   y considerando el siguiente gráfico de información se pide :




a. Que hijos tiene Alicia.
b. Que hermanos tiene Juan.
c. Que hermanas tiene Armando.
d. Quiénes son los antepasados de Monica.
e. Quienes son los parientes de Silvia.
f. Interpretar lazos como : tíos, cuñados, primos y matrimonios.

5. Desarrolle un programa que pueda decir quienes son los que llegan seguros a sus casas después de una fiesta y quienes no, en donde deberá probar si puede hacerlo sin ayuda, o con ayuda, siendo este último carácter probable a través de que : Existe un conductor seguro, que tiene un coche y que existen lazos de amistades entre los individuos. A partir de las siguientes inferencia :

juan, josé, jeremías y jorge poseen autos.
aída y analía viven cerca de l fiesta para ir caminando.
roberto, raúl, rodrigo y ana viven demasiado lejos.
josé y jeremías solo bebieron jugo.
anacleta vive lejos y es amiga de juan.
anastasia vive lejos y es amiga de josé.
raúl y josé no son amigos, pero si roberto y jeremías.
ana es amiga de josé.
analía es amiga de juan.
aída no conoce a ninguno.
juan bebió litros de vino.
jorge se dedicó a bailar.
jorge fué el único que no trajo el auto.

Dada la amistad de cada uno de los personajes, considere que puede ofrecerle a su amigo el hecho de llevarlos en el auto.

6. Desarrolle un programa que pueda, a través de distintas preguntas, decir que animal es el que esta pensando el usuario que ejecute tal programa, para ello es necesario que el ser humano ante las cuestiones que realice la computadora (tales como : tiene pelos, plumas, garras, cantidad de patas, en donde es su hábitat, tipo de genero, etc.), conteste si o no.

7. Desarrolle un programa que resuelva cual es la alternativa de viaje más corta y la más larga para un turista, al cual se le presenta el siguiente mapa :

| DESDE | HASTA | DISTANCIA |
|---|---|---|
| Nueva Córdoba | Alta Córdoba | 3000 ms. |
| Nueva Córdoba | Casco Céntrico | 200 ms. |
| Casco Céntrico | Villa Allende | 4000 ms. |
| Alta Córdoba | Villa Allende | 750 ms. |
| Nueva Córdoba | Villa Arguello | 5000 ms. |
| Villa Arguello | Villa Allende | 3000 ms. |
| Nueva Córdoba | Las Palmas | 4000 ms. |
| Las Palmas | Villa Allende | 3200 ms. |
| Nueva Córdoba | Los Plátanos | 2000 ms. |
| Los Plátanos | San Fernando | 3000 ms. |
| San Fernando | Villa Allende | 1700 ms. |

Su destino es Nueva Córdoba - Villa Allende

8. Desarrolle un programa, tal que pueda resolver el factorial de un número entero ( que acepte el cero), ingresado desde teclado.

**SOLUCIONES**

```
Ejercicio Nº 4

domains
       persona=symbol
predicates
       masc (persona)
       fem (persona)
       casado (persona, persona)
       padre (persona, persona)
       madre (persona, persona)
       hermanos (persona, persona)
       hijo (persona, persona)
       hija (persona, persona)
       abuelo (persona, persona)
       nieto (persona, persona)
       tio (persona, persona)
       primo (persona, persona)
       suegro (persona, persona)
       suegra (persona, persona)
       sobrino (persona, persona)
clauses
       masc (angel)
       masc (carlos)
       masc (eduardo)
       masc (basilio)
       masc (michel)
       masc (mario)
       fem (ofelia)
       fem (ramona)
       fem (ana)
       fem (elena)
       fem (betty)
       fem (silvia)
       padre (basilio,angel)
       padre (basilio,elena)
       padre (carlos,ofelia)
       padre (angel,betty)
       padre (angel,eduardo)
       padre (michel,mario)
       padre (michel,silvia)
       madre (ramona,angel)
       madre (ramona,elena)
       madre (ana,ofelia)
       madre (ofelia,betty)
       madre (ofelia,eduardo)
       madre (elena,mario)
       madre (elena,silvia)
       hijo (X,Y) if masc (X) and padre (Y,X)
       hijo (X,Y) if masc (X) and madre (Y,X)
       hijo (X,Y) if fem (X) and padre (Y,X)
       hijo (X,Y) if fem (X) and madre (Y,X)
       hermanos (X,Y) if padre (Z,X) and padre (Z,Y) and X<>Y
       casado (X,Y) if hijo (Z,Y) and hijo (Z,X) and X<>Y
       abuelo (X,Y) if padre (X,Z) and padre (Z,Y)
       abuelo (X,Y) if padre (X,Z) and madre (Z,Y)
       nieto (X,Y) if padre (Y,Z) and padre (Z,X)
       nieto (X,Y) if padre (Y,Z) and madre (Z,X)
       nieto (X,Y) if madre (Y,Z) and padre (Z,X)
       nieto (X,Y) if madre (Y,Z) and madre (Z,X)
       primo (X,Y) if
              nieto (X,Z) and nieto (Y,Z) and
              not (hermanos(X,Y)) and   X<>Y
```

```
      tio (X,Y) if
            padre (Z,Y) and hermanos (X,Z)
      tio (X,Y) if
            madre (Z,Y) and hermanos (X,Z)
      tio (X,Y) if
            casado (X,Z) and madre (W,Y) and hermanos (W,Z)
      suegro (X,Y) if masc (X) and
            padre (X,Z) and casado (Y,Z)
      suegra (X,Y) if fem (X) and
            madre (X,Z) and casado (Y,Z)
      sobrino (X,Y) if tio (Y,X)

GOAL
      1)    madre (elena,X)
            X = mario                 Y = silvia
      2)    abuelo (X,eduardo)
            X = basilio               Y = carlos
      3)    nieto (X,carlos)
            X = eduardo               X = betty
      4)    primo (X,mario)
            X = eduardo               Y = betty
      5)    tio (elena,X)
            X = betty                 X = eduardo
      6)    sobrino (X,ofelia)
            X = mario                 X = silvia
      7)    suegro (X,Y)
            X = basilio               Y = ofelia
            X = carlos                Y = angel
            X = basilio               Y = michel
      8)    hermanos (betty,X)
            X = carlos

Ejercicio Nº 5

domains
      pers = symbol
predicates
      auto (pers)
      cerca (pers)
      lejos (pers)
      amigos (pers,pers)
      borracho (pers)
      seguro (pers)
      no_seguro (pers)
clauses
      lejos (roberto)
      lejos (raul)
      lejos (rodrigo)
      lejos (anacleta)
      lejos (anastacia)
      cerca (aida)
      cerca (ana)
      cerca (analia)
      auto (juan)
      auto (jose)
      auto (jeremias)
      auto (jorge)
      amigos (ana,jose)
      amigos (analia,juan)
      amigos (anacleta,juan)
      amigos (anastacia,jose)
      borracho (juan)
      seguro (X) if cerca (X) and not (borracho (X))
      seguro (X) if lejos (X) and
            not (borracho (X)) and
            auto (X)
      seguro (X) if auto (X) and not (borracho (X))
      seguro (X) if
            lejos (X) and
            amigos (X,Y) and
```

```
              auto (Y) and
              not (borracho (Y))
       no_seguro (X) if borracho (X)
       no_seguro (X) if lejos (X) and
              amigos (X,Y) and
              borracho (Y)

GOAL
       1)     seguro (X)
              X = aida            X = ana             X = jeremias
              X = jorge           X = anastacia       X = analia
              X = jose
       2)     no_seguro (X)
              X = juan                  X = anacleta

Ejercicio Nº 6

predicates
       tipo (symbol,symbol)
goal
       write (“                          “)
       write (“Ingrese un animal :”),
       readln (Animal),
       tipo (Animal,Medio),
       write (“El medio de este animal es : “, Medio)
clauses
       tipo (“perro”, “doméstico”)
       tipo (“gato”, “doméstico”)
       tipo (“gallina”, “corral”)
       tipo (“ballena”, “océano”)
       tipo (“leon”, “selva”)
       tipo (“elefante”, “sabana”)
       tipo (“hornero”, “monte”)
       tipo (“canario”, “domestico”)
       tipo (“vaca”, “corral”)
       tipo (“camello”, “desierto”)
       tipo (“trucha”, “rio”)
       tipo (“rana”, “laguna”)
       tipo (“mono”, “selva”)
       tipo (“caballo”, “domestico”)
       tipo (“mosca”, “doméstico”)
       tipo (“condor”, “cordillera”)
       tipo (“tiburon”, “oceano”)

GOAL
       1)     Ingrese un animal : perro
              El medio de este animal es : domestico
       2)     Ingrese un animal : mono
              El medio de este animal es : selva
       3)     Ingrese un animal : elefante
              El medio de este animal es : sabana
```