



# **UNIDAD N° 5:**

## **PARADIGMA LÓGICO**

**EL PRESENTE CAPÍTULO HA SIDO ELABORADO, CORREGIDO Y COMPILADO POR:**

**MGTR. ING. CORSO, CYNTHIA**  
**ESP. ING. COLACIOPPO NICOLÁS**  
**ESP. ING. GUZMAN, ANALIA**  
**ESP. ING. LIGORRIA, KARINA**  
**ESP. ING. LIGORRIA, LAURA**  
**DR. ING. MARCISZACK, MARCELO**

## ÍNDICE

### Contenido

1. INTRODUCCIÓN.....	3
1.1 CAMPOS DE APLICACIÓN DEL PARADIGMA LÓGICO. ....	4
1.3 LIMITACIONES:.....	4
2. LÓGICA PROPOSICIONAL.....	5
2.1 ELEMENTOS DE LENGUAJE DE PRIMER ORDEN.....	6
3. INTRODUCCIÓN A PROLOG.....	11
3.1 PROLOG: CÁLCULO DE RELACIONES.....	13
3.2 ELEMENTOS DE UN PROGRAMA EN PROLOG.....	15
3.6 PROLOG: PRINCIPIO DE RESOLUCIÓN O REGLA DE INFERENCIA.....	23
3.8 PROLOG: BÚSQUEDA DE SOLUCIONES.....	26
3.9 BACKTRACKING.....	28
3.10 EL CONTROL DEL BACKTRACKING:.....	30
3.10.1 EL PREDICADO CORTE.....	30
3.10.2 EL PREDICADO FAIL.....	31
3.10.3 EL CONTROL DEL BACKTRACKING: EJEMPLO DE CORTE Y FALLO. ....	31
4.2 APRECIACIONES FINALES DEL LENGUAJE PROLOG.....	38
5. OBJETOS COMPUESTOS O ESTRUCTURAS.....	39
6. RECURSIVIDAD.....	41
7.1 PREDICADOS SWI-PROLOG PARA EL MANEJO DE LISTAS.....	45
7.2 EJEMPLIFICACIÓN DE PREDICADOS PARA EL MANEJO DE LISTAS.....	46
7.3 RECURSIVIDAD EN LISTAS.....	48

## ÍNDICE DE FIGURAS Y TABLAS

Esquema 1. Una aplicación basada en el paradigma lógico.....	17
TABLA 1. OPERADORES LÓGICOS.....	20
Figura 1: Ventana Principal de SWI-Prolog.....	33
Figura 2: Ejemplificación del comando ls.....	33
Figura 3: Ejemplificación del comando pwd (Ventana Principal).....	34
Figura 4: Ejemplificación del comando edit.....	34



Figura 5: Ventana de edición .....	35
Figura 6: Ejemplo de programa en SWI-Prolog .....	36
Figura 7: Ejemplificación del comando consult .....	36
Figura 8: Ejemplo de consultas a un programa lógico .....	37
Figura 9: Ejemplificación del comando para edición de un archivo .....	37
Figura 10: Ejemplificación del comando halt .....	38
TABLA 2. Ejemplo de predicados para el manejo de listas .....	47



### **OBJETIVOS DE LA UNIDAD**

Que el alumno comprenda acabadamente los principios constitutivos y filosóficos que dan origen a este paradigma.

Que el alumno utilice el concepto de relaciones de predicados y clausulas de primer orden para la construcción de programas en la resolución de problemas.

### **CONTENIDOS ABORDADOS**

Introducción a la Programación Lógica. Campos de Aplicación del Paradigma Lógico. Ventajas. Limitaciones. Lógica Proposicional. Elementos del lenguaje de Primer Orden. Introducción a Prolog. Relaciones en Prolog. Cálculo de Relaciones. Elementos de un programa en Prolog. Sintaxis. Tipos de Datos. Operadores. Principio de Resolución o Regla de Inferencia. Unificación. Búsqueda de Soluciones. Backtracking. Operadores de control del Backtracking: Corte y Fallo. Objetos Compuestos y Estructuras. Recursividad. Listas. Predicados para el manejo de listas. Recursividad en Listas.

## **1. INTRODUCCIÓN**

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.



## 1.1 CAMPOS DE APLICACIÓN DEL PARADIGMA LÓGICO.

- Los campos de aplicación de este paradigma son:
  - Inteligencia Artificial.
  - Sistemas basados en el conocimiento.
  - Procesamiento del lenguaje natural.
  - Sistemas expertos, donde un sistema de información imita las recomendaciones de un experto sobre algún dominio de conocimiento.
  - Demostración automática de teoremas, donde un programa genera nuevos teoremas sobre una teoría existente.
  - Reconocimiento de lenguaje natural, donde un programa es capaz de comprender (con limitaciones) la información contenida en una expresión lingüística humana.

## 1.2 VENTAJAS:

- Simplicidad
- Cercanía a las especificaciones del problema realizada con lenguajes formales.
- Metodología rigurosa de especificación.
- Sencillez en la implementación de estructuras complejas

## 1.3 LIMITACIONES:

- Poco eficientes
- Poco utilizado en aplicaciones comerciales o transaccionales



## 2. LÓGICA 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**. 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. 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, 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.

A continuación desarrollaremos algunos conceptos elementales de la lógica.



## 2.1 ELEMENTOS DE LENGUAJE DE PRIMER ORDEN

### 2.1.1 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.

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.

### 2.1.2 SÍMBOLOS 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.





### 2.1.3 ENUNCIADOS ATÓMICOS

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.

### 2.1.4 ENUNCIADOS ATÓMICOS COMBINADOS

Son enunciados que combinan dos o más enunciados simples utilizando un conectivo lógico (o,y,no). Ejemplo:



docente(Alejandra) y alumno(Juan)

docente(Alejandra) y enseña(Alejandra,Juan)

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

*profesional( profesor)*

*matricula(profesor)*

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

*profesional (jefe de trabajos prácticos)*

Evidentemente, Jefe de Trabajos Prácticos es un símbolo que denota que es un profesional, lo que restringe las posibilidades de lo que éste puede ser, ya que Jefe de Trabajos Prácticos satisface el predicado Profesional.

Se puede expresar otras restricciones con otros predicados, como dicta\_clase y corrige. De hecho, es posible limitar los objetos que jefe de trabajos prácticos puede designar a aquellos objetos que satisfacen ambos predicados al mismo tiempo, al afirmar que las dos expresiones siguientes son verdaderas:

*dicta\_clase (jefe de trabajos prácticos)*

*corrige (jefe de trabajos prácticos)*

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.

*dicta\_clase (jefe de trabajos prácticos) y corrige (jefe de trabajos prácticos)*

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

*dicta\_clase (jefe de trabajos prácticos) o corrige (jefe de trabajos prácticos)*

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

Observe que “y” y “o” se conocen como conectivos lógicos porque transforman combinaciones de falso y verdadero.



## 2.1.5 PREDICADOS CON CONSECUENTE

Constituído por una serie de argumentos, uno llamado conclusión que son apoyados por otros llamados antecedentes.

Ejemplo:

docente(Alejandra) si alumno(Juan) y enseña(Alejandra,Juan)

Conclusión      Antecedentes

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). He aquí un ejemplo:

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

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  $P, Q, R, \dots$  es una demostración paso a paso que muestra que  $S$  debe ser verdadera en cualquier circunstancia en que las premisas  $P, Q, R, \dots$  son todas verdaderas.

### 3. INTRODUCCIÓN 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.

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 adecuado para las aplicaciones que implican búsqueda de patrones, búsqueda con rastreo inverso o información incompleta.

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, diversas compañías de software que han creado sus propias versiones del mismo. La diferencia es mínima entre las versiones, ya que su sintaxis y semántica es la misma. La variación que más resalta es el cambio de plataforma para el cual fueron desarrollados.

A continuación se citan algunas versiones:

- **SWI – Prolog IDE. Sitio oficial:** <http://www.swi-prolog.org>
- **Prolog Development Tools (ProDT): Sitio Oficial:** <http://prodevtools.sourceforge.net/>
- **eeWhiz: Sitio Oficial:** <http://hulles.supersized.org/>
- **GNU Prolog: Sitio Oficial:** <http://www.gprolog.org/>
- **Visual Prolog: Sitio Oficial:** <http://www.visual-prolog.com/>

### 3.1 PROLOG: CÁLCULO 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**<sup>1</sup>, llamadas así en honor al lógico Alfred Horn, quien las estudió.

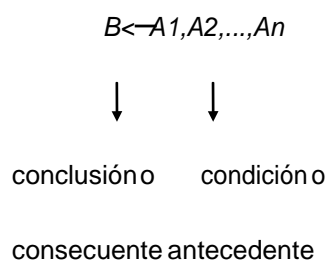
Estas reglas están compuestas por dos partes: el consecuente y el antecedente. El consecuente, que es la primera parte de la cláusula, es lo que se quiere probar, la conclusión de la regla. El antecedente es la condición que determinará en qué casos el consecuente es verdadero o falso. Esta estructura 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.

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.



---

<sup>1</sup> Las cláusulas de Horn son fórmulas (predicados) bien formadas, que se encuentran en forma clausal.



Estas cláusulas pueden adoptar las siguientes formas:

Conclusión	$B \leftarrow$	Afirmación	Hecho
conclusión si condición	$B \leftarrow A_1, A_2, \dots, A_n$	Implicación	Regla
si condición	$\leftarrow A_1, A_2, \dots, A_n$	Negación	Objetivo

### Afirmación

$B \leftarrow$  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.

### Implicación

$B \leftarrow A_1, A_2, \dots, A_n$  Afirmación condicional, donde el predicado B es

verdadero si  $A_1, A_2, \dots, A_n$  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.

### Negación

$\leftarrow A_1, A_2, \dots, A_n$  Cláusula objetivo o goal

Cada uno de los predicados  $A_1, A_2, \dots, A_n$  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 está 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.

### 3.2 ELEMENTOS DE UN PROGRAMA EN PROLOG

*Un programa en Prolog está conformado por una serie de elementos:*

**Base de Conocimiento:** representado por un conjunto de afirmaciones (hechos y reglas) representando los conocimientos que poseemos en un determinado dominio de campo o de nuestra competencia.

**Motor de Inferencia:** es el que se encarga de la “*ejecución del programa*”. Que en esencia es: Un comprobador de teoremas, el cual utiliza la Regla Inferencia que será explicada en detalle en la próxima sección.

**Resolución (Intérprete de comandos):** cuyo objetivo es permitir la posibilidad de responder consultas.

El hecho de programar en Prolog consiste en brindar a la computadora un universo Finito en forma de hechos y reglas, proporcionando los medios para realizar inferencias de un hecho a otro. A continuación si se hacen las preguntas adecuadas, Prolog buscará la respuesta en dicho universo y las presentará en pantalla.

Se puede resumir que la estructura de un programa en Prolog es:

LOGICA + CONTROL = PROGRAMA

*Los pasos a seguir para escribir un programa en Prolog:*

- a) Declarar **HECHOS** sobre los objetos y relaciones.
- b) Definir **REGLAS** sobre los objetos y relaciones.
- c) Hacer **PREGUNTAS** sobre los objetos y relaciones.

a) **Hechos:** Expresan relaciones entre objetos. Supongamos que queremos expresar el hecho de que "un coche tiene ruedas". Este hecho, consta de dos objetos, "coche" y "ruedas", y de una relación llamada "tiene". La forma de representarlo en PROLOG es:

tiene (coche,ruedas).

Algunas características de los hechos son:

- Los nombres de objetos y relaciones deben comenzar con una letra minúscula.





- Primero se escribe la relación, y luego los objetos separados por comas y encerrados entre paréntesis.
- Al final de un hecho debe ir el carácter "." (punto).

El orden de los objetos dentro de la relación es arbitrario, pero debemos ser coherentes a lo largo de la base de hechos.

**b) Reglas:** Las reglas se utilizan en Prolog para significar que un hecho depende de uno o más hechos. Es la representación de las implicaciones lógicas del tipo  $p \rightarrow q$  (p implica q).

Ejemplo: `sabe(persona):-estudia(persona).`

Algunas características son:

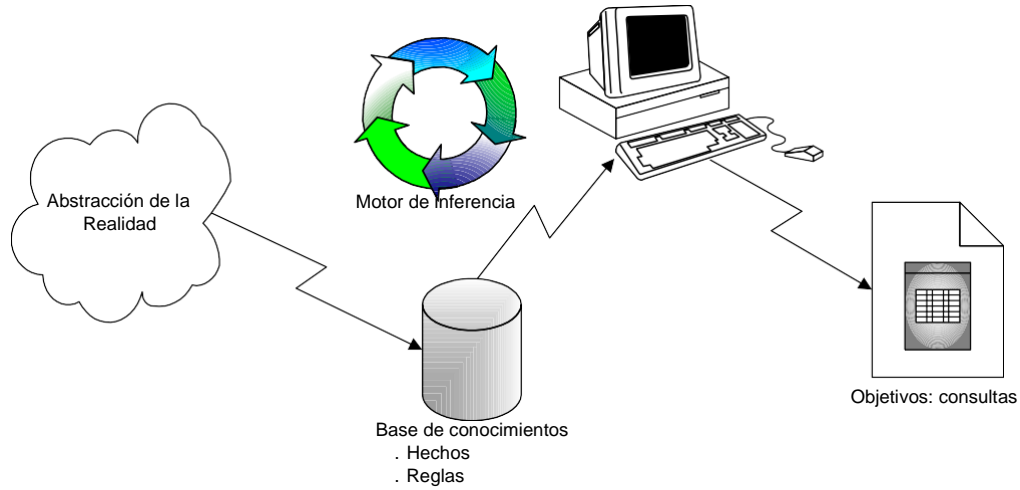
- Una regla consiste en una cabeza y un cuerpo, unidos por el signo " :- ".
- La cabeza está formada por un único hecho.
- El cuerpo puede ser uno o más hechos (conjunción de hechos), separados por una coma (","), que actúa como el "y" lógico.
- Las reglas finalizan con un punto (".").

**b) Preguntas o Consultas:** las preguntas son las herramientas que tenemos para recuperar información desde Prolog. Al hacer una pregunta a un programa lógico queremos determinar si esa pregunta es *consecuencia lógica* del programa. Prolog considera que todo lo que hay en la Base de conocimiento es verdad, y lo que no, es falso. De manera que si Prolog responde "yes" es que ha podido demostrarlo, y si no, es que no lo ha podido demostrar (no debe demostrarse como "falso" sino con lo que Prolog conoce no puede demostrarse su veracidad).

Cuando se hace una pregunta a Prolog, éste efectuará una búsqueda por toda la Base de Conocimiento intentando encontrar hechos que coincidan con la pregunta.

En la siguiente figura se ilustra los elementos que intervienen en un programa desarrollado bajo el enfoque del paradigma lógico.

Esquema de una aplicación basada en el paradigma lógico



**ESQUEMA 1. UNA APLICACIÓN BASADA EN EL PARADIGMA LÓGICO**

De lo expuesto en esta sección, se puede concluir que Prolog es un programa conformado por un conjunto de hechos y reglas que representan el problema que se pretende resolver. Ante una determinada pregunta sobre el problema, el Prolog utilizará estos hechos y reglas para intentar demostrar la veracidad o falsedad de la pregunta que se le ha planteado.



### 3.3 PROLOG: SINTAXIS

#### Comentarios

- Los comentarios comienzan con %.

% Hola, esto es un comentario.

% Y esto también.

#### Finalización de expresiones

- Las expresiones o cláusulas terminan con punto (punto).

docente(ana).

#### Variable lógica:

- Es una incógnita, algo que está sin resolver.

*docente(X).*

X comienza con mayúscula, lo que me indica que no es un valor fijo. Podría hacer docente(Quien) si lo quisiera hacer más legible.

- Las variables deben escribirse con Mayúsculas.
- Caso especial: Variables anónimas representadas por \_ (guión bajo), son variables sin nombre.

*docente(\_).*

#### Constantes

- Representan valores concretos ligados a las expresiones.
- Las constantes se escriben en Minúsculas.
- No se pueden dejar espacios entre los nombres de las constantes.
- Son términos cero-arios, pueden ser constantes de carácter (átomos) o constantes numéricas.
- \_Ejemplo:        docente(ana).



?- docente(X).

resultado X=ana el valor ana se liga a X

### 3.4 PROLOG: TIPO DE DATOS

- Prolog no es un lenguaje con asignación de tipos fuerte. La lógica se preocupa más de las relaciones entre objetos que del tipo de éstos, dando a todos ellos un tratamiento similar.
- Los datos que maneja Prolog son los términos. Sin embargo, podemos construir otros tipos de datos a partir de estos. El valor que puede tomar una variable consiste en cualquier término, por Ejemplos:

*J(3), 23.2, 'HOLA QUE TAL', ETC.*

- Prolog tiene definiciones de números, secuencias de caracteres, listas, tuplas y patrones.

#### Tipos simples:

- Booleanos: true, false
- Números: enteros, reales
- Secuencias de caracteres: átomos, string o symbol.

#### Tipos Compuestos (se profundizará mas adelante)

- Listas
- Objetos Compuestos



### 3.5 PROLOG: OPERADORES

- Operadores lógicos:

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

TABLA 1. OPERADORES LÓGICOS

- Operadores aritméticos:

$+$ ,  $-$ ,  $*$ ,  $/$

//: cociente de la división (división entera)

mod : resto de la división (módulo)

$^$ : potencia (primer operando elevado a segundo operando)

**Ejemplo:** operador suma (" $+$ ")

*forma prefija* '+ (2,5)'

*forma infija*, '2 + 5'.

- Operadores relacionales:  $>$ ,  $<$ ,  $>=$ ,  $<=$

- Operadores de igualdad:

$=$  igualdad lógica.

$\backslash=$  desigualdad lógica.

$=$  = "es exactamente igual que". Compara términos sin evaluar expresiones.

$:=$  igualdad aritmética

$\backslash:=$  desigualdad aritmética



- **Operador de asignación:**

- No existe la asignación / no hay efecto colateral: una vez más una variable no es una posición de memoria que almacena estados intermedios. Una variable es una incógnita, no tiene sentido que yo le haga  $X = X + 1$ , porque esa condición nunca se puede cumplir.
- Existe la unificación que consiste en que las variables lógicas toman un valor o se ligan a ellos.

- **Operador is: evaluador**

- Se utiliza para evaluar las expresiones aritméticas y funciones.
- Evalúa la parte de la derecha y unifica a la parte izquierda.
- Si no se usa el *is*, las expresiones se mantienen en su forma original:

$X=3+3$  se evalua como  $X=3+3$

$X \text{ is } 3+3$  se evalua  $X=6$

**Ejemplo:**

?- X is 3+4.	?- X is 8, X = 3+5.
X = 7	No
Yes	?- X is 8, X is 3+5.
?- X+Y = 3+5.	X = 8
X=3, Y=5	Yes
Yes	?- 3 = = 1+2.
?- X=3+5.	No
X=3+5	
Yes	

**Instanciación de variables con operadores:**

- Una variable está instanciada cuando Prolog le ha asignado un valor.



- Los operadores aritméticos y relacionales necesitan que todas las variables implicadas en la expresión correspondiente estén instanciadas en el momento en que se realiza la evaluación.

**Ejemplo:**

- $? - 5 \text{ is } X + 4$  ¡Error! X no está instanciada y no se puede realizar la suma.
- $? - Y \text{ is } 1 + 4$ . Correcto:  $Y = 5$
- $? - X \text{ is } 1, Y \text{ is } X + 4$ . Correcto:  $Y = 5$
- $? - 5 := X + 4$ . ¡Error! X no está aún instanciada y no se puede realizar la suma.
- $? - 5 := X + 4, X = 1$  ¡Error! X no está instanciada y no se puede realizar la suma.



### 3.6 PROLOG: PRINCIPIO DE RESOLUCIÓN O REGLA DE INFERENCIA

**El principio de resolución o regla de Inferencia:** Es un algoritmo que, a partir de la negación de la pregunta y los hechos y reglas del programa, intenta llegar al absurdo para demostrar que la pregunta es cierta.

Este principio de resolución fue propuesto por Robinson, que propone una **regla de inferencia** a la que llama **resolución**, mediante la cual la demostración de un teorema puede ser llevada a cabo de manera automática.

En definitiva la resolución es una regla de inferencia que permite a la computadora decir qué proposiciones siguen lógicamente a otras proposiciones. Prolog utiliza este principio de resolución y trabaja con cláusulas de Horn. Utiliza la unificación para intentar identificar las partes derecha e izquierda de las cláusulas de una forma lógica, investigando los valores de la variable que permitirán una identificación correcta.

A partir de la Regla de Resolución puede concebirse un sistema de demostración automática que evalúe los programas en lógica, mediante la aplicación reiterada de la Regla de Resolución.

La implementación de la Regla de Inferencia en Prolog se basa en los conceptos de Unificación y Backtracking que serán explicados con más detalle en la próxima sección.

### 3.7 PROLOG: LA UNIFICACIÓN

La unificación es el mecanismo mediante el cual las variables lógicas toman valor en Prolog.

Cuando una variable no tiene valor se dice que está libre. Pero una vez que se le asigna valor, éste ya no cambia, por eso se dice que la variable está ligada.

Ejemplo:

?-docente(X).

docente('alejandra') el valor Alejandra se liga a la variable X

nota(X).

nota(7) el valor 7 se liga a la variable X

Se dice que dos términos unifican cuando existe una posible ligadura (asignación de valor) de las variables, tal que ambos términos son idénticos sustituyendo las variables por dichos valores.

Por ejemplo:  $a(X,3)$  y  $a(4,Z)$  unifican dando valores a las variables: X vale 4, Z vale 3. Obsérvese que las variables de ambos términos entran en juego.





La unificación no debe confundirse con la asignación de los lenguajes imperativos puesto que representa la igualdad lógica. Muchas veces unificamos variables con términos directamente y de manera explícita (ya veremos cómo se hace esto).

Para saber si dos términos unifican podemos aplicar las siguientes normas:

- Una variable siempre unifica con un término, quedando ésta ligada a dicho término.

**Ejemplo:**  $\text{alumno\_regular}(X)$ :- $\text{alumno}(X)$ ,

$\text{aprobó\_primer\_parcial}(X)$ ,

$\text{aprobó\_segundo\_parcial}(X)$ .

?- $\text{alumno\_regular}(\text{juan})$ .

- Dos variables siempre unifican entre sí, además, cuando una de ellas se liga a un término, todas las que unifican se ligan a dicho término.
- Para que dos términos unifiquen, deben tener el mismo functor y la misma aridad. Después se comprueba que los argumentos unifican uno a uno manteniendo las ligaduras que se produzcan en cada uno.

**Ejemplo:**  $\text{enseña}(\text{alejandra}, \text{juan})$ .

?- $\text{enseña}(\text{Quien}, X)$ .

Donde: Quien se unifica con el valor alejandra, y X con juan.

- Si algún término no unifica, ninguna variable queda ligada.

**Ejemplo:** Definición de hechos y reglas:

$\text{alumno}(\text{jose})$ .

$\text{alumno}(\text{ana})$ .

$\text{alumno}(\text{juan})$ .

$\text{aprobo\_primer\_parcial}(\text{jose})$ .

$\text{aprobo\_primer\_parcial}(\text{juan})$ .

$\text{aprobo\_primer\_parcial}(\text{ana})$ .

$\text{aprobo\_segundo\_parcial}(\text{jose})$ .



aprobo\_segundo\_parcial(ana).

alumno\_regular(X):-alumno(X),

aprobo\_primer\_parcial(X),

aprobo\_segundo\_parcial(X).

**Consultas:**

?- alumno\_regular(jose).

true

?- alumno\_regular(juan).

false

?- alumno\_regular(X).

X = jose

X = ana

2 soluciones.



### 3.8 PROLOG: BÚSQUEDA DE SOLUCIONES

Una llamada concreta a un predicado, con unos argumentos concretos, se denomina **objetivo** (en inglés, goal). Todos los objetivos tienen un resultado de éxito o fallo tras su ejecución indicando si el predicado es cierto para los argumentos dados, o por el contrario, es falso.

Cuando un objetivo tiene éxito las variables libres que aparecen en los argumentos pueden quedar ligadas. Estos son los valores que hacen cierto el predicado. Si el predicado falla, no ocurren ligaduras en las variables.

**Ejemplo:** Se definen los siguientes hechos:

docente(ana).

docente(juan).

docente(pedro).

El caso más básico es aquél que no contiene variables:

?- docente(ana).

Verdadero

?- docente(maria).

Falso

Si utilizamos una variable libre:

?- docente (X).

es posible que existan varios valores para dicha variable que hacen cierto el objetivo.

X =ana

X = juan

X = pedro

En este caso obtenemos todas las combinaciones de ligaduras para las variables que hacen cierto el objetivo.



Las secuencias de objetivos o consultas tienen las siguientes características:

- Los objetivos se ejecutan secuencialmente por orden de escritura (es decir, de izquierda a derecha).
- Si un objetivo falla, los siguientes objetivos ya no se ejecutan. Además la conjunción, en total, falla.
- Si un objetivo tiene éxito, algunas o todas sus variables quedan ligadas, y por tanto, dejan de ser variables libres para el resto de objetivos en la secuencia.
- Si todos los objetivos tienen éxito, la conjunción tiene éxito y mantiene las ligaduras de los objetivos que la componen.

Partiendo de un objetivo a probar se busca las aseveraciones que pueden probar el objetivo. Este proceso de búsqueda de soluciones, se basa en dos conceptos: la Unificación y el Backtracking.

Como ya hemos visto, en el proceso de Unificación cada objetivo determina un subconjunto de cláusulas susceptibles de ser ejecutadas (puntos de elección). Prolog selecciona el primer punto de elección y sigue ejecutando el programa hasta determinar si el objetivo es verdadero o falso.

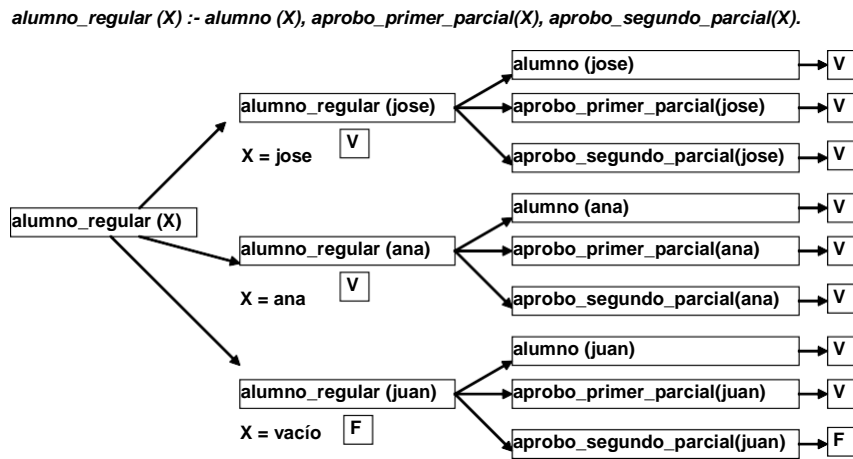
A continuación abordaremos como funciona el proceso de Backtracking.

### 3.9 BACKTRACKING

Es una técnica general que consiste en recorrer sistemáticamente todos los caminos posibles. Cuando un camino no conduce a la solución, se retrocede al paso anterior para buscar un nuevo camino.

El algoritmo de backtracking es utilizado para la resolución de diversas problemáticas, Prolog lo incorpora para su mecanismo de búsqueda de soluciones. En el caso de existir una solución segura la encuentra, el problema es el tiempo de procesamiento.

Las etapas por las que pasa el algoritmo se pueden expresar mediante un árbol de expansión (ímplicito en el algoritmo)





Es posible evidenciar la manera que funciona el proceso de backtracking usando comandos que dispone Prolog como: `trace` (consola) y `gtrace` (forma gráfica).

Las características generales del proceso de backtracking son:

- Cada solución es el resultado de una secuencia de decisiones.
- Las decisiones pueden deshacerse ya sea porque no lleven a una solución o porque se quieran explorar todas las soluciones (para obtener la solución óptima)
- Existe una función objetivo que debe ser satisfecha u optimizada por cada selección
- Las etapas por las que pasa el algoritmo se pueden representar mediante un árbol de expansión.
- El árbol de expansión no se construye realmente, sino que está implícito en la ejecución del algoritmo.
- Cada nivel del árbol representa una etapa de la secuencia de decisiones.

El funcionamiento del backtracking procede de la siguiente manera:

- Cuando se va ejecutar un objetivo, Prolog sabe de antemano cuantas soluciones alternativas puede tener. Cada una de las alternativas se denomina punto de elección. Dichos puntos de elección se anotan internamente y de forma ordenada. Para ser exactos, se introducen en una pila.
- Se escoge el primer punto de elección y se ejecuta el objetivo eliminando el punto de elección en el proceso.
- Si el objetivo tiene éxito se continúa con el siguiente objetivo aplicándole estas mismas normas.
- Si el objetivo falla, Prolog da marcha atrás recorriendo los objetivos que anteriormente sí tuvieron éxito (en orden inverso) y deshaciendo las ligaduras de sus variables. Es decir, comienza el backtracking.
- Cuando uno de esos objetivos tiene un punto de elección anotado, se detiene el backtracking y se ejecuta de nuevo dicho objetivo usando la solución alternativa. Las variables se ligan a la nueva solución y la ejecución continúa de nuevo hacia adelante. El punto de elección se elimina en el proceso.
- El proceso se repite mientras haya objetivos y puntos de elección anotados. De hecho, se puede decir que un programa Prolog ha terminado su ejecución cuando no le quedan puntos de elección anotados ni objetivos por ejecutar en la secuencia.



### 3.10 EL CONTROL DEL BACKTRACKING:

El Backtracking se puede controlar mediante el uso de dos predicados:

- El corte (!)
- El fail

#### 3.10.1 EL PREDICADO CORTE

El corte es un predicado predefinido que no recibe argumentos. Se representa mediante un signo de admiración (!). Es un predicado que siempre se cumple, que genera un resultado verdadero en la primera ejecución, y falla en el proceso de *backtracking*, impidiendo dicho retroceso.

El corte tiene la propiedad de eliminar los puntos de elección del predicado que lo contiene. Es decir, cuando se ejecuta el corte, el resultado del objetivo (no sólo la cláusula en cuestión) queda comprometido al éxito o fallo de los objetivos que aparecen a continuación. Es como si a Prolog "se le olvidase" que dicho objetivo puede tener varias soluciones.

Otra forma de ver el efecto del corte es pensar que solamente tiene la propiedad de detener el *backtracking* cuando éste se produce. Es decir, en la ejecución normal el corte no hace nada. Pero cuando el programa entra en *backtracking* y los objetivos se recorren marcha atrás, al llegar al corte el *backtracking* se detiene repentinamente forzando el fallo del objetivo.

El corte se utiliza muy frecuentemente, cuanto más diestro es el programador más lo suele usar.

Los motivos por los que se usa el corte son, por orden de importancia, son los siguientes:

- **Para optimizar la ejecución.** El corte sirve para evitar que por culpa del *backtracking* se exploren puntos de elección que, con toda seguridad, no llevan a otra solución (fallan). Para los entendidos, esto es podar el árbol de búsqueda de posibles soluciones.
- **Para facilitar la legibilidad y comprensión del algoritmo** que está siendo programado. A veces se sitúan cortes en puntos donde, con toda seguridad, no van a existir puntos de elección para eliminar, pero ayuda a entender que la ejecución sólo depende de la cláusula en cuestión.
- **Para implementar algoritmos diferentes según la combinación de argumentos de entrada.** Algo similar al comportamiento de las sentencias case en los lenguajes imperativos.



- **Para conseguir que un predicado solamente tenga una solución.** Esto nos puede interesar en algún momento. Una vez que el programa encuentra una solución ejecutamos un corte. Así evitamos que Prolog busque otras soluciones aunque sabemos que éstas existen.

### 3.10.2 EL PREDICADO FAIL

Es un predicado predefinido, sin argumentos que siempre falla, por lo tanto, implica la realización del proceso de retroceso (backtracking) para que se generen nuevas soluciones.

Cuando la *máquina Prolog* encuentra una solución se detiene y devuelve el resultado de la ejecución. Con *fail* podemos forzar a que no se detenga y siga construyendo el árbol de búsqueda hasta que no queden más soluciones que mostrar.

### 3.10.3 EL CONTROL DEL BACKTRACKING: EJEMPLO DE CORTE Y FALLO.

<pre>/*Hechos y reglas*/ alumno(ana). alumno(juan). alumno(pedro). alumnos(X) :- alumno(X). unAlumno(X) :- alumno(X),!. losAlumnos(X) :- alumno(X), write(X), nl, fail.</pre>	<pre>?- alumnos(X). X = ana ; X = juan ; X = pedro. ?- unAlumno(X). X = ana. ?- losAlumnos(X). ana juan pedro false.</pre>
---	--





## 4. SWI-PROLOG

SWI-Prolog es una implementación en código abierto (en inglés, open source) del lenguaje de programación Prolog. Su autor principal es Jan Wielemaker. En desarrollo ininterrumpido desde 1987, SWI-Prolog posee un rico conjunto de características, bibliotecas (incluyendo su propia biblioteca para GUI, XPCE), herramientas (incluyendo un IDE) y una documentación extensiva. SWI-Prolog funciona en las plataformas Unix, Windows y Macintosh.

El nombre SWI deriva de Social-Wetenschappelijke Informatica ("Informática de Ciencias Sociales"), el antiguo nombre de un grupo de investigación en la Universidad de Ámsterdam en el que Wielemaker está integrado. El nombre de ese grupo se cambió posteriormente a HCS (Human-Computer Studies).

Prolog es un lenguaje de programación seminterpretado. Su funcionamiento es muy similar a Java. El código fuente se compila a un código de byte el cuál se interpreta en una máquina virtual denominada Warren Abstract Machine (comúnmente denominada WAM).

Por eso, un entorno de desarrollo Prolog se compone de:

- Un **compilador**. Transforma el código fuente en código de byte. A diferencia de Java, no existe un estándar al respecto. Por eso, el código de byte generado por un entorno de desarrollo no tiene por qué funcionar en el intérprete de otro entorno.
- Un **intérprete**. Ejecuta el código de byte.
- Un **shell o top-level**. Se trata de una utilidad que permite probar los programas, depurarlos, etc. Su funcionamiento es similar a los interfaces de línea de comando de los sistemas operativos.
- Una **biblioteca de utilidades**. Estas bibliotecas son, en general, muy amplias. Muchos entornos incluyen (afortunadamente) unas bibliotecas estándar-ISO que permiten funcionalidades básicas como manipular cadenas, entrada/salida, etc.

Generalmente, los entornos de desarrollo ofrecen extensiones al lenguaje como pueden ser la programación con restricciones, concurrente, orientada a objetos, etc.

SICStus, CIAO Prolog, y posiblemente otros más, ofrecen entornos integrados generalmente basados en Emacs que resultan muy fáciles de usar. CIAO Prolog además ofrece un autodocumentador similar al existente para Java además de un preprocesador de programas.

## 4.1 ENTORNO DE SWI-PROLOG

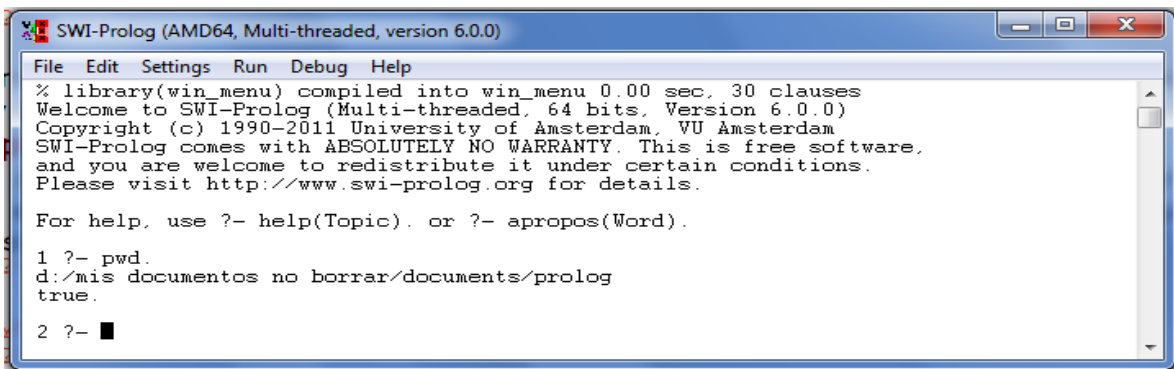
Está basado en dos ventanas:

- La ventana principal, con una línea donde se ejecutan los objetivos
- La ventana de edición, donde se editan y compilan los programas

La ventana principal siempre existe, la de edición sólo cuando se está editando el código de un programa.

Podemos usar tres predicados basados en Unix:

**pwd:** nos permite visualizar el directorio actual de trabajo. Al preguntar a Prolog por el directorio actual obtenemos lo que se muestra en la Figura 1.



```
SWI-Prolog (AMD64, Multi-threaded, version 6.0.0)
File Edit Settings Run Debug Help
% library(win_menu) compiled into win_menu 0.00 sec, 30 clauses
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.0.0)
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

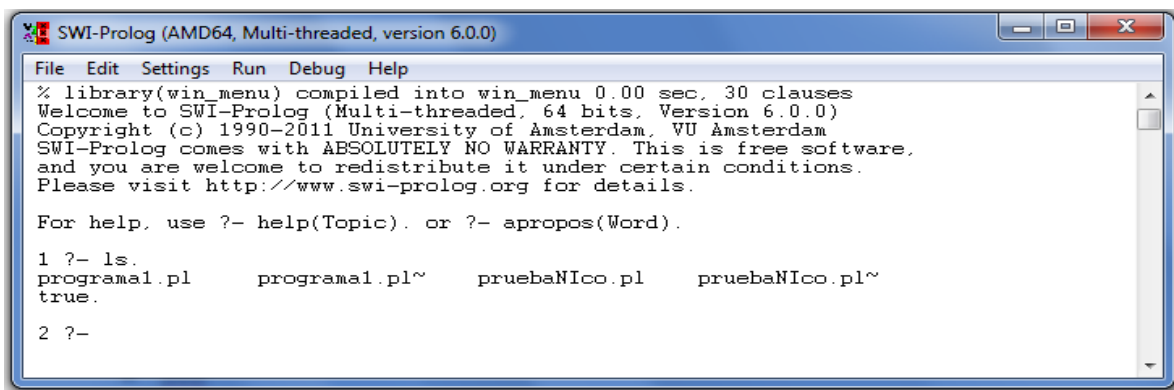
For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- pwd.
d:/mis documentos no borrar/documents/prolog
true.

2 ?- █
```

FIGURA 1: VENTANA PRINCIPAL DE SWI-PROLOG

**ls:** ver el contenido de un directorio.



```
SWI-Prolog (AMD64, Multi-threaded, version 6.0.0)
File Edit Settings Run Debug Help
% library(win_menu) compiled into win_menu 0.00 sec, 30 clauses
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.0.0)
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

1 ?- ls.
programa1.pl      programa1.pl~    pruebaNIco.pl   pruebaNIco.pl~
true.

2 ?-
```

FIGURA 2: EJEMPLIFICACIÓN DEL COMANDO LS

**cd:** nos movemos al directorio de conexión del usuario.

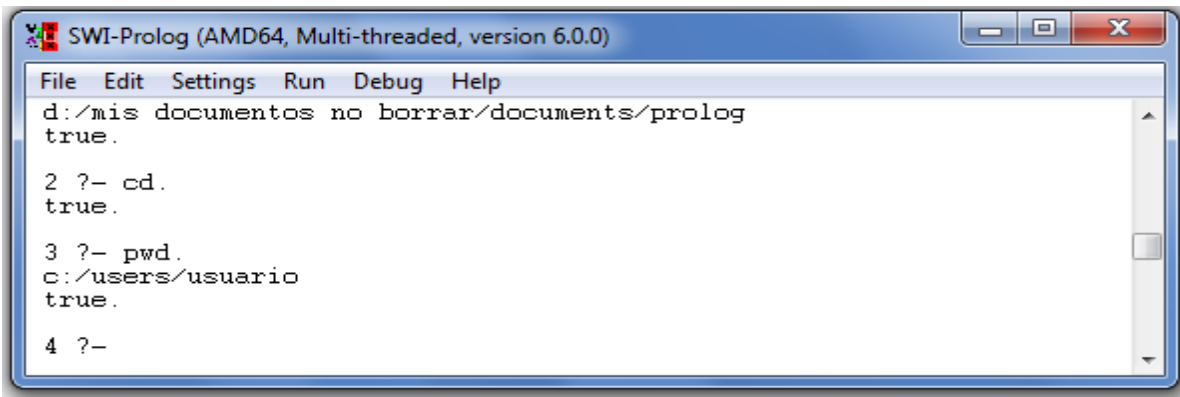


FIGURA 3: EJEMPLIFICACIÓN DEL COMANDO PWD (VENTANA PRINCIPAL)

Al verificarlo con el comando `pwd`, podemos ver que efectivamente nos hemos desplazado al directorio de conexión del usuario.

También podemos movernos a otro directorio el cual su ruta deberá ser especificada como argumento del comando `cd`.

#### Procedimiento para la creación de un programa nuevo.

Ejemplo:

`?- edit(file('ejemplo.pl')).`

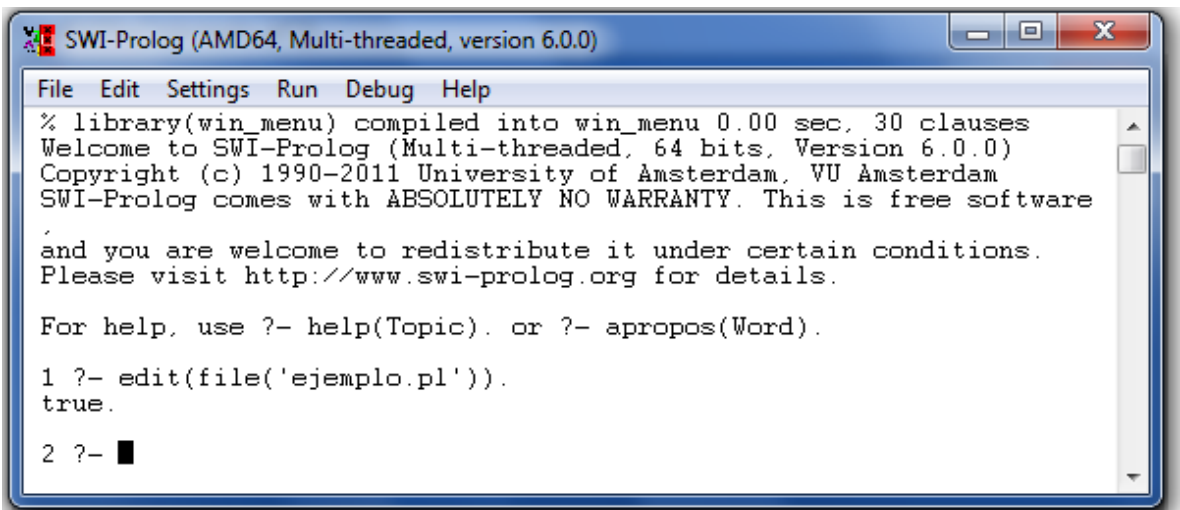


FIGURA 4: EJEMPLIFICACIÓN DEL COMANDO EDIT

- El functor **file** es necesario para crear un programa nuevo.

- La extensión debe ser **.pl**.
- La ruta se especifica entre comillas simples.
- El fichero no debe existir.
- El fichero creado será un fichero de texto Unix (aunque usemos Windows)

Se abre un editor de texto en el cual podremos escribir nuestro programa el prolog.

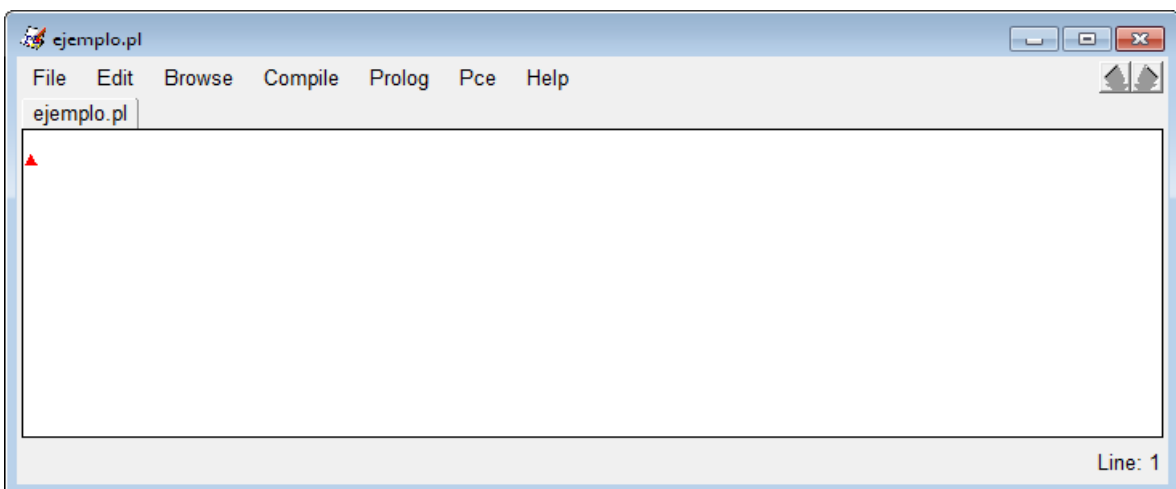


FIGURA 5: VENTANA DE EDICIÓN

**Escribimos nuestro programa:**

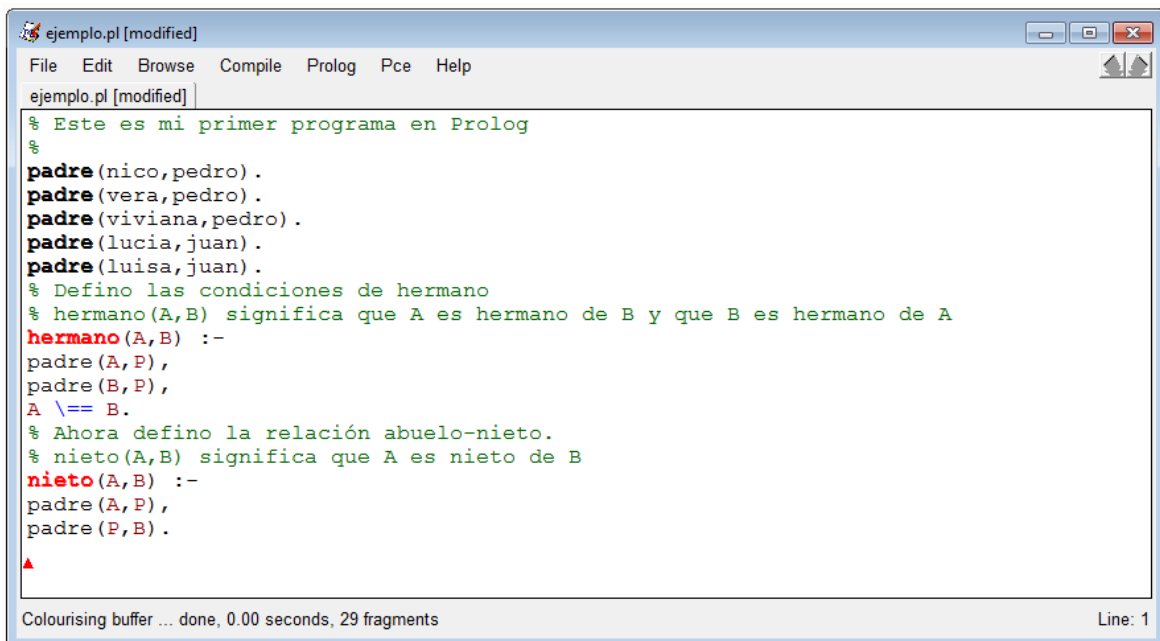


FIGURA 6: EJEMPLO DE PROGRAMA EN SWI-PROLOG

### Procedimiento para la compilación.

En el menú del editor se debe seleccionar: Compile/Compile Buffer

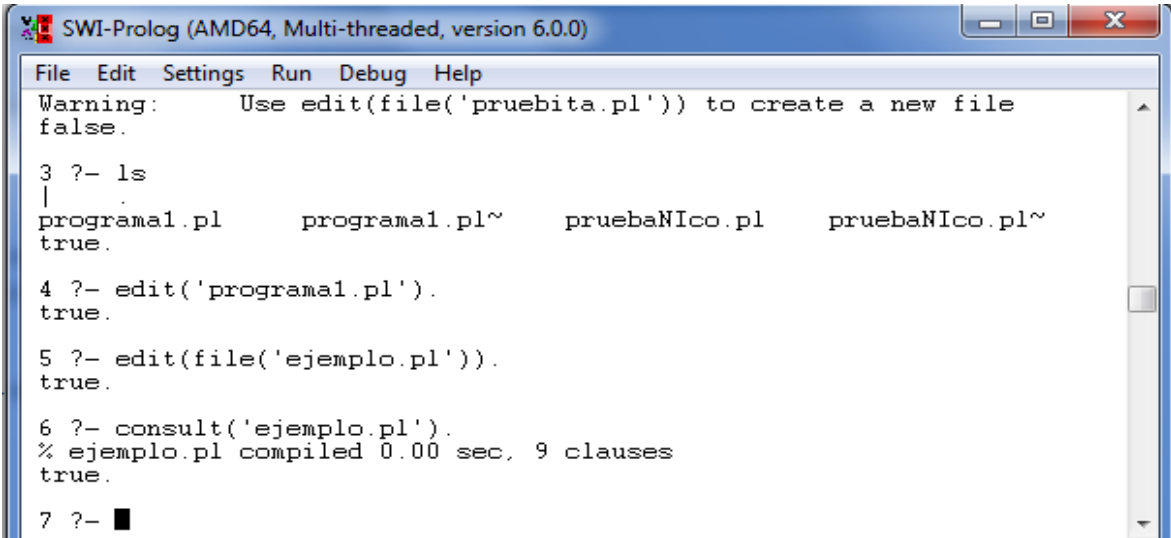
Si hay errores aparecen en una ventana emergente.

Si el programa ha sido modificado se ofrece la posibilidad de guardarlo antes de compilarlo.

### Procedimiento para la ejecución.

En la ventana principal ejecutamos el siguiente comando: `consult`, especificando entre paréntesis y comillas simples, el nombre de nuestro programa.

`consult('ejemplo.pl').`

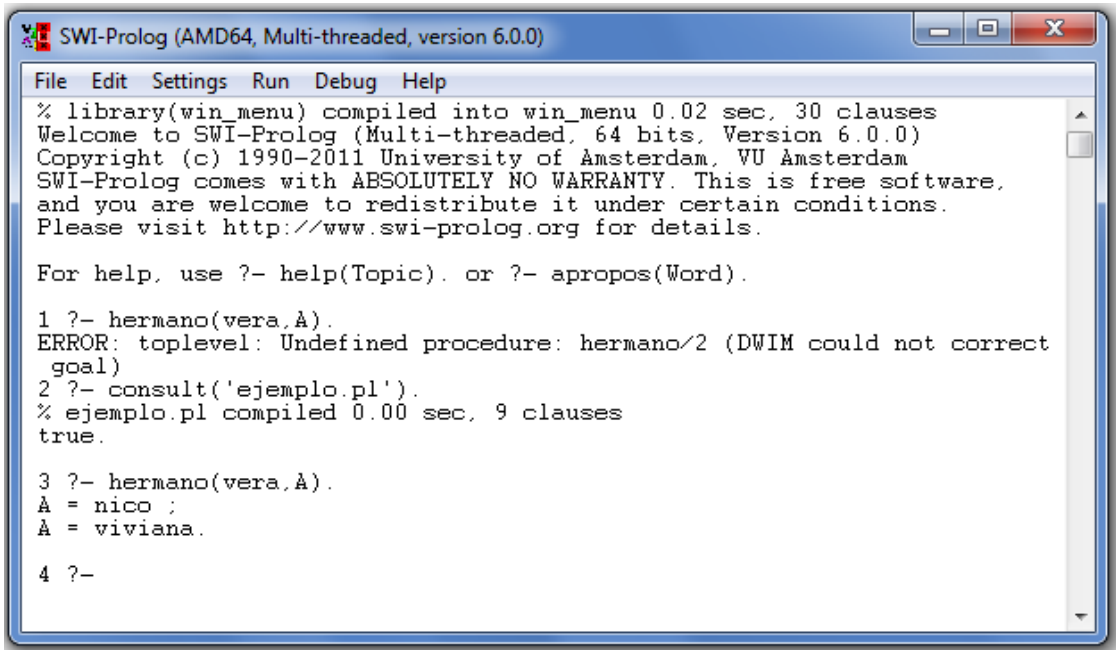


```
SWI-Prolog (AMD64, Multi-threaded, version 6.0.0)
File Edit Settings Run Debug Help
Warning: Use edit(file('pruebita.pl')) to create a new file
false.
3 ?- ls
|
programa1.pl      programa1.pl~   pruebaNico.pl   pruebaNico.pl~
true.
4 ?- edit('programa1.pl').
true.
5 ?- edit(file('ejemplo.pl')).
true.
6 ?- consult('ejemplo.pl').
% ejemplo.pl compiled 0.00 sec, 9 clauses
true.
7 ?- █
```

FIGURA 7: EJEMPLIFICACIÓN DEL COMANDO CONSULT.

Desde la ventana principal, podremos ejecutar los predicados. Por ejemplo, quiero saber quiénes son hermanos de vera, debo ejecutar:

`?-hermano(vera,A).`



```
SWI-Prolog (AMD64, Multi-threaded, version 6.0.0)
File Edit Settings Run Debug Help
% library(win_menu) compiled into win_menu 0.02 sec, 30 clauses
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.0.0)
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

1 ?- hermano(vera,A).
ERROR: toplevel: Undefined procedure: hermano/2 (DWIM could not correct
goal)
2 ?- consult('ejemplo.pl').
% ejemplo.pl compiled 0.00 sec, 9 clauses
true.

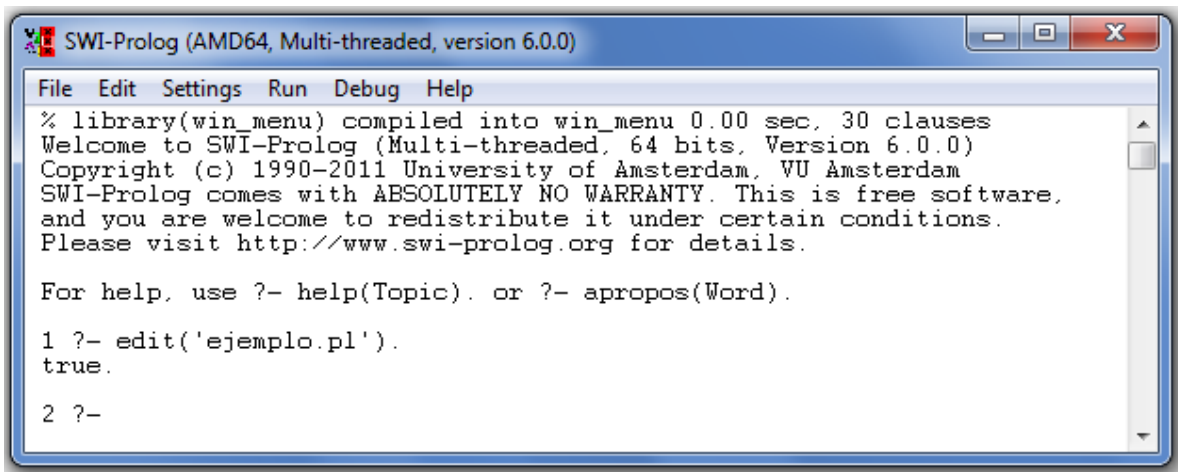
3 ?- hermano(vera,A).
A = nico ;
A = viviana.

4 ?-
```

FIGURA 8: EJEMPLO DE CONSULTAS A UN PROGRAMA LÓGICO

Para seguir viendo más resultados, debo presionar ; (punto y coma).

Si quisiéramos modificar un programa ya creado, utilizamos el comando edit, pero sin el functor file(), de la siguiente forma: **edit('ejemplo.pl')**.



```
SWI-Prolog (AMD64, Multi-threaded, version 6.0.0)
File Edit Settings Run Debug Help
% library(win_menu) compiled into win_menu 0.00 sec, 30 clauses
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.0.0)
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

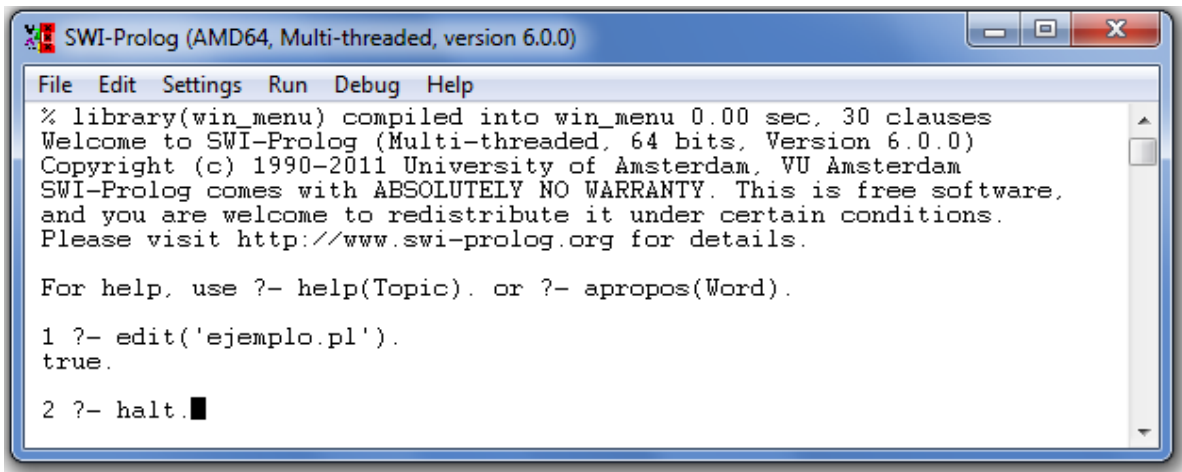
For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- edit('ejemplo.pl').
true.

2 ?-
```

FIGURA 9: EJEMPLIFICACIÓN DEL COMANDO PARA EDICIÓN DE UN ARCHIVO

Finalmente, para salir utilizamos el comando: **halt**.



```
SWI-Prolog (AMD64, Multi-threaded, version 6.0.0)
File Edit Settings Run Debug Help
% library(win_menu) compiled into win_menu 0.00 sec, 30 clauses
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.0.0)
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

1 ?- edit('ejemplo.pl').
true.

2 ?- halt.█
```

FIGURA 10: EJEMPLIFICACIÓN DEL COMANDO HALT

## 4.2 APRECIACIONES FINALES DEL LENGUAJE PROLOG

De lo expuesto en secciones anteriores, se puede establecer que el lenguaje Prolog está orientado a la Inteligencia Artificial, usando la programación lógica. Gracias a su facilidad de programar y su sencilla sintaxis gramatical y numérica, se pueden escribir rápidamente y con pocos errores programas claramente legibles, además cualquier usuario puede acceder a él si lo desea y sin problemas de entendimiento.

También utiliza pocos comandos en comparación con otros lenguajes de programación. Por otra parte en este lenguaje al igual que otros, hay que tener en cuenta la asociatividad de los operadores antes de trabajar con él.

Además Prolog se puede trabajar en diferentes sistemas operativos, tales como UNIX, MS-DOS, MAC-OS, entre otros.



## 5. OBJETOS COMPUESTOS O ESTRUCTURAS.

El Prolog, además de trabajar con objetos simples, permite trabajar con argumentos más complejos. . Es una forma de introducir datos estructurados en Prolog.

Los objetos compuestos están formados por un functor y un conjunto de argumentos.

Los hechos con objetos compuestos son de la forma:

***predicado (arg\_1, arg\_2, ..., arg\_N)***

Ejemplo.

***trabaja (lara, panadería)***

***trabaja (vera, 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 (lara, panadería (sucursal, 12))***

***trabaja (vera, 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 “funtores” 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.

***trabaja (lara, panadería (sucursal, 12)) .***

***trabaja (vera, taller (sección, 97, c)) .***





Ahora veremos qué obtenemos con algunos objetivos:

?- trabaja(Quién,Dónde) .

Quién = lara,

Dónde = panadería(sucursal, 12) ;

Quién = vera,

Dónde = taller(sección, 97, c) .

?- trabaja(Quién,taller(En,Nro,Categoría)) .

Quién = vera,

En = sección,

Nro = 97,

Categoría = c.

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

En el ejemplo que se expone a continuación se puede ver otra utilidad a la hora de aplicar términos compuestos. **Ejemplos:** dado el registro Persona: Nombre, Apellido, DNI. Con los datos:

**Persona 1:** Juan, Perez, 123456789.

**Persona 2:** Pedro, Lopez, 223456789.

**En Prolog:**

persona(nombre('Juan'), apellido('Perez'), dni( 123456789)).

persona(nombre('Pedro'), apellido('López'), dni( 223456789)).

**Consulta:** ?- persona( nombre(X), apellido(Y), \_).

X='Juan', Y='Perez';

X='Pedro', Y='López';

## 6. RECURSIVIDAD

Las reglas que hemos visto hasta ahora estaban definidas en función de otras reglas ya existentes. La recursión es una técnica que debe ser tenida en cuenta cuando se quiere programar en Prolog. Se basa en definir relaciones en términos de ellas mismas.

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.

En general en una definición recursiva, es necesario considerar dos casos:

- **Caso Básico:** Momento en que se detiene el proceso o computo.
- **Caso Recursivo:** Suponiendo que ya se ha solucionado un caso más simple, se descompone el caso actual hasta llegar al caso más simple.

Por ejemplo, las siguientes reglas y hechos, expresan la idea de que se puede hablar de alguien si conocemos a ese alguien o si conocemos a alguien que nos habla de él.

regla1: habla\_de(Uno,Otro) :- conoce(Uno,Otro),!.

regla2: habla\_de(Uno,Otro) :- conoce(Uno,AlguienMas) , habla\_de(AlguienMas,Otro).

regla3: conoce(juan,maria).

regla4: conoce(maria,jose).

regla5: conoce(maria,ana).

regla6: conoce(pedro,juan).

Y supongamos que queremos consultar lo siguiente:

```
?- habla_de(X,Y).
```

¿Cuáles serán las respuestas que obtendremos?

```
X=juan, Y=maria;  
X=maria, Y=jose;  
X=maria, Y=ana;  
X=pedro, Y=juan;
```

Estas primeras respuestas tienen que ver con el orden que siguen los hechos en nuestro programa lógico. Pero cuando a la última respuesta le agregamos “;”, gracias al backtracking intentará buscar más por alguna rama aún no explorada.

Nótese que las respuestas dadas hasta ahora, se deducen de aplicar la regla1 y luego la regla3; para la segunda respuesta, se aplicaron las reglas 1 y 4; para la tercera respuesta se aplicaron las reglas 1 y 5; y para la última respuesta se aplicaron las reglas 1 y 6.

Hasta acá no se usó nunca la regla 2. Recién para dar la siguiente respuesta, aplica la regla 2 (dado que ya no tiene más combinaciones posibles de regla1 con algún hecho) y combinándola con, por Ej., la regla3, luego la regla1 y luego la regla4 obtenemos:

```
X=juan, Y=jose;
```

El resto de las respuestas serían:

```
X=juan, Y=ana;
```

```
X=pedro, Y=maria;
```

no

Pero aún podemos utilizar Prolog en su máxima potencia si trabajamos con estructuras infinitas o potencialmente infinitas, como podrían ser las listas. El poder de los programas lógicos está en su natural habilidad de manipular tipos de datos recursivos.

Otros ejemplos:

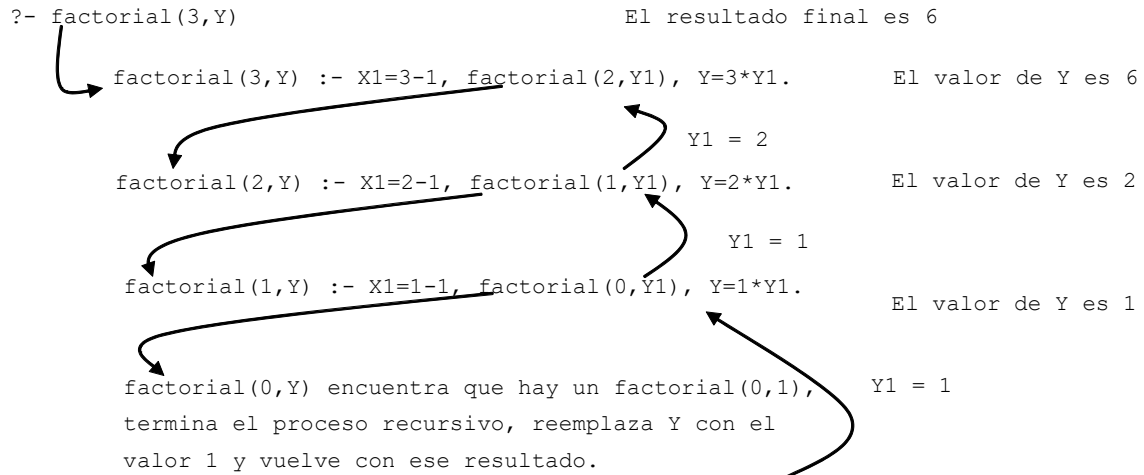
1. Obtener el factorial de un número.

Reglas

```
factorial(0,1) :-!.
```

```
factorial(X,Y) :- X1 is X-1, factorial(X1,Y1), Y is X*Y1.
```

Consulta



**Figura 11: Ejemplificación de Recursividad**

## 2. Obtener el descendiente de una persona.

### Reglas

```

Hijo_de(maria, carlos).
Hijo_de(carlos, cristina).
Hijo_de(cristina, luis).
Descendiente(X,Y) :- hijo(X,Y).
Descendiente(X,Y) :- hijo(X,Z), descendiente(Z,Y).
  
```

### Consulta

```

?- Descendiente(maria, luis)
true
  
```

## 7.LISTAS

Son estructuras de datos que almacenan y manipulan un conjunto de términos. Se trata de un par ordenado donde cada componente es un término, una lista o el término NIL (que es la lista vacía).

La manera más sencilla de escribir listas es enumerar sus elementos.

### Ejemplos:

La lista vacía se representa: []

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] ]

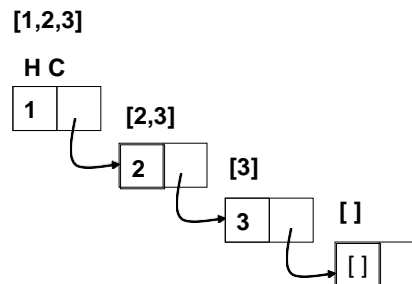
[a, | [b, c] ]

Un caso especial de esta notación es una lista con cabeza X y cola Xs, representada como [X|Xs] donde X: cabeza de la lista (primer elemento de la lista) y Xs: la cola (todos los elementos de la lista, exceptuando el primer elemento)

La cabeza puede usarse para extraer los componentes de una lista, de manera que no se requieran operadores específicos para extraer la cabeza y la cola.

La siguiente lista de terminos: [1, 2, 3 ] se representa como:

[ 1 | [ 2 | [ 3 | [ ] ] ] ] donde el termino constante 1 es la cabeza y la cola es la lista [ [ 2 | [ 3 | [ ] ] ] ].



**Figura 12: Ejemplificación de Listas**

A continuación se expone otro ejemplo para demostrar el uso de listas. Supongamos que disponemos de una base de datos que incluye una lista de materias el ciclo básico y del ciclo superior.

**Ciclo Básico**

Matemática

Física

Química

**Ciclo Superior**

análisis

diseño

programación



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

**Ejemplo:**

```
%Definición de hechos  
  
basica (matemática) .  
  
basica (química) .  
  
basica (física) .  
  
superior (análisis) .  
  
superior (diseño) .  
  
superior (programación) .
```

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

Trabajando con listas esto no es necesario, además las definiciones recursivas de “miembro” (predicado para el manejo de listas que será explicado en la siguiente sección) hacen lo mismo y en menos espacio.

El programa mostrado anteriormente tenía hechos simples. En el próximo ejemplo veremos la utilización de listas.

**Ejemplo:**

```
basica (X) :- member (X, [matemática, física, química]) .  
  
superior (Y) :- member (Y, [análisis, diseño, programación]) .
```

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 “superior (diseño)” obtendremos como resultado “TRUE”.

## 7.1 PREDICADOS SWI-PROLOG PARA EL MANEJO DE LISTAS



SWI-Prolog nos brinda un set de predicados útiles para poder manipular listas de elementos. Este es un módulo que se carga por defecto.

- **is list(+Term):** cierto si Term es una lista.
- **length(?List, ?Int):** Int es el número de elementos de la lista List.
- **sort(+List, -Sorted):** Sorted es la lista ordenada de los elementos de List sin duplicados.
- **append(?List1, ?List2, ?List3):** List3 es la concatenación de List1 y List2
- **member(?Elem, ?List):** Elem es elemento de List.
- **nextto(?X, ?Y, ?List):** Y está después de X en la lista List.
- **delete(+List1, ?Elem, ?List2):** List2 es la eliminación de todos los elementos que unifican simultáneamente con Elem de List1.
- **nth0(?Index, ?List, ?Elem):** Elem es el Index-ésimo elemento de List, comenzando por el 0.
- **reverse(+List1, -List2):** List2 es List1 pero con el orden de los elementos cambiado.
- **findall(termino\_o\_variable, objetivo, lista\_resultado):** genera una lista con todas las soluciones de un determinado predicado. Si no hay soluciones genera una lista vacía.

## 7.2 EJEMPLIFICACIÓN DE PREDICADOS PARA EL MANEJO DE LISTAS.

Predicado	Ejemplo
<b>is_list(unaLista)</b>	?- is_list([1,2,3]).  true
<b>length(UnaLista, Longitud)</b>	?- length([a,b,c,d],X).  X = 4.
<b>sort(UnaLista, ListaOrdenada)</b>	?- sort([20,5,25,10],Y).  Y = [5, 10, 20, 25].
<b>append(UnaLista, OtraLista, ListaConcatenada)</b>	?- append([20,4,5],[a,b,c],Z).  Z = [20, 4, 5, a, b, c].
<b>member(UnElemento, UnaLista)</b>	?- member(1,[4,5,6]).  false.
<b>findall(termino_o_variable, objetivo, lista_resultado)</b>	--Este ejemplo obtiene los nombres de los



--alumnos que tienen 20 años o más, y los

--almacena en una lista.

alumno('pepe',19).

alumno('tom',20).

findall(X,(alumno(X,E),E>=20),L).

**resultado->L=[tom].**

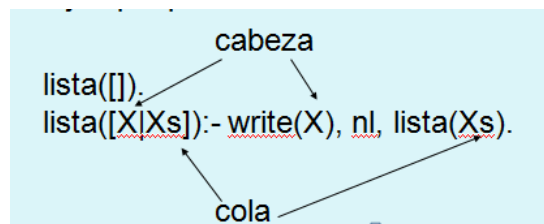
TABLA 2. EJEMPLO DE PREDICADOS PARA EL MANEJO DE LISTAS



### 7.3 RECURSIVIDAD EN LISTAS

Las listas son estructuras de datos definidas de manera recursiva. La mayoría de las definiciones identifican el caso básico con la lista vacía y el caso recursivo con  $[X|Xs]$ . (No siempre es así)

**Ejemplo:** Para mostrar los elementos de la lista, se plantea la siguiente regla o relación recursiva:



Si tenemos la siguiente consulta:

?- lista([1,2,3])

El árbol de prueba sería:

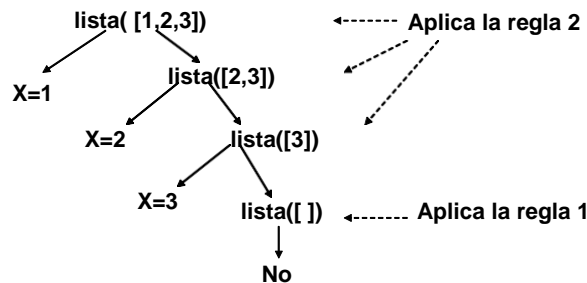


Figura 13: Ejemplificación de recursividad en listas.

#### Miembro de una lista

Una regla elemental es determinar si un elemento pertenece o no a la lista. Para ello, debemos programar un predicado miembro que dado un elemento nos responda “yes” si pertenece a la lista y “no” en otro caso.

El predicado sería:

miembro(X, [ X | \_ ]).

miembro(X, [ \_ | Z ]) :- miembro(X,Z).

- La primera regla : Indica X es miembro de la lista que tiene X como cabeza.
- La segunda expresa: X es miembro de una lista si X es miembro del resto de dicha lista

Si tenemos las siguientes consultas:

?-miembro(1,[1,2,3]).

Yes

?- miembro(X,[1,2,3]).

X=1;

X=2;

X=3;

no

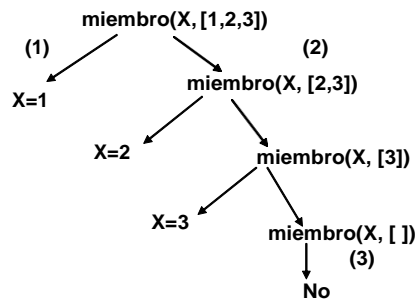


Figura 14: Ejemplificación de regla recursiva miembro.



### Referencias y Fuentes

- Lógica de Primer Orden, Lógica Computacional y Ampliación de Lógica. Autor: Llorens Largo Faraón; Castel de Haro Jesús. Departamento de Ciencia de Computación e Inteligencia Artificial. Universidad de Alicante (2001)
- Programación Práctica en Prolog. Autor: Labra G. José. Areas de Lenguajes y Sistemas Informáticos. Departamento de Informática. Universidad de Oviedo (1998)
- Lógica con Prolog. Autor: Césari Matilde. Universidad Tecnológica Nacional. Facultad Regional de Mendoza. Cátedra de Inteligencia Artificial.
- Apunte de Paradigmas de programación Universidad Tecnológica Nacional. Facultad Regional Buenos Aires.
- Curso básico, intermedio y avanzado de programación en Prolog. Autor: Fernandez Angel.

### Enlaces varios

- SWI Prolog: [www.swi-prolog.org](http://www.swi-prolog.org)
- Prolog Development Tools (ProDT): <http://prodevtools.sourceforge.net/>
- GeeWhiz: <http://hulles.supersized.org/>
- GNU Prolog: <http://www.gprolog.org/>
- Visual Prolog: <http://www.visual-prolog.com/>