LENGUAJES DE PROGRAMACIÓN

(Sesión 8)

4. PROGRAMACIÓN DECLARATIVA
4.3. Método de prueba por resolución
4.4. Estrategias de búsqueda y resolución

Objetivo: Familiarizarse con los distintos métodos de creación de algoritmos e implementar diferentes formas de resolución.

ESTRATEGIAS DE RESOLUCIÓN : SLD-RESOLUCIÓN

El resolvente obtenido al aplicar la regla de resolución de Robinson no es único, ya que puede haber más de una manera de escoger *L* y *M*. Además, la elección de las cláusulas padre a resolver tampoco es única. Estas elecciones son importantes y dan lugar a las *estrategias de resolución*. Hay por tanto dos decisiones a tomar al hacer la resolución:

1) la selección de las cláusulas a unificar y

2) la selección de los átomos a unificar dentro de esas cláusulas.

El punto 1 se conoce como *regla de búsqueda* y el punto 2 como *regla de selección.* Vamos en lo que sigue a estudiar qué estrategias utilizaremos en nuestras demostraciones automáticas.

Como ya hemos visto antes, a partir de este momento vamos a trabajar únicamente con Cláusulas de Horn Definidas, es decir, aquellas que tienen como máximo un átomo afirmado. Distintos estudios han demostrado que todo problema representado en cláusulas de Horn será resoluble si tenemos sólo una cláusula decapitada, y el resto con cabeza, es decir, si sólo existe una pregunta. Esto es fácil de ver de forma intuitiva:

- *Al menos una cláusula decapitada*: al aplicar el método de resolución a dos cláusulas de Horn con cabeza, obtenemos como resultado otra cláusula con cabeza, con lo que no podríamos derivar la cláusula vacía.

- *Sólo una cláusula decapitada*: de existir varias cláusulas decapitadas, cualquier prueba por resolución de una nueva cláusula puede ser convertida en una prueba usando como mucho una de ellas. Al resolver con la primera cláusula decapitada, obtenemos una nueva cláusula decapitada, y por tanto no necesitamos resolver con las otras cláusulas decapitadas.

Basándonos en este resultado definiremos una estrategia de resolución que partirá de la cláusula sin cabeza. Esta estrategia se le conoce como *Resolución lineal* y trata de resolver cláusulas con cabeza con cláusulas objetivo (sin cabeza).

Cláusulas Padre

cláusula objetivo *O* cláusula con cabeza *C*

$A_1, ..., A_k, ..., A_n$ A $B_1, B_2, ..., B_m$

$= UMG(A_k,A)$

$(A_1, ..., A_{k-1}, B_1, ..., B_m, A_{k+1}, ..., A_n)$

Cláusula Resolvente O'

Resolución Lineal con cláusulas de Horn Definidas

Como regla de selección utilizaremos la de unificar el átomo más a la izquierda de la cláusula objetivo. Así, la regla de resolución quedaría:

Cláusulas Padre

cláusula objetivo *O* cláusula con cabeza *C*

$A_1, A_2,..., A_n$ A $B_1, B_2, ..., B_m$

$= UMG(A_1,A)$

$(B_1, ..., B_m, A_2, ..., A_n)$

Cláusula Resolvente O'

Resolución Lineal con cláusulas de Horn Definidas

con elección del átomo más a la izquierda

Así, dado un conjunto de cláusulas definidas { C_1, C_2, ..., C_n, ... } (programa lógico *P*) y una cláusula negativa O (objetivo), una *derivación lineal* vendría dada en la siguiente figura :

Objetivos	*Programa Lógico*	*Sustituciones*
$O_0 = O$	C_1	σ_1
O_1	C_2	σ_2
O_2		
.	.	.
.	.	.
.	.	.
O_{n-1}	C_n	σ_n
O_n		
.	.	.
.	.	.

donde

O_0, O_1, ..., O_n, ... es una secuencia de objetivos

C_1, C_2, ..., C_n, ... es una secuencia de cláusulas del programa *P*

σ_1, σ_2, ..., σ_n, ... es una secuencia de unificadores más generales para O_{i-1} y C_i, respectivamente.

Ejercicio:

Realizar la derivación lineal del siguiente programa lógico :

```
P(a)
← P(b)
← Q(b)
← Q(c)
←
R(x,y) ← P(x), Q(y)
R(c,y) ← Q(y)
← R(b,c)
```

Objetivos	*Programa Lógico*		*Sustituciones*
R(b,c)	R(x,y)	P(x), Q(y)	{b/x, c/y}
P(b), Q(c)	P(b)		
Q(c)	Q(c)		
NADA			

Esta elección de una estrategia fija (SLD-Resolución) en el control de la búsqueda representa una desventaja frente a otros tipos de programación (LISP, CLISP,...) pero hace que el programador se despreocupe del control y se centre en la especificación del problema a resolver. Llamamos SLD-Resolución porque:

- utilizamos cláusulas de Horn **D**efinidas
- utilizamos la estrategia de búsqueda **L**ineal: primero en profundidad
- utilizamos una regla de **S**elección: primero a la izquierda

Al utilizar resolución lineal el programa lógico estará razonando hacia atrás desde el objetivo propuesto hasta que encuentre el modo de terminar utilizando las cláusulas del programa, es decir se parte de las metas a conseguir. Este método de razonamiento se denomina también *razonamiento dirigido al objetivo*. Esto resultará adecuado para sistemas interrogadores. Por contra, también existe el razonamiento hacia adelante, que parte de las cláusulas iniciales.

SISTEMAS DE REFUTACIÓN POR RESOLUCIÓN

Como ya hemos comentado antes, los sistemas basados en la resolución están concebidos para producir demostraciones por reducción al absurdo o **refutaciones**:

- Partimos de un conjunto de fórmulas bien formadas *C* a partir del cual deseamos demostrar cierta fbf *objetivo O*.
- Negamos la fbf objetivo ($\neg O$) y añadimos ésta al conjunto de fórmulas *C*
- Usando la resolución sobre este conjunto ampliado, intentaremos llegar a una contradicción, representada por la cláusula vacía *NADA*.
- Si hemos encontrado la cláusula vacía entonces podemos concluir que la fórmula objetivo *O* se deduce del conjunto de cláusulas *C*.

Ejemplo:

Utilizando el Método de Refutación por Resolución demostrar el siguiente argumento:

Cualquiera que puede cantar es cantante
Los pájaros no son cantantes

Algún pájaro tiene buena voz
Luego, Alguien que tiene buena voz no puede cantar

Formalizamos:

∃x (R(x) ♦ C(x))	R: poder cantar
∃x (P(x) ♦ ¬C(x))	C: ser cantante
∃x (P(x) ∧ V(x)) P:	ser pájaro
∧ ∃x (V(x) ∧ ¬R(x))	V: tener buena voz

Convertimos a Forma Clausal:

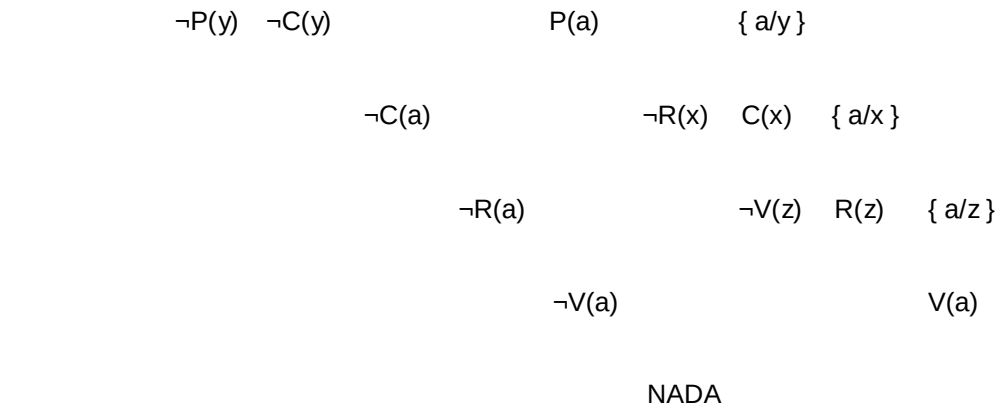
∃x (R(x) ♦ C(x)) ∧ ∃x (P(x) ♦ ¬C(x)) ∧ ∃x (P(x) ∧ V(x)) ∧ ¬∃x (V(x) ∧ ¬R(x))

C_1. ¬R(x) ∨ C(x)
C_2. ¬P(y) ∨ ¬C(y)
C_3. P(a)
C_4. V(a)
C_5. ¬V(z) ∨ R(z)

Aplicando Resolución vamos obteniendo las siguientes nuevas cláusulas:

C_6.	¬C(a)	de 2 y 3 con { a/y }
C_7.	¬R(a)	de 1 y 6 con { a/x }
C_8.	¬V(a)	de 5 y 7 con { a/z }
C_9.	NADA	de 4 y 8

Árbol de refutación:



Una importante propiedad formal que presenta la resolución es la de ser de *refutación completa*, es decir, si un conjunto de cláusulas no es *coherente*, la resolución podrá deducir de ellas la cláusula vacía. Así, si nuestras cláusulas (programa lógico) son coherentes, sólo tenemos que añadirles la negación de lo que queremos probar. Si por resolución deducimos la cláusula vacía, nuestro objetivo quedará probado. Pero como hemos comentado antes, nosotros utilizamos un tipo concreto de resolución, la SLD-Resolución, por lo que deberemos estudiar sus características particulares.

Hablaremos de *SLD-Refutación* cuando a partir de $C \quad \{O\}$ realizamos una SLD-derivación finita que tiene la cláusula vacía como el último objetivo de la derivación.

Ejemplo:

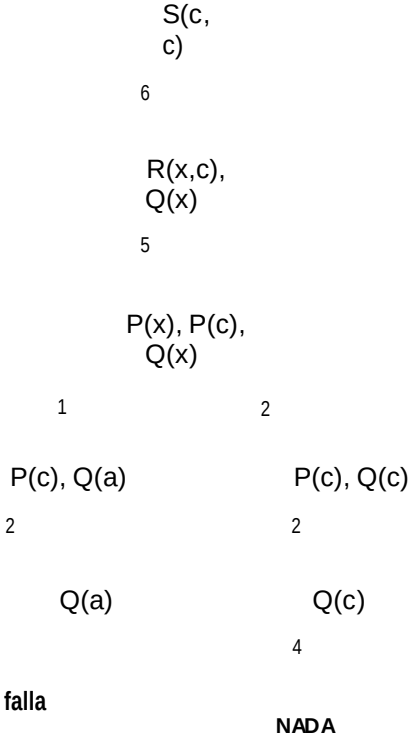
Necesitaremos alguna estrategia para buscar la cláusula vacía. La *reevaluación* o vuelta atrás ("backtracking") consiste en que al hacer la demostración, si llegamos a un punto en el que no podemos demostrar nada, volvemos hacia atrás sobre nuestros pasos y lo intentamos de nuevo por otro camino. Como hemos visto siempre buscamos el camino de la izquierda (estrategia de selección del átomo más a la izquierda), de forma que al iniciar la reevaluación subiremos en el árbol y buscaremos la alternativa de la derecha. Esta reevaluación se repetirá hasta que se encuentre un camino que nos lleve a la demostración que buscamos (en nuestro caso la cláusula NADA).

Ejemplo :

Sea el siguiente programa lógico P : P(a) ←
P(c)
← Q(b)
← Q(c)
←
R(x,y) · P(x), P(y)
S(c,y) · R(x,y), Q(x)

y el siguiente objetivo a demostrar
· S(c,c)

si desarrollamos el árbol de SLD-Refutación obtenemos :



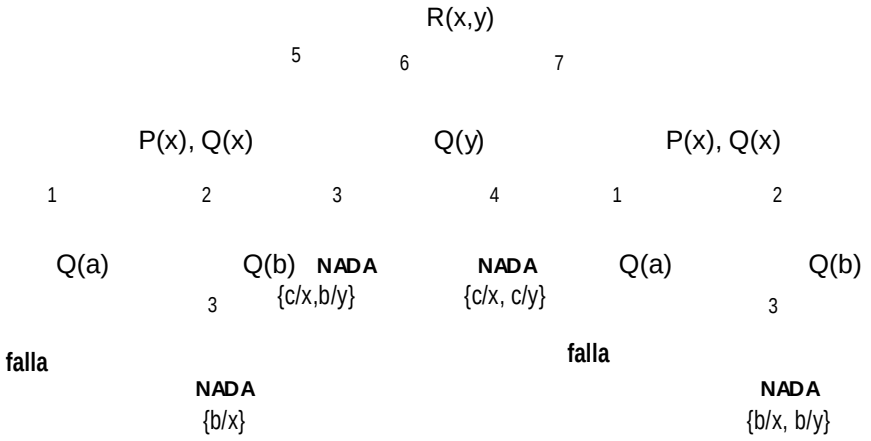
Como podemos ver en el ejemplo, si utilizamos la estrategia de primero en profundidad y el átomo más a la izquierda, la prueba falla (siguiendo el camino marcado por las cláusulas 6, 5, 1 y 2). Pero al reevaluar (deshacer el paso de la cláusula

2 y el de la cláusula 1 y tomar el camino alternativo con la cláusula 2) encuentra una demostración válida del objetivo (siguiendo el camino marcado por las cláusulas 6, 5, 2, 2 y 4). Veamos otro ejemplo un poco más complejo.

Ejemplo :

Sea el siguiente programa lógico P : P(a) ←
P(b)
← Q(b)
← Q(c)
←
R(x,y) · P(x), Q(x)
R(c,y) · Q(y)
R(x,x) · P(x), Q(x)

y el objetivo a demostrar
· R(x,y)
si desarrollamos el árbol de SLD-Refutación obtenemos :



En el árbol de SLD-Refutación anterior podemos ver que existen 6 posibles caminos para realizar la demostración, dos de los cuales terminan en "fallo" y cuatro con soluciones correctas con las correspondientes sustituciones. El mismo proceso de reevaluación entra en funcionamiento cuando queremos obtener más respuestas (respuestas alternativas) para un objetivo dado. Así, ante el ejemplo anterior nos daría una primera respuesta con *x=b*, y pararía el proceso; ante la solicitud de otra respuesta alternativa nos daría *x=c* y *y=b*. Y así sucesivamente.

Pero como ya hemos comentado el hecho de tomar una estrategia concreta de resolución (SLD-Resolución) tendrá repercusiones en la forma de operar. Veámoslo con un ejemplo.

Ejemplo :

Dado el siguiente programa lógico P :

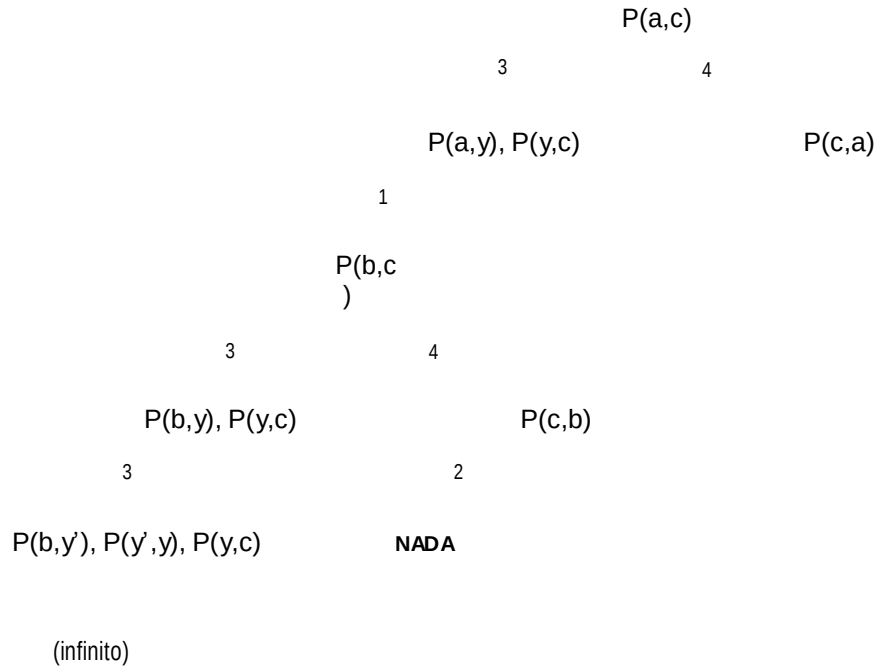
C_1 : P(a,b) ←

C_2 : P(c,b) ←

C_3 : P(x,z) · P(x,y), P(y,z)

C_4 : P(x,y) · P(y,x)

obtener el SLD-árbol para el objetivo · P(a,c)



Como podemos observar en este ejemplo, si la estrategia utilizada para la regla de búsqueda es el *primero en profundidad* y para la regla de selección el *átomo más a la izquierda*, la anterior derivación seria infinita, mientras que vemos claramente que existe un derivación que nos lleva a la cláusula vacía en pocos pasos. Esto ilustra claramente el porqué el orden de las cláusulas de un programa lógico es crucial cuando la estrategia de resolución utilizada es la SLD.

Al programar en lógica, los programas son nuestras hipótesis acerca del Universo, y las preguntas son como teoremas que se pretende demostrar. Una máquina lógica pone en marcha los mecanismos de inferencia lógica para buscar las respuestas a cualquier pregunta planteada. Si el programa contiene suficientes conocimientos para que puedan deducirse soluciones, estas le serán dadas al usuario.

OBTENCIÓN DE RESPUESTAS POR RESOLUCIÓN

Para realizar la contestación a una pregunta la idea clave es convertir la pregunta en una fbf objetivo que contenga un cuantificador existencial tal que la variable cuantificada existencialmente represente una respuesta a la pregunta. Vamos a comentar la forma de trabajo mediante un ejemplo. Supongamos que sabemos que "Pedro va a donde quiera que va Juan" y "Juan está en el colegio". Ahora nos planteamos la pregunta "¿Donde está Pedro?".

Formalización:

$\forall x$ [(Está(juan, x) $\Rightarrow$ Está(pedro, x))]
Está(juan, colegio)
$\exists x$ Está(pedro, x)

Negamos la conclusión y obtenemos la Forma Clausal:

¬∀x Está(pedro, x) ∗ ∀x ¬Está(pedro, x) ∗ ¬Está(pedro, x)

∀x [(Está(juan, x) ♦ Está(pedro, x))] ∗ ¬Está(juan, y) ∗ Está(pedro, y) (premisa)

Está(juan, colegio) (premisa)

Árbol de refutación:

¬Está(pedro, x) ¬Está(juan, y) Está(pedro, y) { x/y }

¬Está(juan, x) Está(juan, colegio) {colegio/x}

NADA

A continuación extraemos una **respuesta** a la pregunta dada mediante ese mismo árbol de refutación, pero con la siguiente modificación.

Proceso:

1.- Añadir a cada cláusula de las que origina la negación de la fbf objetivo, su propia negación. Así, ¬*Está(pedro, x)* da lugar a la tautología:

¬Está(pedro, x) Está(pedro, x)

2.- Siguiendo la estructura del árbol de refutación, realizar las mismas resoluciones que antes hasta encontrar una sola cláusula en la raíz.

3.- La cláusula raíz es una sentencia de respuesta.

¬Está(pedro, x) Está(pedro, x) ¬Está(juan, y) Está(pedro, y)

¬Está(juan, x) Está(pedro, x) Está(juan, colegio)

Está(pedro, colegio)

La única diferencia con la pregunta es que en la respuesta *se ha sustituido la variable* que teníamos por una constante. La respuesta por tanto es "Pedro está en el colegio".

La justificación teórica de esto sería que cuando hacemos un árbol de refutación, en la raíz obtenemos la cláusula vacía NADA y entonces sabemos que hay una incompatibilidad, o sea que el argumento es correcto. Obtenemos una respuesta si en lugar de esa cláusula encontramos una cláusula como respuesta. Como esta conversión se hace sustituyendo cada una de las cláusulas en que se convierte la negación de la fbf

objetivo por una tautología, *el árbol de demostración* obtenido es una demostración por resolución de que la sentencia que figura en la raíz se sigue lógicamente de las premisas más la tautología. De ahí que también se sigue de las premisas tan solo.

Ejemplo:

Premisa 1: Para todo x y para todo z, si x es padre de y e y es padre de z, entonces x es abuelo de z
Premisa 2: Cualquier persona tiene padre
Pregunta: ¿ Existen individuos x e y tales que x es abuelo de y ?

Formalización: $\forall x \forall z [\forall y (P(x, y) \rightarrow P(y, z)) \rightarrow A(x, z)]$
$\forall y \forall x P(x, y)$
$\forall x \forall y A(x, y)$

Demostramos esta fórmula por refutación mediante resolución:

Negación del objetivo: $\neg[\forall x \forall y A(x, y)] \leftrightarrow \forall x \forall y \neg A(x, y)$

Obtención de las cláusulas: $\neg P(x, y) \vee \neg P(y, z) \vee A(x, z)$ premisa 1
$P(f(w), w)$ premisa 2
$\neg A(u, v)$ pregunta negada

$\neg A(u, v)$	$\neg P(x, y)$	$\neg P(y, z)$	$A(x, z)$	{ u/x, v/z }
$\neg P(u, y)$	$\neg P(y, v)$		$P(f(w), w)$	{ f(v)/y, v/w }
	$\neg P(u, f(v))$		$P(f(w), w)$	{ f(f(v))/u, f(v)/w }

NADA

NOTA: la función f(_) se utiliza para representar el padre de un individuo.

El árbol de demostración modificado es el siguiente: la negación de la fbf objetivo se transforma en una tautología y las resoluciones se han realizado siguiendo las pautas de las del árbol anterior. Cada resolución del árbol modificado usa conjuntos de unificación que corresponden precisamente a los conjuntos de unificación del árbol de refutación.

Árbol de Demostración modificado:

¬A(u, v) A(u, v) ¬P(x, y) ¬P(y, z) A(x, z)

¬P(u, y) ¬P(y, v) A(u, v) P(f(w), w)

¬P(u, f(v)) A(u, v) P(f(w), w)

A(f(f(v)), v)

Esta cláusula de la raíz representa: ∃v A(f(f(v)), que es la sentencia respuesta. Esta proporciona una respuesta a la pregunta ¿Hay x e y tal que x sea abuelo de y?. La respuesta implica la definición de la función f (el padre de), y así, cualquier v y el padre del padre de v son ejemplos de individuos que satisfacen las condiciones de la pregunta.

En la práctica, la respuesta se puede obtener por la composición de las sustituciones realizadas en el proceso de unificación.

Ejemplo:
Si calculamos la composición de las sustituciones del ejemplo anterior:

④ = {u/x, v/z} o {f(v/y, v/w_1} o {f(f(v))/u, f(v)/w_2} = {f(f(v))/x, v/z, f(v)/y, v/w_1, f(f(v))/u, f(v)/w_2}

y como la pregunta era ¬A(u, v) tenemos que la respuesta es:

[¬A(u, v)] s = ¬A(f(f(v)), v)

SEMÁNTICA DECLARATIVA

Para realizar el estudio de la semántica declarativa caracterizaremos a un programa lógico *P* con el *Modelo Mínimo de Herbrand*. Para ello veamos previamente algunos conceptos.

La comprobación de que una fórmula del Cálculo de Predicados es insatisfacible exige realizar la verificación para todas las interpretaciones posibles, y como las variables toman valores en un dominio y puedo tomar infinitos dominios, esta tarea se convierte en muy costosa. Para reducir este estudio Herbrand [27] planteó un modelo que trabajaba sobre un dominio abstracto y estudiaba la inastisfacibilidad en dicho universo. Dicho dominio se conoce como universo de Herbrand.

El *Universo de Herbrand* (*H*) asociado a una fórmula (o un programa lógico) es el conjunto de todos los términos sin variables que se pueden construir con las

[27] Herbrand, J. *Recherches sur la théorie de la démonstration*. Thesis at the University of Paris. Traveaux de la Societé des Sciences de Varsoria, nº 33, 1930

constantes y los símbolos de función que aparecen en dicha fórmula (o programa lógico). Si el programa no contiene constantes introduciremos una artificial "*a*".

La *Base de Herbrand* (*B*) asociada a un programa lógico es el conjunto constituido por todas las fórmulas atómicas resultantes de combinar los símbolos de predicados que aparecen en el programa y los elementos del universo de herbrand.

Así, una *Interpretación de Herbrand* será un subconjunto de la base de Herbrand formado por todas las fórmulas atómicas de la base que se asumen como ciertas respecto a esa interpretación.

Si una interpretación de herbrand de un programa lógico hace ciertas todas las cláusulas de dicho programa, es decir es modelo, la llamaremos *Modelo de Herbrand* (*M*).

Ejemplo :

Sea el siguiente programa lógico

P(a)
← Q(b)
← R(c)
←
P(x) ← Q(x)
R(y) ← P(y)

El universo de Herbrand será : H = { a, b, c }

La base de Herbrand será : B = { P(a), P(b), P(c), Q(a), Q(b), Q(c), R(a), R(b), R(c) }

Un modelo de Herbrand será : M = { P(a), P(b), Q(a), Q(b), R(a), R(b), R(c) }

En cambio no lo será : M = { P(a), P(b), Q(a), Q(b), R(a), R(c) }

Podemos ver el significado de un programa lógico *P* como el dado por un modelo de herbrand de *P*. Pero algunos modelos son más grandes que lo estrictamente necesario. Por ello podemos tomar la intersección de todos los modelos de herbrand, que, evidentemente, también será un modelo para *P* y no existirá ningún modelo más pequeño.

Modelo de Herbrand Mínimo (M_P) : intersección de todos los modelos de Herbrand para ese programa

Se puede demostrar que M_P es el conjunto de todos los objetivos básicos (sin variables) que se pueden obtener por resolución usando las cláusulas del programa *P*, y por tanto de todas las fórmulas atómicas que son consecuencia lógica de *P*.

Algunas veces el modelo mínimo de herbrand no se puede obtener fácilmente a simple vista. Veamos un método recursivo para calcularlo. Para ello utilizaremos el operador de consecuencias inmediatas, definido de la siguiente manera

$$f_P(M) = M + \{\ A / A \leftarrow B_1, ..., B_n * \text{ y } \{B_1, ..., B_n\} \times M\ \}$$

$* A \leftarrow B_1, ..., B_n$ es una instancia básica (sin variables) de una cláusula del programa P

El modelo mínimo de herbrand será el menor punto fijo de f_P, es decir la menor solución de la ecuación $f_P(M) = M$, por lo que podemos partir del conjunto vacío y parar cuando no sea modificado.

Ejemplo :

El modelo de Herbrand mínimo del programa del ejemplo anterior lo calculamos, de manera recursiva con el operador de consecuencias inmediatas:

$M_0 = \{\}$
$M_1 = f_P(M_0) = \{ P(a), Q(b), R(c) \}$
$M_2 = f_P(M_1) = \{ P(a), Q(b), R(c), P(b), R(a) \}$
$M_3 = f_P(M_2) = \{ P(a), Q(b), R(c), P(b), R(a), R(b) \}$
$M_4 = f_P(M_3) = \{ P(a), Q(b), R(c), P(b), R(a), R(b) \} = M_3$

M_P = { P(a), Q(b), R(c), P(b), R(a), R(b) }

INTERPRETACIÓN DE LAS CLÁUSULAS DE HORN COMO LENGUAJE DE PROGRAMACIÓN

Podemos realizar una interpretación de las cláusulas de Horn desde un punto de vista más cercano a los lenguajes de programación. Si estamos intentando resolver un determinado problema, tenemos que:

los átomos negados $\{\neg N_1 , ..., \neg N_m\}$ son los *objetivos* o problemas a resolver. Si los átomos N_i contienen variables x_i, la interpretación que hacemos es que debemos hallar los valores de x_i que resuelvan los problemas N_i.

$$\leftarrow N_1 , ..., N_m$$

un átomo afirmado y varios átomos negados $\{A, \neg N_1 , ..., \neg N_m\}$ son interpretados como un *procedimiento* o *método* para resolver un subproblema. Así, ante un problema B que empareje con A con ④=UMG(A,B), el problema original se reduce a resolver los subproblemas $N_1$④ , ..., N_m④.

$$A \leftarrow N_1 , ..., N_m$$

un átomo afirmado {A} es interpretado como un procedimiento que resuelve de forma directa un determinado problema.

$$A \leftarrow$$

De forma general, cuando tenemos que resolver un determinado problema B, buscamos un procedimiento, directo ($A \leftarrow$) o no ($A \leftarrow N_1 , ..., N_m$), que empareje con él (④=UMG(A,B)). Esto tiene lugar en dos fases:

1. Una primera fase que afecta a las variables del procedimiento (N_1 , ..., N_m), de forma que se transfiere la *entrada* del problema *B* al procedimiento que trata de resolverlo. Tendremos por tanto un *unificador de entrada* θ_e.
2. Una segunda fase que afecta a las variables del problema a resolver (*B*), que transfiere la *salida* del procedimiento una vez resuelto. Tendremos por ello un *unificador de salida* θ_s.

Un *programa lógico* se compone de un conjunto de procedimientos (directos o hechos y compuestos o reglas) en forma de cláusulas de Horn. Su ejecución consistirá en su activación por medio de una pregunta objetivo o problema a resolver. Dicha pregunta irá activando distintos procedimientos para su resolución. Los programas lógicos son más abstractos que los programas convencionales ya que no incorporan control sobre los procedimientos invocados ni el orden en que deben realizarse. Los programas lógicos expresan únicamente la lógica de los métodos que resuelven el problema y son, por tanto, más fáciles de comprender, de verificar y de modificar.

Otra característica destacable de los programas lógicos viene dada por los parámetros de los procedimientos, que en este caso son argumentos de predicados y que por tanto no tienen dirección, es decir, igual pueden servir como parámetros de entrada como de salida dependiendo del contexto en el cual se haya invocado el procedimiento. Así, los parámetros dados en forma de constante en la llamada son considerados como parámetros de entrada y el resto de parámetros son computados como parámetros de salida.

Ejemplo:
Sea el siguiente programa lógico:

padresDe(clara,felix,chelo)
← padresDe(borja,felix,chelo)
←
hermanos(x,y) · padresDe(x,p,m), padresDe(y,p,m)

Si lanzamos el objetivo
· hermanos(clara,borja)

derivaremos la cláusula vacía, lo que indica que queda demostrado. Aquí, los dos argumentos, tanto “clara” como “borja”, son considerados como parámetros de entrada. Por otra lado, si lanzamos el objetivo
· hermanos(clara,x)
también derivaremos la cláusula vacía resultando la siguiente sustitución {borja/x}, es decir nos responde que si x=borja el problema queda solucionado. En este caso el primer argumento, “clara”, es considerado como parámetro de entrada y el segundo, x, es considerado como parámetro de salida, obteniéndose como respuesta x=borja.

En comparación con los programas convencionales, los programas lógicos con cláusulas de Horn son *no deterministas*, en el sentido de que:

- la estrategia de resolución por medio de la cual se van probando procedimientos alternativos no está determinada
- el orden de ejecución dentro de los subobjetivos no está determinada.

Esta propiedad la perdemos al tomar decisiones concretas a la hora de realizar la resolución (SLD-Resolución).

Para conseguir repeticiones de la misma orden, *estructura repetitiva* de los lenguajes de programación, mediante cláusulas de Horn deberemos utilizar procedimientos *recursivos*.

Así, para calcular el sumatorio de un número dado calculamos el sumatorio para el número anterior y le sumamos el número dado.:
sumatorio(n,s) ← restar(n,1,m), sumatorio(m,t), sumar(t,n,s)

Al tratarse de una llamada recursiva debemos tener cuidado de incluir el caso base (o condición de parada), que en este caso será que el sumatorio de 1 es 1:
sumatorio(1,1) ←

Para finalizar este apartado veremos algunas ventajas de los lenguajes de programación lógica frente a los lenguajes convencionales:

- Expresividad: un programa (base de conocimiento) escrito en un lenguaje de programación lógica puede ser leído e interpretado intuitivamente. Son, por tanto, más entendibles, manejables y fáciles de mantener.
- Ejecución y búsqueda incorporada en el lenguaje: dada una descripción en programación lógica válida de un problema, automáticamente se obtiene cualquier conclusión válida.
- Modularidad: cada predicado (procedimiento) puede ser ejecutado, validado y examinado independiente e individualmente. La programación lógica no tiene variables globales, ni asignación. Cada relación está autocontenida, lo que permite una mayor *modularidad*, *portabilidad* y *reusabilidad* de relaciones entre programas.
- Polimorfismo: se trata de lenguajes de programación sin tipos, lo que permite un alto nivel de abstracción e independencia de los datos (objetos).
- Manejo dinámico y automático de memoria.

PROLOG

El primero y más usado del los lenguajes de programación lógica es Prolog (“PROgrammation en LOGique”) diseñado e implementado alrededor de 1970 por

Colmerauer y otros miembros del Grupo de Inteligencia Artificial de Luminy en la Universidad de Aix-Marseille[28], apoyándose en trabajos de Kowalski[29] de la Universidad de Edimburgh. David Warren[30], de la Universidad de Edimburgh, desarrolló el primer compilador de Prolog (WAM – "Warren Abstract Machine"). Una descripción de Prolog ya considerada como estándar se puede encontrar en el libro de Clocksin y Mellish "Programación en Prolog".

Aunque se asocia Programación Lógica con Prolog, no es lo mismo, y el lenguaje Prolog sólo es un caso particular, existiendo distintos sistemas basados en otras lógicas o que incorporan nuevas características: MOLOG (Lógica Modal), GOEDEL (Lógica "many-sorted"), PARLOG (Programación Lógica Paralela), Prolog++ (Programación Lógica Orientada a Objetos), CLP (Programación Lógica con restricciones), ...

Prolog es una realización aproximada del modelo de computación de la Programación Lógica sobre una máquina secuencial. Desde luego no es la única realización posible, pero sí es la mejor elección práctica, ya que equilibra por un lado la preservación de las propiedades del modelo abstracto de la programación lógica, y por el otro lado consigue que la implementación sea eficiente. Prolog ha conseguido incrementar su eficiencia introduciendo ciertos predicados extralógicos, ofreciéndonos un sistema de programación práctico que tienen algunas de las ventajas de claridad y declarativiad que ofrecería un lenguaje de programación lógica y, al mismo tiempo, permite cierto control y operatividad.

Un sistema Prolog está basado en un comprobador de teoremas por Resolución usando cláusulas de Horn, y la estrategia de búsqueda primero en profundidad (resolución de entrada lineal) y la vuelta atrás ("backtracking"). La Base de Conocimientos de Prolog estará formada por cláusulas positivas o cláusulas con cabeza (*hechos* y *reglas*). Su ejecución consistirá en la introducción de una cláusula negada u *objetivo* que queremos hacer cumplir. Prolog razonará hacia atrás desde este objetivo hasta que encuentre el modo de terminar utilizando las cláusulas del programa

Ejemplo 1: (población en miles de habitantes y superficie en miles de km.^{2})

Hechos:

```
/* poblacion(Prov,Pob) <- la población, en miles de habitantes,
                          de la provincia Prov es Pob        */
poblacion(alicante,1149).
```

[28] A. Colmerauer. *Les Systèmes-Q, ou un formalisme pour analyser et synthétiser des phrases sur ordinateur*. Internal Report 43, Computer Science Dpt., Univesité de Montreal, septiembre 1970.
A. Colmerauer, H. Kanoui, P. Roussel y R. Pasero. *Un Systeme de Communication Homme-Machine en Français*, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973

[29] R. Kowalski. *Predicate Logic as a Programming Language*. En Proc. IFIP, Amsterdam, 1974

[30] D. Warren. *The runtime environment for a prolog compiler using a copy algorithm.* Technical Report 83/052, SUNY and Stone Brook, New York, 1983.

```
poblacion(castellon,432).
poblacion(valencia,2066).

/* superficie(Prov,Sup) <- la superficie, en miles de km2, de la
                           provincia Prov es Sup              */
superficie(alicante,6).
superficie(castellon,7).
superficie(valencia,11).
```

Reglas:

```
/* densidad(Prov,Den) <- la densidad de población,habitantes/km2,
                         de la provincia Prov es Den           */
densidad(X, Y)    :-    poblacion(X, P),
                        superficie(X, S),
                        Y is P/S.
```

Pregunta:

```
?- densidad(alicante, X).

X=191.5
```

Pregunta:

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

X=alicante
Y=191.5 ;

X=castellon
Y=61.7143 ;

X=valencia
Y=187.818 ;

No
```

Ejemplo 2:

Programa:

```
/* factorial(N,F) <- F es el factorial de N */
factorial(0,1).
factorial(N,F) :- N1 is N-1,
                  factorial(N1,F1),
                  F is N*F1.
```

Pregunta:

```
?- factorial(4,24).
Yes
```

Pregunta:

```
?- factorial(4,F).
F=24
```

LA LÓGICA EN LA VIDA

1. En un cuento de Gordon Dickson, "The Monkey Wrench" unos científicos consiguen salvar la vida inutilizando un ordenador. La técnica que emplearon fue decirle a la máquina: "Tienes que rechazar el enunciado que te estoy proponiendo, porque todos los enunciados que yo propongo son incorrectos".

2. A uno de los primeros programas de traducción entre el ruso y el inglés se le introdujo la frase "el espíritu está presto, pero la carne es débil". La tradujo al ruso y luego al inglés de nuevo, siendo el resultado "el vodka es agradable, pero la carne está demasiado blanda".

3. ***Test de Turing*** [31] para determinar razonablemente si una máquina piensa: un ordenador y un humano se ocultan a la vista del interrogador; este debe de tratar de averiguar cuál es el ordenador y cuál es el ser humano mediante el planteamiento de preguntas a cada uno de ellos; el humano responderá a las preguntas sinceramente tratando de persuadirle de que él es realmente el ser humano y el otro es el ordenador, y la computadora está programada para mentir intentando convencer al interrogador de que es el ser humano y el otro la máquina. Si el interrogador es incapaz de identificar de una forma definitiva al ser humano real, el ordenador ha superado la prueba.

4. ***La Sala China*** [32]: supongamos que tenemos un sistema que pasa la prueba de Turing. Dicho sistema podría semejarse a un sistema formado por un humano que únicamente entiende el castellano encerrado en una habitación con un libro de reglas. Mediante una apertura recibe del exterior unas papeletas con símbolos indescifrables, pero que localizados en el libro de reglas le permiten ejecutar una serie de instrucciones. La salida final es devuelta al exterior. Visto desde el exterior este sistema trabaja con oraciones escritas en chino como entrada y produce una respuesta también en chino. Pero, si el humano no entiende el chino, el libro de reglas no entiende chino ¿aquí nadie comprende el chino?

5. "Si el cerebro del ser humano fuera tan sencillo que lo pudiéramos entender, entonces seríamos tan estúpidos que tampoco lo entenderíamos"

   Jostein Gaarder, "El mundo de Sofía"

[31] Descrito por primera vez en el artículo "Computing Machinery and Intelligence" de Alan M. Turing publicado en 1950 en la revista "Mind". Reimpreso en "Computation & Intelligence", Collected Reading editados por George F. Luger, AAAI Press y The MIT Press, 1995

[32] Descrito por primera vez en el artículo "Minds, Brains, and Programs" de John R. Searle publicado en 1980 en la revista "The Behaviorial and Brain Sciences" 3, pág. 417-424.

ANEXO: CUADRO COMPARATIVO DE LAS DIFERENTES NOTACIONES

Como resumen final veamos un cuadro comparativo de las diferentes notaciones usadas para las sentencias, desde las fbf del cálculo de predicados hasta la sintaxis de Prolog, pasando por la notación clausal y la de la programación lógica:

Ejemplo:

CONTROL EN LA PROGRAMACIÓN LÓGICA

Control métodos de búsqueda e inferencia

Prolog es un lenguaje orientado a programar aplicaciones para la Inteligencia Artificial que requieren representar conocimiento a través de **predicados** en forma de **hechos** y **reglas,** procurando encontrar soluciones a problemas mediante el ejercicio de búsquedas **ciegas** en sentido **hacia atrás** (backward) y **a profundidad**, realizando inferencias a través del método de **Resolución**. En suma, Prolog representa una **"máquina de inferencia"** que tiene implementadas estas técnicas de almacenamiento y explotación del conocimiento para ponerlas al alcance del usuario liberándolo de la carga de implementar por sí mismo estas funciones. De esta manera el programador se puede concentrar en codificar los elementos relacionados con el problema, dejando a Prolog la búsqueda de la solución.

Prolog como lenguaje de "**Programación Lógica"**emplea una sintaxis muy sencilla para codificar las instrucciones que el usuario le proporciona mediante las convenciones del **Cálculo de Predicados** y de las **"Cláusulas de Horn",** representa los hechos y reglas conforme a los siguientes postulados:

-Un programa Prolog se compone de **cláusulas**
-Hay dos tipos de cláusulas, **hechos y reglas**
-Los **hechos** son **afirmaciones** compuestas por un predicado
-Las **reglas** son cláusulas de Horn
-Las reglas se componen por una conclusión y una condición
-La conclusión consta de un solo predicado que está afirmado
-La condición se integra por al menos un predicadoCuando la condición tienen varios predicados, generalmente estos se encuentran unidos por el conjuntivo **"and"**
-Para que sea valida una regla, es necesario que todas las condiciones se prueben como válidas (en el orden en que aparecen) para que la conclusión también así resulte y por lo tanto la regla como un todo
-Los **predicados** se componen por un nombre "reservado" o "definido por el usuario" (comenzado por una letra en minúscula o símbolo especial) y opcionalmente pueden ir acompañados por uno o más **términos** los cuales van entre paréntesis
-Los términos son variables, constantes o funciones (equivalentes a predicados), separados por comas
-Las variables utilizan como primer carácter una letra mayúscula
-Las constantes son literales que inician con minúscula, pueden ir entre comillas, son números y caracteres especiales
-Las funciones son equivalentes a los predicados, es decir llevan un nombre que comienza con una letra en minúscula y pueden ir acompañadas por términos
-Generalmente el **punto** se utiliza para indicar el término de una instrucción
-Normalmente los símbolos.**-** separan la conclusión de la(s) condición(es), la cual aparece primero (de izquierda a derecha)
-La **,** (coma) separa a las condiciones de una regla, representando el "and", mientras que el **;** representa al "or"
-Los hechos y las reglas terminan con un punto

Para efectos de ilustración de la forma en que Prolog representa conocimiento, se muestra a continuación los siguientes ejemplos, basados en la sintaxis, convenciones y elementos de la versión Prolog V-Plus desarrollada por Chalcedony Software, Inc; en donde se usa el carácter > para editar una instrucción directamente sobre el ambiente de trabajo y el símbolo % para identificar un comentario:

>!.
>fail.
>write("hola").
>padre(juan, mario).
>padre(mario, carlos).
>abuelo (X, Z) :- padre(X, Y), padre(Y, Z).

La codificación de un programa Prolog, consta de reglas y hechos esencialmente, representando en conjunto una "Base de Conocimientos" (otros la denominan Base de Datos), para procesarlas se emplea un **predicado que se inserta en forma negada.** Esta cláusula representa el **"teorema a demostrar"** o el **"problema a resolver"** conforme a lo estipulado por el método de inferencia de "**reducción a lo absurdo**" – en donde se introduce una preposición negada que se refuta contra las afirmaciones existentes, hasta generar la preposición "nula", momento en el cual se **demuestra la veracidad de la afirmación** -.

Debido a que Prolog maneja **predicados** en lugar de **preposiciones**, emplea el método de **Resolución** propuesto por Robinson - el cual aprovecha los fundamentos del **reducción a lo absurdo** -, por lo que su activación también parte de la formulación de los postulados originales (hechos y reglas) y de un predicado negado (el teorema o problema). Este último se inserta a la Base de Datos para producir nuevas cláusulas, que al trabajar con ellas generarán otras, hasta que produzcan a una vacía conocida en el argot como "**nil"**, **obteniendo así la demostración del teorema o la solución del problema.**

Cuando se desea **comprobar un teorema**, el predicado que se introduce no lleva términos o bien todos ellos deben **estar instanciados**, es decir todas sus variables tienen valores previamente asignados, ninguna de ellas se encuentra "libre" al ser introducida, como se aprecia en la siguiente reseña:

Teorema: demostrar que: "Juan es abuelo de Carlos"
La Base de Datos (conocimiento) de postulados iniciales es:

>padre (juan, mario).
>padre (mario, carlos).
>abuelo (X, Z) :- padre (X, Y), padre (Y, Z).

Se introduce el teorema negado con el símbolo ? :

>abuelo (juan, carlos)?

Se relaciona contra la regla:

>abuelo (X, Z) :- padre (X, Y), padre (Y, Z).

Se pretende unificar: abuelo(juan, carlos)? == abuelo (X, Z) :- padre (X, Y), padre (Y, Z)
Se refuta contra la conclusión (ya que está afirmada) y se propagan los valores de las variables, generando una lista de sustitución: *{ juan / X, carlos / Z }*

La cual al propagarse (sustituir los valores por las variables de la lista) en los predicados restantes de la regla, genera una nueva cláusula conocida como "Resolvente":
padre (juan, Y), padre (Y, carlos)

Ahora, se deberá probar las condiciones resultantes (recordar que están negadas), repitiendo el proceso: Al relacionarse con el hecho:

>padre (juan, mario)

Pretendiendo unificar: padre(juan, mario). == padre (juan, Y), padre (Y, carlos) La nueva lista de sustitución que se genera: *{ mario/Y, carlos/Z }*

Al propagarse a la condición restante produce como nuevo Resolvente: *padre (mario, carlos)*
Como esta es la última condición restante y también está negada, se relaciona con el hecho:

>padre (mario, carlos)

Se trata de unificar: padre (mario, carlos) == padre (mario, carlos)
Y como el hecho está afirmado y la condición es negada, entonces se produce la contradicción generando así la cláusula vacía **"nil"**

Con lo cual queda demostrado el teorema que declara: *"Juan es abuelo de Carlos"*

Si en cambio lo que se desea es **resolver un problema,** este se plantea como un predicado que se introduce con cuando menos un término **no instanciado**, es decir, debe tener al menos una variable sin valor previamente asignados, es decir que está "libre" al ser introducida. El mecanismo de inferencia que se aplica es el mismo que para la demostración de teoremas por lo que solo se muestra a continuación el proceso aplicado a las cláusulas:
Problema: "¿Quién es el nieto de Juan"
Se emplea la misma Base de Datos

>padre (juan, mario).
>padre (mario, carlos).
>abuelo (X,Z):- padre (X, Y), padre (Y, Z).

Se introduce el problema negado con :

>abuelo (juan, K)?

Al relacionarse con la regla: abuelo(juan, K)? == abuelo (X, Z) :- padre (X, Y), padre (Y, Z)
Genera la lista de sustitución: *{ juan / X, K / Z }*
Emitiendo el "Resolvente": *padre(juan, Y), padre(Y, K)*
Al relacionarse con el hecho: padre (juan, mario). == padre (juan, Y), padre (Y, K)
Genera una nueva lista de sustitución: *{ mario/Y }*
Emitiendo el "Resolvente": *padre (mario, K)*
La condición restante se relaciona con el hecho: *padre (mario, carlos) == padre (mario, K)*
Genera una nueva lista de sustitución: *{ carlos / K }*
Se produce la contradicción. **"nil"**
Al **propagar el valor de K** hasta el postulado introducido: *abuelo(juan, carlos)?*
Se obtiene la solución al problema: *"Carlos es el nieto de Juan"*

Cuando en una demostración de teorema o solución de problema no se llega a generar la cláusula vacía **nil** entonces no se puede afirmar el postulado o bien no existe solución, como se muestra en la siguiente figura 1.1:

Demostrar que "Juan es abuelo de carlos"

abuelo (juan, carlos)

1 **3** **5**

padre(juan, carlos) **padre(juan, carlos)** **abuelo (X, Z) :- padre(X, Y), padre(Y, Z)**

2 **4** **6**

XX **XX** **XX**

No es demostrable que *"Juan es abuelo de carlos",* ya que no se generó la cláusula NIL

Figura. 1.1 Imposibilidad de demostración

Para que dos cláusulas se puedan relacionar, es necesario que exista la **"Unificación"** entre un predicado de cada una de ellas. Para ello se aplica el método denominado **"Unificación"**, el cual exige la observancia de una serie de criterios para permitir una inferencia – es decir, la generación de un nuevo conocimiento a partir de otro – representada por la **"lista de sustitución"** y el **"Resolvente".** Los requisitos que dos cláusulas deben cubrir para ser unificables son:

-Los predicados deben tener el mismo nombre
-Un predicado debe estar afirmado y el otro negado
-Deben tener la misma cantidad de términos
-Los términos de los predicados deberán ser correspondientes, es decir, se comparan entre sí de acuerdo con la posición en la que aparecen (el uno contra el primero, el dos contra el segundo, ...)
-Si los dos son constantes, ambas deben ser las mismas
-Si son variables instanciadas, sus valores asignados deberán ser los mismos
-Si un término es una variable libre y el otro es una constante o función, el primero tomará el valor del segundo
-Si un término es una variable libre y el otro es una variable instanciada, la primera tomará el valor de la segunda
-Si un término es una variable libre y el otro también, la primera sustituye a las ocurrencias de la segunda
-Si un término es una función el otro deberá ser una variable libre o bien otra función con el mismo nombre, número de términos y estos deberán ser correspondientes como se marca para los predicados (es recursivo)

Cuando todos estos criterios son observados con éxito, por las predicados contendientes de las cláusulas que se pretende relacionar, entonces existirá **Unificación** entre ellas, procediendo a generar la **lista de sustitución** a partir de la asignación de valores que se deriva de los criterios 7, 8 y 9. Estas sustituciones son propagadas en el resto de predicados de las dos cláusulas que se van a relacionar, procediendo a eliminar a los predicados que se unificaron (por haber contradicción, ya que uno está afirmado y el otro negado) para generar la cláusula resolvente con la unión de los predicados restantes de las dos partes, los cuales ya tienen los nuevos valores resultantes de las dos sustituciones.

Al incumplirse cualquiera de los criterios se impide que las dos cláusulas se relacionen como ocurrió en el ejemplo de la figura 1.1, en donde por usar distintas constantes (en los primeros dos intentos) y no poder unificar un mismo predicado afirmado contra otro negado (tercer caso) no fue posible demostrar el teorema.

Por lo que respecta a la búsqueda que Prolog automáticamente emplea, esta es **ciega,** ya que no se emplea algún criterio en particular, información o conocimiento especializado – conocido como **heurística** – para orientar la dirección de la búsqueda, la selección de las declaraciones (hechos y reglas); limitándose únicamente a tratar de relacionar las cláusulas **con base al orden** en que aparecen, tal como se ilustró en la figura anterior, al observar los números del proceso se aprecia su correspondencia con respecto a la secuencia de los postulados en la Base de Datos previamente declaradas; es decir, que primero se compara el teorema negado contra los hechos y al final contra la regla.

En cuanto a la **dirección de la búsqueda** que Prolog utiliza esta es **backward** ya que el procedimiento arranca a partir del teorema (conclusión) que supuestamente se deriva de los postulados iniciales; lo cual implica que a partir de estos (la Base de Datos) se deduce el teorema (dirección **forward**). Por ese motivo, el teorema o problema a resolver se introduce negado para procurar relacionarlo con cualesquiera de las declaraciones iniciales para generar un nuevo resolvente, y con él repetir el proceso hasta producir la cláusula vacía **nil** y obtener la respuesta deseada o bien agotar todas las posibilidades sin satisfacer la expectativa inicial.

Con respecto a la expansión de la búsqueda que Prolog realiza, se emplea el método a **Profundidad** en donde se privilegia desarrollar la primer alternativa en cada nivel de búsqueda hasta llegar al final, alcanzar la solución o bien llegar a un punto donde no se puede progresar más, con lo cual se obliga a dar un paso atrás denominado **backtrack** en donde se regresa al estado en el que se tomó la última decisión para considerar otra alternativa que no se habían contemplado antes por ser la segunda, tercera, ... opción. En caso de que en ese nivel no existan más o bien que todas ellas ya fueron exploradas sin haber resultado exitosas, entonces se hace un nuevo retroceso al siguiente nivel inmediato anterior, para procurar otras rutas. Este proceso se repite hasta encontrar el primer caso de éxito o bien agotar todo el posible universo de búsqueda, lo cual ocurre cuando al realizar el backtrack se arriba al estado inicial (primer nivel) y ya no hay más caminos por explorar.

Prolog se orienta a realizar **búsquedas exhaustivas**, en donde no conforme con haber encontrado la primer solución, continua buscando más, aplicando el mecanismo de **backtrack** hasta cubrir el universo completo de búsqueda o agotar los recursos de proceso disponibles. Con este lineamiento, se pueden generar todas las posibles soluciones de un problema, elegir entre ellas o bien tener la certidumbre de que no existe una solución satisfactoria o que es necesario ampliar los recursos o replantar el problema para hacer viable encontrar una respuesta a satisfacción. En contraparte, si el usuario solo se interesa en conocer la primer solución, habrá un desgaste de recursos durante la ejecución de la aplicación, el cual se podrá evitar con el empleo del predicado ! (cut).

Recursividad normal y tipo de cola (TAIL)

En computación **un proceso recursivo es aquel que se llama así mismo**, generando una nueva "versión" de las variables de estado del programa independientes al que tienen al momento de que ocurre el llamado.

Este proceso es cíclico y **debe estar controlado por una condición** que limite el término de las llamadas de lo contrario se vuelve infinito. Cuando se alcanza esa condición que da por terminado el ciclo (después de "n" interacciones), se genera un retroceso en el que puede o debe (según el caso) ocurrir un **"envío de parámetros hacia arriba"** para trasladar los valores de interés que generó el último ciclo al inmediato anterior ("n-1", el penúltimo)– es decir, al ciclo que lo llamó –.

Para que este los procese y genere con ellos un nuevo resultado, o bien simplemente los pase al ciclo inmediato anterior que le corresponde ("n-2" el antepenúltimo). Este proceso se hace de manera sucesiva hasta regresar al estado original del programa (nivel 0) en el punto en que ocurrió la llamada, para que la aplicación continúe a partir de la siguiente instrucción a la que hizo el llamado.

Al iniciar el ciclo recursivo se pueden enviar ciertos parámetros con valores previamente definidos y otros libres sin instanciar, cuando se ingresa al primer ciclo recursivo este los recibe, procesa y procede a generar nuevos valores, los cuales pueden "perderse", grabarse en el algún dispositivo, enviarse al siguiente ciclo recursivo o bien en el **"envío de parámetros hacia arriba"**, trasmitirse al nivel que los llamó para asignarse a las variables libres originales. De esta manera ocurre el **"paso de parámetros hacia abajo"** (en cada llamada recursiva).

En Prolog la recursividad ocurre cuando un predicado aparece como la conclusión de una regla y también como una de las condiciones de la misma, es decir que la regla se define así misma en términos de ella. A nivel de implementación de la recursividad, Prolog emplea estructuras de datos tipo "pila" para registrar, almacenar y recuperar los valores de las variables de estado para cada ciclo.

La **recursividad normal** es precisamente la que se ha definido hasta el momento, enfatizando el hecho que una vez concluida, al volver al estado inicial el resultado final que se obtuvo es recuperado para ser empleado a continuación por otro predicado que está después del que hizo el llamado en el mismo "cuerpo" de condiciones de la regla. Como ejemplo de aparece el cálculo del factorial el cual es implementado por medio de las siguientes declaraciones:

```
>% Programa 1
>% Recursividad Normal
>factorial (N, 1) :- 1 >= N.                          % cláusula 1
>factorial (W, X) :- W > 1, Y is W - 1, factorial (Y, Z), X is W * Z.    % cláusula 2
```

En el programa 1, se aprecian comentarios en las dos primeras líneas, mientras que la tercera es una regla cuya conclusión se representa por medio del predicado "factorial" con dos términos, mientras que la conclusión dispone de un solo predicado. La última instrucción también es una regla con el mismo predicado como conclusión y con cuatro predicados en la parte de condición, en donde el tercero de ellos también es el predicado "factorial", representando con ello una regla recursiva, cuya condición terminal ocurrirá cuando sea satisfecha la primera.

El programa se activa digitando la instrucción que represente al factorial a calcular, como por ejemplo, si se desea estimar el factorial del número 3, se introduce la cláusula negada:

```
>factorial(3, X)?
```

Con ella se plantea el problema, negando el predicado **factorial** con el símbolo **?** declarando como primer término el número **3**, y la variable libre **X** como medio para obtener el resultado.

El **"envío de parámetros hacia abajo"** se realiza a través de los dos términos del predicado factor, en cada llamado recursivo el valor del primer de ellos – el cual representa el número cuyo factorial ha calcularse - se altera debido al decremento que calcula la segunda condición de la segunda regla. Mientras que el segundo término permanece como una variable libre. Cuando en la primer regla el valor del primer término de la conclusión es **1**, entonces se cumple su única condición, poniendo fina a las llamadas.

Una vez satisfecha la condición que controla la recursividad, la conclusión de la regla 1, devuelve como valor de sustitución para el segundo término del predicado "factorial" el número **1** el cual instanciará a la variable libre del segundo término del predicado que hizo la llamada correspondiente a la tercer condición de la segunda regla, con ello se obtiene el valor final del último nivel recursivo para dar comienzo al **"paso de parámetros hacia arriba"**. Con 1 como primer valor retornado al penúltimo nivel recursivo.

Este procede a ejecutar el predicado que sigue al responsable de la llamada recursiva ubicado en la segunda regla, dicha condición se encarga de calcular el factorial de un número (**n!**) al multiplicarlo por el factorial del número inmediato inferior (**n * (n-1)!**). El producto obtenido es propagado a la variable libre del segundo término del predicado de la conclusión de la segunda regla, para que se retorne al nivel en que se hizo el llamado recursivo. Este procedimiento se repite tantas veces como ciclos recursivos se hayan realizado hasta alcanzar el estado inicial y reanudar la validación de la regla que provocó la recursividad, la cual calculará el valor final del factorial para ser devuelto al predicado que planteó el problema a través de la conclusión de la segunda regla.

En la siguiente tabla (1.1) se muestra la cronología de la transferencia de valores hacia abajo, cálculo, envío de valores hacia arriba, generación de valores finales y el resultado de la unificación con fracaso "fail" o éxito "true". Para facilitar la interpretación se abrevió al predicado "factorial" por "f", las celdas que representan a los predicados que no se han procesado hasta ese momento por haberse presentado el llamado "recursivo" aparecen con un fondo oscuro, mientras que las ya procesadas al retorno recursivo se identifican con una asurado en diagonal. En las columnas se muestran los valores de los términos o bien las variables libres para los predicados de la conclusión y de las condiciones de las dos reglas. En los renglones se muestra los valores que toman los predicados del problema, de la regla 1 y de la 2, según el orden de operación que se realiza. Mientras que la columna "nivel" señala el estado de ejecución del programa (0 el normal, mayor a 0 constituye el nivel de recursividad) y la de "resultado" indica el fracaso, éxito, llamado ("lla. rec.") o retorno recursivo ("return") de la operación.

Nivel	**Problema**	**Regla 1 conclusión**	**Regla 1 condición 1**	**Regla 2 conclusión**	**Regla 2 condición 1**	**Regla 2 condición 2**	**Regla 2 condición 3**	**Regla 2 condición 4**	**Resultado**
0	f(3,X)								
0		f(3,1)	1 >= 3						fail
0				f(3, X)	3 > 1	Y = 3 - 1	F(2, Z)		lla. rec. 1
1		f(2,1)	1 >= 2						fail
1				f(2, X)	2 > 1	Y = 2 - 1	F(1, Z)		lla. rec. 2
2		f(1,1)	1 >= 1						true, return
1							F(1, 1)	X = 2 * 1	return
0							F(2, 2)	X = 3 * 2	true
0	f(3,3)								fin

Tabla. 1.1 Recursividad normal

La **recursividad tail** representa una variante de la "normal", procurando que el valor que se calculó en el último nivel de recursividad alcanzado ya no se altere más durante el "retorno de valores hacia arriba" y una vez que se ha vuelto al nivel original donde ocurrió la primer llamada, el predicado responsable sea el último de la condición de la regla, es decir la **"cola**", o bien que en caso de haber otros predicados después, estos no utilicen en lo absoluto ninguno de los valores retornados por el llamado recursivo. Para ilustrar este concepto se procede a modificar el programa original 1 para dar vida a la versión 2:

```
>% Programa 2
>% Recursividad Normal
>factorial (N, 1) :- N =< 1.                                    % cláusula 1
>factorial (2, 2).                                              % cláusula 2
>factorial (N, V) :- N > 2, fa (N, 3, 6, V).                    % cláusula 3
>fa (L, L, Va, Va).                                             % cláusula 4
>fa (L, Na, Va, Vf) :- L > Na, Nn is Na + 1, Vn is Nn * Va, fa (L, Nn, Vn, Vf).% cláusula 5
>factorial (4, X)?                                              % problema
```

Esta nueva versión computa el factorial en dirección "forward", es decir en sentido opuesto al programa original, ya que parte de los valores iniciales 1, 2, 3, ... y así sucesivamente hasta alcanzar el número solicitado; su activación emplea al mismo predicado acompañado por el valor del número del factorial a calcular y la variable libre donde se almacenará el resultado, tal como aparece en la última instrucción ("problema").

La cláusula 1 se emplea para el cálculo del factorial de números menores o iguales a **1**, la segunda ofrece el valor del factorial para el número **2**, la tercera se ocupa de generar el factorial para números mayores a **2**, invocando al predicado "fa" por medio de cuatro términos. El primero corresponde al número "original" cuyo factorial se desea obtener (variable instanciada "N"), el segundo constituye el número "actual" (valor 3) a partir del cual comenzará la estimación, acompañado por su factorial (valor 6) como tercer término, mientras que como cuarto aparece la variable libre donde se depositará el factorial producido para el número "original".

La cláusula 4 representa un "hecho" con el propósito de establecer la condición terminal de la recursión al momento en el que el valor del primer término es igual al segundo (es decir, que el número "original" es igual al número "actual") por lo que el factorial del número "actual" –representado por el tercer término- instanciará a la variable libre del cuarto término, devolviendo este como el resultado final del ciclo recursivo.

La última cláusula es una regla recursiva y procede a calcular el factorial del número "actual" inmediato posterior al obtener el producto de este por el factorial del número "actual", para después hacer un llamado recursivo con el número "actual" incrementado en 1, acompañado por su factorial recién calculado. Este ciclo se repite hasta satisfacer la condición terminal expresada por el "hecho"; sin embargo, para evitar que el "backtrack" impulsado por el espíritu de búsqueda exhaustiva característico en Prolog, continué procurando encontrar más soluciones es necesario emplear la primer condición de la última regla que evita procesar el factorial de números mayores al "original".

Aplicando la misma mecánica del ejercicio anterior y las convenciones adoptadas en la representación de la tabla 1.1; a continuación se muestra la tabla 1.2 para la "ejecución" del segundo programa, con la salvedad de dedicar una columna para la conclusión de las reglas de las cláusulas 3 y 5, etiquetándolas con el nombre del predicado.

Nivel	**Problema**	**Cláusula 1 / 2**	**Cláusula 3 factorial**	**Cláusula 3 condición**	**Cláusula 4**	**Cláusula 5 fa**	**Cláusula 5 condición 1, 2, 3**	**Cláusula 5 condición 5 (fac)**	**Resultado**
0	f(4,X)								
0		fail							fail
0		fail							fail
0			f(4, V)	4 > 2 fa(4,3,6,					lla. rec. 1
1					fa(4,3,6,6				fail
1						fa (4,3,6,Vf)	4 > 3 4=3+6	fa (4,4, 24,	lla. rec. 2
2					fa (4,4, 24,24)				true, return
1						fa (4,3,6,24)		fa (4,4, 24, 24)	true, return
0			f(4, 24)	4 > 2 fa (4,3, 6,24)					true
0	f (4,24)								fin

Tabla. 1.2 Recursividad tail

El principal atributo de la recursividad "tail" es evitar el "postproceso" sobre los valores calculados en cada ciclo recursivo, haciendo ágil el retorno de valores hasta el llamado original y menos complicado el rastreo y depuración del programa. En suma el modelo de recursividad "tail" es equivalente a las iteraciones clásicas carentes de recursividad puesto que esencialmente no requieren del empleo del mecanismo y sobrecarga de trabajo tradicional que exige la recursividad normal.

BackTrak uso del corte (CUT) y de falla (FAIL)
Por "default" Prolog se encarga de realizar una búsqueda exhaustiva de soluciones a lo largo (profundidad) y ancho (amplitud) del universo de búsqueda que se genera para el problema. Esto significa que al encontrar la primera solución activa su mecanismo de backtrack (retroceso) para optar por otras rutas aún no exploradas en búsqueda de nuevas soluciones.

Cuando la solución a un problema exige conocer todas las posibles respuestas, el retroceso natural de Prolog las produce en forma automática siempre y cuando los recursos de proceso y memoria se lo permitan. Sin embargo no siempre el usuario se interesa por enterarse de todas las posibles opciones y se conforma con la primera o con algún conjunto de ellas sin necesidad de ser exhaustivo; o bien se enfrenta a problemas que tienen una sola posible solución.

Al carecer de una heurística que facilite encontrar soluciones o bien que las evalué para recomendar la mejor conforme a ciertos criterios específicos, Prolog simplemente expondrá las respuestas en el orden "casual" (basado en el estilo de programación y secuencia de las cláusulas) conforme las vaya encontrando (si es que existe alguna o la capacidad de recursos se lo permite).

Al momento en que el diseñador de la aplicación decide ofrecer la primer respuesta que satisfaga al problema o una parte de este (representado a través de hechos o condiciones de reglas), el programador entonces podrá emplear el predicado **"cut"** mediante el símbolo **!** para indicarle a Prolog evitar la activación del backtrack al momento que regrese a buscar nuevas opciones de satisfacción para las condiciones que se encuentran antes de él en la misma regla y para el predicado de la conclusión a la que pertenece.

Para comprender mejor este concepto, es conveniente revisar el primer programa presentado en la sección anterior, partiendo del ejemplo que se desea calcular el *factorial de 1, 0 o de cualquier número negativo*. La respuesta la proporciona Prolog al momento de unificar el problema negado *factorial(1, X)?* con la primer cláusula, sin embargo una vez que presenta el resultado *x = 1* el programa activa el backtrack para continuar buscando más soluciones y encuentra como una nueva ruta a la segunda regla, logrando unificar al predicado de su conclusión, pero fallando en satisfacer a su primer condición *(w > 1).* Como ya no hay mas opciones de búsqueda concluye el proceso.

Para evitar este sobretrabajo "overhead" innecesario, bastará con modificar el programa agregando una condición más a la primer regla, representada por el predicado "cut" mediante su símbolo !. Como esta regla es la condición que controla la recursividad, una vez que satisfaga la primer condición *(1 >= N)* ya no habrá más retroceso para satisfacer al predicado (*factorial)* tanto a través de esa regla como de cualquier otra más que exista en el código (en este caso únicamente hay 2). Con estos elementos de edita el programa 3 que se comparte a continuación:

```
>% Programa 3
>% Recursividad Normal
>% Empleo del cut   !

>factorial (N, 1) :- 1 >= N,  !.                          % cláusula 1
>factorial (W, X) :- W > 1, Y is W - 1, factorial (Y, Z), X is W * Z.      % cláusula 2
```

Al aplicar el predicado "cut" al segundo programa de recursividad "tail", se agrega el símbolo **!** como última condición de la primer regla; la segunda cláusula se transforma de "hecho" a "regla" al insertar como única condición el **!**; y de igual forma la cuarta también es convertida en regla para emitir la nueva versión del programa denominada 4, la cual consta de las siguientes instrucciones:

```
>% Programa 4
>% Recursividad Tail
>% Empleo del cut   !

>factorial (N, 1) :-  N =< 1,  !.                          % cláusula 1
>factorial (2, 2):-   !.                                   % cláusula 2
>factorial (N, V) :- N > 2, fa (N, 3, 6, V).               % cláusula 3
>fa (L, L, Va, Va):-   !.                                  % cláusula 4
>fa (L, Na, Va, Vf) :-  L > Na, Nn is Na + 1, Vn is Nn * Va,  fa (L, Nn, Vn, Vf).% cláusula 5
```

Con estos ajustes el **cut** evita al backtrack de la siguiente forma:
Cuando el problema consiste en calcular el *factorial de un número igual o menor a 1*, el **!** de la primer regla impide que el bactrack intente explorar nuevas soluciones a través de las cláusulas restantes

Si el objetivo es obtener el *factorial de 2*, el **!** de la segunda regla evita el retroceso que pretende buscar mas soluciones por medio de las cláusulas faltantes

Y para el resto de casos *(factorial de números igual o mayor a 3),* el **!** es el responsable de evitar mas overhead

Como recomendación final al usuario, se observa el cuidado que debe ejercer en el manejo de búsqueda exhaustiva o bien limitada, ya sea por el estilo de programación natural de Prolog –aquel que no emplea el **cut** – o bien el contrastado por el que explota el **!**, el cual puede generar resultados inesperados e inclusive impedir el descubrimiento de posibles soluciones por un apropiado manejo, algo que provoca graves problemas al momento de depurar la aplicación.

Debido a que las cláusulas que normalmente integran un programa Prolog se componen por "hechos" y "reglas" y la única negación es la cláusula que se introduce para representar el teorema o problema a resolver - propiamente dicho, no hay más cláusulas negadas en una aplicación – el programador se encuentra ocasionalmente con la necesidad de implementar un **"hecho negado"** es decir una aseveración de la falsedad de una declaración, como por ejemplo: "maría *es padre de pedro"* o inclusive el negar la veracidad de la conclusión resultante de una regla tal que diga: *"si X es mujer entonces X es padre de Y".*

Para representar estas situaciones Prolog ofrece el predicado **fail** el cual una vez satisfecho provocará el efecto inverso, es decir negará lo que se está afirmando; como por ejemplo para negar el hecho:

>padre (maría, pedr0). ***% cláusula 1***

La cláusula se convierte en regla y se agrega como única condición **fail**, el cual una vez probado producirá el rechazo de la afirmación, por lo que el proceso aplicará el backtrack para descubrir otra solución.

>padre (maría, pedr0):- fail. ***% cláusula 1***

De la misma forma se procede a modificar la regla que se preocupa por validar el género del término que caracteriza la paternidad y cuya conclusión debe ser rechazada al adicionar el **fail** como segundo predicado de la condición.

>padre (X, Y):- mujer(X), fail. ***% cláusula 1***

Si esta regla la probamos junto con el hecho de que *maría es mujer* y el problema es determinar la paternidad de *pedro*, tenemos la siguiente secuencia de instrucciones que conforman al programa 5:

>% Programa 5
>% Ejemplo del empleo del "fail"

> mujer(maria). ***% cláusula 1***
>padre (X, Y):- mujer(X), fail. ***% cláusula 2***

Al plantea el problema *"maría es padre de pedro ?"* mediante el hecho negado:

> padre(maria, pedro)? ***% problema a resolver***

Prolog rechazará cualquier intento de instanciar a *maría* como *padre* de alguien por la regla cuya primer condición valida su genero.

Finalmente, volviendo a la naturaleza exhaustiva de Prolog, en el caso anterior buscará otras soluciones al problema de "*paternidad en una mujer"* pero esto es un absurdo, por lo que si la base de datos de la aplicación es muy voluminosa entonces habrá un gran overhead gastado inútilmente en algo que no tiene solución por ser en este caso inverosímil. Por tal motivo se deber emplear la combinación **cut** and **fail** (es decir secuenciar los predicados **!, fail)** en el cuerpo de la condición para que de esta forma se cancele el intento de backtrack. Al aplicar este concepto al programa, se inserta en la segunda regla el **cut** antes del **fail** para generar una nueva versión representada por el programa 6:

>% Programa 6
>% Ejemplo del empleo del "cut - fail" para el manejo de absurdos

> mujer(maria). ***% cláusula 1***
>padre (X, Y):- mujer(X), !, fail. ***% cláusula 2***

Esta combinación es muy útil en aplicaciones complejas del campo de la Inteligencia Artificial en donde se puede intentar prever la exploración de rutas que representan "absurdos" o bien en problemas que no son solubles, y que al identificar estos casos se evita un desgaste inútil de recursos de procesamiento, las cuales se denominan "escépticos".

Aplicaciones: factorial, creación y búsqueda en listas, etc.

Durante el proceso de aprendizaje del lenguaje de Programación Lógica "Prolog" es una práctica común el **implementar el cálculo del factorial** como una forma de entender el manejo de la recursividad a través del empleo de cláusulas en forma de hechos y reglas. Como ya se ha vista anteriormente a lo largo de las secciones 1.2 y 1.3, existen variantes en su diseño, por lo que en este apartado se expresan algunos aspectos destacados.

En primer lugar por lo que respecta a la representación de conocimiento, todos los programas usan bases de datos **compuestas** por **reglas,** salvo el segundo que emplea un par de **hechos**
Todos emplean mecanismos de recursividad
Aplican búsqueda ciega a profundidad
La dirección de búsqueda es hacia **atrás** para todos los casos

Sin embargo el cálculo del factorial en los programas 3 y 4 es hacia delante – a partir del número 1, 2, 3 .... y así sucesivamente hasta alcanzar el número deseado -. Mientras que en el 1 y 2, la orientación es hacia atrás, debido a que comienza decrementando el número deseado hasta llegar al 1, y a partir de él se computa el factorial respectivo durante los retornos de cada nivel recursivo hasta llegar al original donde se genera el valor del factorial solicitado

Existen condiciones que evitan el overhead al impedir el calculo de factoriales menores a 1 – como sucede con la condición (*1 >= N)* de la primer regla del programa # 1- o la estimación del factorial de números superiores al requerido – tal como se encarga la primer condición (*L > Na)* de la quinta cláusula del cuarto programa –

En suma, los cuatro ejercicios ofrecen conceptos adicionales al proceso del factorial como es el empleo de los tipos de recursividad, el uso del cut y el manejo del backtrack; los cuales son indispensables aplicar en programas más complejos

Por lo que respecta al manejo de listas, en primer lugar es conveniente considerar su naturaleza. **Una lista es una estructura de datos lineal,** compuesta por varios, uno o ningún elemento; los cuales pueden ser otras listas o átomos (constantes y variables) y que únicamente puede ser accesada a partir del primero.

Como por ejemplo, para representar una lista de colores en Prolog, se digita la siguiente instrucción:

```
>colores ( [rojo, azul, verde] ).        %  cláusula 1
>colores ( [ ] ).                        %  cláusula 2
```

En la primer cláusula se aprecia que el único término del predicado *colores* es una *lista* compuesta por 3 elementos separados por comas, los cuales aparecen entre los corchetes [ ]. Mientras que la segunda cláusula representa un hecho cuyo predicado consta de un término caracterizado por una lista **vacía** es decir **nula.**

Para acceder a los elementos de una lista existen varias opciones, la mas elemental es mediante el establecimiento de una correspondencia entre dos predicados a unificar, en donde uno de ellos contiene como término a una lista con "n" número de elementos y el otro emplea una variable libre que será sustituida por la lista completa o bien por un elemento específico de ella como se muestra en el programa 7, donde a partir del hecho "colores" que contiene una lista con 3 elementos (atómicos) se plantean dos problemas; el primero dedicado a recuperar la lista completa en una sola variable (X) y el otro que emplea tres variables contenidas dentro de los corchetes [A, B, C] con la intención de obtener el valor individual de cada uno de ellos:

```
>% Programa 7
>% Ejemplo de acceso a listas

>colores ( [rojo, azul, verde] ).     %  cláusula 1
>colores ( X )?                       %  problema 1: acceso a toda la lista
 X = [rojo, azul, verde]              %  solución al problema 1

>colores ( [A, B, C] )?               % problema 2: acceso a cada elemento de la lista
 A = rojo
B = azul
C = verde                             %  solución al problema 2
```

Con respecto al ejemplo anterior las líneas que no comienzan con el símbolo > corresponden a la respuesta que Prolog genera al problema que previamente se le plantea mediante la cláusula negada (predicado (términos) ?). Ahora bien, si el usuario solo se interesa en acceder a un elemento específico bastará con reconocer la posición que ocupa para que le corresponda una variable que será sustituida al momento de la unificación.

Mientras que a los elementos que no son de interés, se podrá usar una variable distinta para cada uno de ellos (como en el ejemplo anterior) o bien se podrá emplear la **variable anónima** representada por el símbolo _ en el lugar correspondiente a los términos que no importa recuperar, tal como se muestra en el programa 8:

```
>% Programa 8
>% Ejemplo de acceso a listas

>colores ( [rojo, azul, verde] ).     %  cláusula 1
>colores ( [ D, _, _ )?               %  problema 1 : acceso al p  1er. elemento
D  =  rojo                            %  solución al  problema 1

>colores ( [ _, E, _ )?               %  problema 2 : acceso al 2do. elemento
E = azul                              % solución al  problema 2
>colores ( [F,_, G )?                 %  problema 3 : acceso al 1er y 3. elementos
F =  rojo                             % solución al  problema 3
G =  verde
```

Por lo que respecta al manejo tradicional de las listas y conforme al estilo que el lenguaje de "Proceso de Listas" LISP realiza, la **búsqueda de elementos** se hace únicamente a través del primer elemento de ellas conocido como "c**abeza"** ("CAR" en el ambiente LISP) dejando el resto, es decir la **"cola"** (denominado "CDR" en LISP) sin acceder. Para representar y ejemplificar el funcionamiento de las dos funciones "primitivas" de LISP en Prolog se ofrece la siguiente sección de código del programa 9:

```
>% Programa 9
>% Implementación del "CAR" y "DR"
>colores ( [rojo, azul, verde] ).      % cláusula 1
>colores ( [ A | B ] )?                % problema 1 : acceso al CAR y CDR de la lista
A = rojo                               % solución al problema 1: valor CAR B = [azul, verde]
                                       % solución al problema 1: valor CDR

>colores ( [ C |_ ] )?                 % problema 2 : acceso al CAR
C = rojo                               % solución al problema 2: valor CAR

>colores ( [ _ | D ] )?                % problema 3 : acceso al CDR
D =[ azul, verde]                      % solución al problema 3: valor CDR
```

En este ejemplo se aprecia el uso del símbolo | para separar la "cabeza" de la "cola", además que cuando se accede al CAR, Prolog devuelve únicamente al primer elemento (ya se lista o átomo), mientras que tratándose del CDR se obtiene como respuesta un **lista** con los elementos restantes después del primero. Ahora bien, cuando se trata de acceder a los elementos de una lista nula Prolog responde negativamente **"no",** pero al tratar de acceder al CDR de una lista que ya no tiene más elementos mas que el CAR, entonces devuelve una respuesta afirmativa con el valor de una **lista vacía**, tal y como se muestra en el programa 10:

```
>% Programa 10
>% Acceso a CDR y lista vacía

>colores ( [rojo, azul, verde] ).   % cláusula 1
>colores ( [ A, _, B | D ] )?      % problema 1 : acceso al CAR, al 3er elemento y al CDR restante
A = rojo                           % solución al problema 1: valor CAR
B = verde                          % solución al problema 1: valor del 3er elemento
D = [ ]                            % solución al problema 1: valor del CDR

>nombres ( [ ] ).                  % cláusula 2
>colores ( [ X | Y ] )?            % problema 2 : acceso al CAR y al CDR NO
                                   % no hay solución al problema 2
```

Por lo que respecta a la **creación de listas**, desde el enfoque de LISP hay tres funciones primitivas que se pueden implementar en su equivalente Prolog tal como se muestra en el programa 11, las cuales se denominan:

CONS.- agrega un elemento a la cabeza de una lista, es decir en la posición
CAR LIST.- crea una lista a partir de dos elementos
APPEND.- genera una nueva lista a partir de la fusión de dos listas

```
>% Programa 11
>% Implementación de  "CONS", "LIST" y "APPEND"
>cons (A, B, [A | B ] ) .          %  cláusula 1
>cons (1,  [2, 3], C )?           %  problema 1: CONS, inserta un elemento en la cabeza de una lista
C = [1, 2, 3]                     % solución al  problema 1: elemento insertado en el CAR de la lista

>cons ([1],  [2, 3], C )?         %  problema 2: CONS, inserta una lista en la cabeza de una lista
C = [[1], 2, 3]                   % solución al  problema 2: lista insertada en el CAR de la lista

>list (D, E, [D, E ] ) .          %  cláusula 2
>list ([1],  [2, 3], F )?         %  problema 2: LIST, crea una lista con dos listas
F = [[1], [2, 3]]                 % solución al  problema 2: devuelve una lista con 2 listas como
elementos

>list (1, [2, 3], G )?            %  problema 3: LIST, crea una lista con un átomo y una lista
G = [1, [2, 3]]                   % solución al  problema 3: devuelve una lista con un átomo y una
lista

>list ([1], 2, H )?               %  problema 4: LIST, crea una lista con una lista y un átomo
H = [[1], 2]                      % solución al  problema 4: devuelve una lista con una lista y un
átomo

>list (1, 2, J )?                 %  problema 5: LIST, crea una lista con dos átomos
J = [1, 2]                        % solución al  problema 5: devuelve una lista con dos átomos

>append ([ ], L, L) .                                          %  cláusula 3
>append ([ Car  | Cdr], L2, [Car |  L3]):- append(Cdr, L2, L3) .      %  cláusula 4
>append ([1],  [2, 3], K )?       %  problema 6: APPEND, crea una lista a partir de listas
K = [1, 2, 3]      % solución al  problema 6: devuelve una lista con los elementos de las listas
originales
>append (1,  [2, 3], M )?         %  problema 7: APPEND, crea una lista a partir de un átomo y una
lista
NO                                % no hay solución al  problema 7: pues el 1er. Elemento no es una
lista
>append ([ ],  [ ], N )?          %  problema 8: APPEND, crea una lista a partir de 2 listas vacías
N = [ ]                           % solución al  problema 8: devuelve una lista vacía
```

Finalmente, entre las funciones clásicas de LISP en la manipulación de listas se encuentra "member" avocada a determinar si un elemento es miembro o no de una lista, como se muestra en el programa 12:

```
>% Programa 12
>% Implementación de  "Member"
>member (_, [ ] ) :-!, fail.                %  cláusula 1: No se encontró el elemento
>member (E, [ E  | _]):- !.                 %  cláusula 2: Si se encontró
>member (E, [ _  | Cdr ] ):- member(E, Cdr) .  %  cláusula 3: regla recursiva de búsqueda

>member (1,  [2, 1] )?            %  problema 1: búsqueda de un elemento que SI es miembro de una
lista
YES                               % solución al  problema 1: si es miembro
>member (5,  [2, 1] )?            %  problema 2: búsqueda de un elemento que NO es miembro de una
lista
NO                                % no hay solución al  problema 2: ya que NO es miembro
```

En el ámbito de la Inteligencia Artificial el uso de listas es ampliamente aprovechado para efectos de representación de conocimiento y procesamiento simbólico, por lo que se sugiere profundizar en su comprensión y aplicación a efecto de estar en condición de aprovecha mejor la literatura, programas y aplicaciones que forman parte del acervo.

Predicados: declarativos, aritméticos, de comparación, entrada y salida
Prolog como cualquier lenguaje de programación dispone de un repertorio de **palabras reservadas** para fines específicos de proceso y declaración de elementos; en el caso particular, corresponde al conjunto de **predicados reservados** representados por palabras o símbolos especiales encargados de probar una condición particular o ejecutar una tarea específica, los cuales pudieran acompañarse de términos obligatorios u opcionales.

En lo que respecta a los **predicados declarativos**, prácticamente son mínimos y se orientan a expresar directivas a la máquina de inferencia en cuanto a controlar la búsqueda de soluciones, el manejo de predicados y la terminación del proceso de una aplicación, tal y como se describen e ilustran en el programa 13 presentado a continuación:

Solveone.- instruye a Prolog para que una vez encontrada la primer solución al problema planteado, concluya la exploración de adicionales
Solveall.- anima a buscar todas las posibles soluciones

```
>% Programa 13
>% Ilustración de predicados declarativos
>color(rojo).          %  cláusula 1:
>color(verde).         %  cláusula 2:
>color(azul).          %  cláusula 3:

>color(X)?             % problema 1: busca todas las posibles soluciones
X =  rojo              % solución al  problema 1: ofrece todas las soluciones
X =  verde
X =  azul

>solveone?.            %  cláusula 4: se introduce una directiva a Prolog
YES
>color(X)?             % problema 2: busca la primer respuesta
X =  rojo              % solución al  problema 2: brinda una sola respuesta
More?>;                % al responder con el símbolo  ?  busca otra solución
X =  verde             % solución al  problema 2: segunda respuesta
encontrada
>solveall?.            %  cláusula 5: se introduce una directiva a Prolog
YES

>color(X)?             % problema 3: busca todas las posibles soluciones
X =  rojo              % solución al  problema 3: ofrece todas las soluciones
X =  verde
X =  azul
```

El predicado **functor (T, F, N)** es muy útil para el análisis y construcción de estructuras, se compone de tres términos; el primero (T) es una estructura, el segundo (F) es el predicado o función de la estructura, mientras que F refleja la "aridad" (es decir el número de términos que contiene el predicado dentro de la estructura). Los primeros dos argumentos (T, F) no pueden ser listas, ni tampoco se pueden usar variables para definir al segundo y tercer término (F, N). Por lo que respecta a su aplicación es posible aprovecharlo como lo hace el programa 14 de la siguiente forma: Descomposición.- cuando T está instanciado por un átomo o estructura (predicado con términos), F retorna el nombre del predicado y N la aridad

Creación.- produce una estructura en la variable libre T, formada por el nombre del predicado expresado por F y por el número de variables libres (_1, _2, ...) que señala el número identificado por N

>% Programa 14
>% Ejemplo del manejo del predicado "functor"
>functor (amigo (juan, pedro)), F, N)? %problema 1: descomposición de una estructura
F = amigo % solución al problema 1
N = 2
>functor (T, amigo, 2)? %problema 2: descomposición de una estructura
T = amigo (_1, _2) % solución al problema 2

Sin embargo, antes de mostrar un ejemplo relacionado con su aplicación en la generación de código dinámico, es conveniente presentar un par de predicados más que contribuirán a editar el ejemplo; el primero de ellos es la función **call (P)** orientado a invocar la prueba del predicado contenido en la estructurada almacenada por la variable P, para que pueda tener éxito. Es decir, si por ejemplo *P = amigo(juan, carlos)* se puede probar esta declaración mediante el predicado *call(P)* claro está que antes *P* debió haber sido instanciada. Esta función es equivalente a invocar el examen del predicado *amigo(juan, carlos)* su ventaja radica en liberar al programador de editar el nombre del predicado antes de la ejecución del programa, lo que significa hacer código estático clásico.

En complemento a la edición de código dinámico se puede emplear el predicado **univ** representado por los símbolos concatenados **=..** los cuales funcionan acompañados por un par de términos de acuerdo con el formato *X =.. Y* encargándose X de instanciar a un término, mientras que Y a una lista. Como por ejemplo, *amigo(juan, carlos)=.. Y* instanciará a la variable libre Y con el valor *[amigo, juan, carlos]*.

En cambio si el planteamiento es *X =.. [amigo, pedro, ana]* dará vida a una estructura con el predicado *amigo* acompañado por los términos *pedro, ana* es decir: *amigo (pedro, ana).* Dicho a lo anterior se procede a mostrar el programa 15 el cual es capaz de responder a diversos tipos de consulta por medio de un predicado único al momento de plantear la consulta:

>% Programa 15
>% Consultas a diversos tipos de conceptos de una base de datos
>amigo (raul, mario). % cláusula 1: hecho
>amigo (raul, ana). % cláusula 2: hecho
>esposo (raul, elena). % cláusula 3: hecho
>padres (david, raul, elena). % cláusula 4: hecho
%cláusula 5: hace consultas con predicados dinámicos, de acuerdo con los siguientes roles:
% R.- nombre del predicado a consultar
% T.- número de términos que tiene el predicado
% V.- variable libre donde se deposita el resultado de la búsqueda
% P.- variable libre donde se deposita el predicado que se genera dinámicamente
% V.- variable libre donde se deposita una lista con la cual se representa a la cláusula con la que se unificó el predicado P para de esa forma generar una respuesta a la consulta
>relacion(R, T, V):- functor (P, R, T), call(P), P=..V.
>relacion (amigo, 2, X)? %problema 1. consulta las relaciones entre "amigos"
X =[amigo, raul, mario] % solución al problema 1
X =[amigo, raul, ana]
>relacion (esposo, 2, Y)? %problema 2. consulta las relaciones entre "esposos"
Y =[esposo, raul, elena] % solución al problema 2
>relacion (padres, 3, Z)? %problema 3. consulta las relaciones entre "padres"
Z =[padres, david, raul, elena] % solución al problema 3

En complemento al uso de predicados declarativos, el **%** representa un comentario, **exit** indicar a Prolog dar por terminada la ejecución del programa y de la sesión, mientras que **halt** suspende el proceso que se esté realizando sin dar por concluida la sesión.

Con relación a los **predicados aritméticos** Prolog no es un lenguaje orientado al cálculo matemático por lo que el usuario deberá implementar sus propios predicados o bien de realizar alguna interfaz con otra herramienta más especializada, sin embargo se pueden hacer cómputos numéricos elementales de asignación mediante el predicado **is** suma, resta, multiplicación, división y el residuo mediante los símbolos **+ - * / mod** respectivamente, tal como se muestra en el programa 16:

>% Programa 16
>% Uso de predicados aritméticos

>suma (E1, E2, R):- R is E1 + E2 % cláusula 1: suma
>resta (E1, E2, R):- R is E1 - E2 % cláusula 2:resta
>multiplica (E1, E2, R):- R is E1 * E2 % cláusula 3: multiplicación
>divide (E1, E2, R):- R is E1 / E2 % cláusula 4: división
>residuo (E1, E2, R):- R is E1 mod E2 % cláusula 5: residuo

%problema 1. operaciones aritméticas
>suma (1, 2, R1), resta (R, 0, R2), multiplica (R2, 3, R3), divide (R3, 4, R4), residuo(R3, 4, R5)
? R1 = 3
% solución al problema 1
R2 = 3
R3 = 9
R4 = 2
R5 = 1

En lo concerniente a los **predicados de comparación** Prolog emplea símbolos convencionales, abreviaturas y palabras reservadas, los cuales son presentados y ejemplificados por el programa 17 que se muestra a continuación:

= igualdad entre dos términos: *X = Y*
eq igualdad entre dos términos: *X eq Y*

<> desigualdad entre dos términos: *X <> Y*
neq desigualdad entre dos términos: *X neq Y*
> mayor que: *X > Y*
< menor que: *X < Y*
>= mayor o igual que: *X >= Y*
=< menor o igual que: *X =< Y*

var determina si el término es una variable libre: *var(X)*
nonvar determina si el término no es una variable libre: *nonvar (X)*
atom identifica si el término es atómico (constante, variable instanciada con un valor constante): *atom (X)*
integer precisa si el término es un valor entero: *integer (X)*
, representa al operador lógico "and" para relacionar a los predicados que son condiciones de una regla
; corresponde al operador lógico "or" para relacionar a los predicados que son condiciones de una regla
not niega al resultado generado por la prueba hecha a un predicado

```
>% Programa 17
>% Uso de predicados de  comparación

>igual1 (A, B):- A = B.                  % cláusula 1: =
>igual2 (A, B):- A eq B.                 % cláusula 2:   eq
>diferente1 (A, B):- A <> B.             % cláusula 3: <>
>diferente2 (A, B):- A neq B.            % cláusula 4: neq
>mayor  (A, B):-A > B.                   % cláusula 5:  >
>menor  (A, B):-A < B.                   % cláusula 6:  <
>mayorIgual  (A, B):- A >= B.            % cláusula 7:  >=
>menorIgual  (A, B):- A =< B.            % cláusula 8:  =<
> variable  (A):- var  (A).              % cláusula 9:  var
> nonvariable (A):- nonvar (A).          % cláusula 10:  nonvar
>atomo(A):- atom(A).                     % cláusula 11:  atom
>entero(A):- integer(A).                 % cláusula 12:  integer
>falso(A):- not(A).                      % cláusula 13:  not
>y(A):- A > 2, A < 9.                    % cláusula 14:  and ,  mediante el rango abierto
(2, 9)
>o(A):- A < 2; A > 9.                    % cláusula 15:  or ; fuera del rango abierto (2, 9)

%serie 1 de problemas:  comparación de igualdad con sus respuestas
>igual1 (1, 2)? NO
>igual (ana, ana)? YES
 >diferente1 (1,2)? YES
 >diferente2 (ana,
ana)? NO

% serie 2 de problema. comparación de menor - mayor con sus respuestas
>mayor
(1, 2)?
NO
> menor (1, 2)? YES
> mayorIgual (3,3)? YES
> menorIgual (4,5)? NO

%serie 3 de problemas. comparación de variable,  átomos y enteros con sus respuestas
>var (X)? YES
> var (2)? NO
 >nonvar (X)? NO
> nonvar (2)? YES
>atomo (3)?YES
>atomo ( [a, b])? NO
>entero (3)? YES
>entero (5.5)? NO

% serie 4 de problemas. comparación con falso, and or con sus respuestas
>falso (o(11))?
> y (5)? YES
> y (11)? NO
> o (1)? YES
>o (4)? NO
> o (11)? YES
>falso (o(4))?YES
>falso (o(11))? NO
```

En cuanto a los **predicados de "entrada" y "salida"** que Prolog pone al alcance del usuario para captar información de dispositivos de "entrada" como el teclado y enviar datos a unidades de "salida" como el video, los más comunes son representados por una serie de predicados reservados con la siguientes características y forma de aplicación ilustrada por el programa 18:

con: y **see(con:)** representa al teclado como unidad de "entrada" y a la consola como medio de "salida"
read(X) lee el carácter, cadena o cadenas cuya terminación emplea un **punto** provenientes de la unidad de "entrada",depositando el valor en la variable libre X
getc(X) obtiene el siguiente carácter de la cadena de "entrada" actual, almacenando el valor en la variable libre X
write(X) y **print(X)** escribe el valor del término X en el dispositivo actual de "salida"
putc(X) envía el valor representado por el término X al dispositivo actual de "salida"
tab(N) envía N caracteres en blanco "espacios" a la unidad actual de "salida"
nl da por terminada la línea de salida actual para posicionarse al comienzo de la siguiente
ascii (C, N) cuando C está instanciado por cualquier carácter, deposita en N su valor de representación decimal, mientras que si es N el término que contiene un valor numérico, almacena en la variable libre C el carácter correspondiente

```
>% programa 18
>% empleo de predicados de entrada y salida

>% serie 1 de ejercicios que declaran dispositivos
>consult ("con:")?          %declara el teclado y consola como dispositivos de e/s
| amigo (carlos, raul).     % se introduce un hecho por medio del teclado
| amigo (jose, pedro).      % se introduce otro hecho por el mismo medio
| end_of_file.              % identifica el fin de la cadena de entrada y el empleo de la unidad

>% serie 2 de ejercicios que emplean predicados de " entrada" por cadena
>% lee(V):- read(V).        % brinda el valor de una variable que se lee
>%lee(X)?                   % solicita un valor que debe ser alimentado de "entrada"
hola.                       % es el valor proporcionado, terminado con un punto
X = hola                    % valor leído

>% serie 3 de ejercicios que emplean predicados de " entrada" por carácter
> prueba :- see("con:"), getc(A), getc(B), A = B.   % declara unidad, lee 2 caracteres y los compara
>prueba?                    % ejecuta prueba 1
vv.                         % se digitan 2 caracteres con el punto al final
YES                         % comparación exitosa
>prueba?                    % ejecuta prueba 2 gh.
NO                          % comparación desigual

>% serie 4  de ejercicios que emplean predicados de "salida" y que aprovechan el  predicado "amigo"
>% introducido como hecho durante la 1er. serie

> lista:- amigo(A, B), write ("amigos"), nl, print(A), tab(5), print(B), nl.
'amigos'                    % despliega el letrero del 1er. Resultado y salta de renglón
 c   r                      % imprime los 2 caracteres con 5 espacios de separación
YES                         % 1er. búsqueda exitosa

'amigos'                    % despliega el letrero del 2do. Resultado y salta de renglón
   c   r                    % imprime los 2 caracteres con 5 espacios de separación
   YES                      % 2da. búsqueda exitosa

>% serie 5  de ejercicios que emplean predicados de "salida" y que aprovechan el  predicado "amigo"
>% introducido como hecho durante la 1er. serie
```

> lista:- amigo(A, B), write ("amigos"), nl, print(A), tab(5), print(B), nl.
'amigos' ***% despliega el letrero del 1er. Resultado y salta de renglón***
c r ***% imprime los 2 caracteres con 5 espacios de separación***
YES ***% 1er. búsqueda exitosa***

>% serie 6 de ejercicios que emplean predicados de "salida" a partir de la creación de caracteres

>numero(N):- ascii (C, N), putc(C), nl.
>numero(64)? ***% solicita imprimir el carácter correspondiente al # ascii 64***
@ ***% imprime el carácter deseado***
YES

>numero(50)? ***% solicita imprimir el carácter correspondiente al # ascii 50***
2 ***% imprime el carácter deseado***
YES

Rastreo

Manipulación de la base de datos, creación, borrado y actualización

La **manipulación de la base de datos** inicia cuando el usuario edita las cláusulas que representan su aplicación mediante el empleo de un Procesador de Textos respetando la sintaxis de la Prolog para codificar hechos y reglas, tal como se ha expuesto en la sección 1.1 "Control Métodos de búsqueda", o bien introduciendo las instrucciones directamente al ambiente de proceso de Prolog.

Cuando se opta por la primer vía, entonces la **creación de la base de datos** es realizada por medio de un **"ambiente externo"** mediante el empleo del programa Editor que el Usuario haya seleccionado (como por ejemplo Note Pad), en cambio al aprovechar la segunda alternativa se opta por la forma denominada "**en tiempo real"** en el que el código que integra la base de datos reside temporalmente en la "memoria de trabajo" de Prolog.

Una vez que se ejecutan procesos para demostrar teoremas o resolver problemas, la base de datos puede sufrir alteraciones provocadas en tiempo de ejecución por el tipo de instrucciones que son capaces de **borrar** y **actualizar** las cláusulas originales, además de **agregar** nuevas. Una vez concluido los procesos que el usuario ha ejecutado en su sesión de Prolog, existe la opción de **almacenar** la base de datos en su estado actual (el que conserve al final después de las modificaciones que haya sufrido), para ello deberá usar el predicado **save** indicando la ruta y nombre de archivo que desea crear en el medio de almacenamiento para preservar la base de datos y poder aprovecharla posteriormente.

Con la opción de **almacenar** la versión disponible de base de datos existente en la memoria de trabajo se otorga al usuario una tercer manera para **crear** bases de datos a partir de los formas iniciales; por lo que en resumen el programador dispone de tres caminos para crear una base de datos.

Por lo que respecta al **borrado** de cláusulas contenidas en la base de datos, hay dos medios, el primero está representado por la vía del **ambiente externo** donde el archivo que la almacena es editado mediante el Procesador de Textos que ofrece facilidades para eliminar las instrucciones indeseadas. La segunda vertiente aparece en **tiempo real** mediante el empleo de los predicados **retract** y **retractall** acompañados por el nombre del predicado a dar de baja, para que al momento de ejecutar esa instrucción Prolog elimine las ocurrencias de dicho predicado en la base de datos.

De la misma forma, la **actualización** de la base de datos se implementa a través de dos vías, la conocida como **ambiente externo** editando el archivo y alternado las instrucciones al gusto del usuario para proceder a guardarlo con los nuevos cambios; y la segunda opción en **tiempo real** mediante el empleo del **retract** y **retractall** para borrar las ocurrencias del predicado a modificar y luego a través del uso de la instrucción **asserta** y **assertz** en el que se introducen nuevas instancias del predicado que previamente se borró pero ya con el nuevo formato, términos y valores deseados acordes a los cambios requeridos.

Inclusive para agregar nuevas declaraciones para predicados ya existentes en la base de datos original o actual, así como para incorporar nuevos en tiempo de ejecución se puede emplear la pareja de predicados **asserta** y **assertz** con lo cual se podrá incorporar código dinámicamente en tiempo de ejecución, haciendo aplicaciones más sofisticadas e interesantes.

Para ilustrar el empleo de los predicados reservados Prolog para la manipulación en **tiempo real** de la base de datos se presenta a continuación el programa número 19, compuesto por varias cláusulas originales, de las cuales algunas de ellas son borradas o actualizadas posteriormente, para que al final se proceda a **salvar** la versión resultante de la base de datos.

Finalmente por lo que respecta al **rastreo** que en momentos de depuración el usuario se ve en la necesidad de implementar a efecto de monitorear el funcionamiento de su aplicación, analizar el flujo de trabajo, la instanciación de valores en las variables y comprender la generación de los resultados que arroja el programa, Prolog ofrece el empleo de los predicados **trace y notrace** sin argumentos para habilitar el rastreo o inhibirlo a lo largo de la ejecución de la aplicación, tal como lo muestra el programa 19:

```
>% Programa 19
>% Uso de predicados de  comparación

>amigo (juan, raul).          % cláusula 1: hecho
>amigo (juan, tomas).         % cláusula 2 hecho
>amigo (juan, lala).          % cláusula 3: hecho

>%problema 1: borrado de cláusulas
>% problema 1:  borra las ocurrencias del predicado "amigo" que tengan "raul" como 2do término
>retract  (amigo(_, raul ) )? YES
>% borra la cláusula 1, pues es la única que tiene como 2do término "raul"

>% problema 2:  borrar las ocurrencias del predicado "amigo" que tengan "juan" como 1er término
>retract  (amigo(juan, _ ) )? YES
>% borra la cláusula 1 y 3, pues tienen como 1er término "juan", dejando vacía la base de datos

>% problema 3: inserta hechos con el  predicado "amigo" al inicio de la base de datos
>asserta  (amigo(maria, juan ) )? YES
>% agregó la cláusula 1, con el  predicado "amigo"

>asserta  (amigo(maria, tomas ) )? YES
>% agregó la cláusula 2, con el  predicado "amigo" al inicio de la base de datos

>assertz  (amigo(maria, pedro ) )? YES
>% agregó la cláusula 3, con el  predicado "amigo" al final de la base de datos

>% problema 4: modifica los hechos cambiando el  predicado "amigo"por el de "amiga"
>% e  intercambiando  los dos términos. Para cada cláusula de la base de datos
```

>% recupera un hecho con el predicado "amigo" e instancia las variables libres A, B con los términos
>% constantes de la cláusula con la que unifica borra la cláusula con la que se unificó
>% inserta una nueva cláusula al inicio de la base de datos con el predicado "amiga" y los términos
>% preservados en las variables A, B en un orden inverso: B, A
>invierte:- amigo(A, B), retract(amigo(A, B)), asserta (amiga(B, A)).
>invierte? %procede a la actualización de la base de datos
YES YES YES
>% se modificaron las cláusulas originales "amigo" por el predicado "amiga" invirtiendo términos
>% problema 5: Rastreo de la ejecución de una regla que restaura la base de datos original

>invierte:- trace, amiga (A, B), retract(amiga(A, B)), untrace, asserta (amigo(B, A)).
CALL (0) > amiga (_0. _1)
EXIT (0) > amiga (maria, tomas)
CALL (0) > rertract (amiga (maria, tomas))
EXIT (0) > rertract (amiga (maria, tomas))
:
: YES

CALL (0) > amiga (_0. _1)
EXIT (0) > amiga (maria, juan)
CALL (0) > rertract (amiga (maria, juan))
EXIT (0) > rertract (amiga (maria, juan))
:
: YES

CALL (0) > amiga (_0. _1)
EXIT (0) > amiga (maria, pedro)
CALL (0) > rertract (amiga (maria, pedro)) EXIT (0) > rertract (amiga (maria, pedro))
:
: YES

>% se han alterado los predicados con el despliegue del rastreo correspondiente
>% problema 6: almacenamiento de la base de datos, salvándola en el archivo "p18.txt"
>save ("p19.txt")? YES

>% problema 7: borrado de todos los predicados "amigo" de la base de datos
>retractall (amigo(_, _))? YES
>% con ellos se ha vuelto a limpiar la base de datos, quedando vacía

Carga y aplicación de la base de datos

En las secciones anteriores se apreciaron diversas formas de crear una base de datos, tanto en el **ambiente exterior** como en **tiempo real**, lo importante es que al final existe un archivo con las cláusulas originales o actualizadas que representan la base de datos de la aplicación que posteriormente interesa aprovechar para la solución de nuevos problemas, por tal motivo es necesario que independientemente del origen de la base de datos, al final exista o se genere un archivo físico donde residan las instrucciones que más tarde volverán emplearse.

Durante una sesión de Prolog se puede **cargar** una base de datos previamente creada por medio del predicado **consult (X)** en el que el término representa la ruta y el nombre del archivo donde reside la base de datos que se desea explotar al **cargarla** a la memoria de trabajo del ambiente de ejecución. Una vez realizada esta operación se pueden emplear algunos predicados para ejercer diversas **aplicaciones** sobre la base de datos, como por ejemplo el listarla mediante la cláusula **listall** y el acceso a las partes de una regla mediante **clause(X, Y)** en donde el primer término X representa a la conclusión y el segundo Y a la condición; tal como se aprecia en el programa 20:

>% Programa 20
>% Carga y apliaci{on de la Base de datos
>consult("p19.txt")? % este planteamiento carga las cláusulas del archivo 19
YES
>listall? % exhibe las cláusulas recién cargadas
amiga (maria, juan).
amiga (maria, tomas). amiga (maria, pedro).
invierte:- amigo(A, B), retract(amigo(A, B)), asserta (amiga(B, A)).
YES
>clause(invierte, Y)? % solicita la condición de la regla cuya conclusión es "invierte"
Y = amigo(_7, 6), retract(amigo(_7, 6)), asserta (amiga(_6, _7))

Operaciones especializadas para uso de archivos

Si bien Prolog no es un lenguaje orientada al proceso de información masiva, si cuenta con facilidades para el acceso a archivos que contienen información en el formato preestablecido por el usuario, así como para grabar los resultados intermedios y finales que le interesan bajo las convenciones que el propio diseñador establezca. Para tal efecto se cuentan con un repertorio de predicados con las siguientes características y formas de empleo, tal como aparece en el programa 21:

tell(X) abre el archivo especificado por X como de "salida"
telling(X) determina si el valor de X corresponde al archivo que actualmente funge como de "salida"
told cierra el archivo de "salida" declarado por X
see (X) abre el archivo especificado por X como de "entrada"
seeing(X) determina si el valor de X corresponde al archivo que actualmente funge como de "entrada"
seen cierra el archivo de "entrada" declarado por X
eof detecta el final del archivo que se está leyendo

>% Programa 21
>%Operaciones especializadas para uso de archivos

>% Serie 1 de ejercicios de grabación
>amiga (maria, juan). % declaración de hechos
>amiga (luisa, tomas).
>abre(X):- tell(X), !. % abre archivo X de salida

>% ciclo encargado de acceder las cláusulas "amiga", para grabar los términos con salto de renglón
>% en cada uno de ellos en el archivo de salida, gracias al uso del predicado "fail" encargado
>% de provocar el retroceso para acceder a todos los hechos
>graba :- amiga(X, Y), write(X), nl, write(Y), nl, fail.
>% hecho usado para concluir el ciclo "graba", una vez que se han grabado todos los valores
>graba.
>cierra :- told, !. % cierra el archivo de salida

>% ciclo encargado de acceder las cláusulas "amiga", para grabar los términos con salto de renglón
>% en cada uno de ellos en el archivo de salida, gracias al uso del predicado "fail" encargado
>% de provocar el retroceso para acceder a todos los hechos
>graba :- amiga(X, Y), write(X), nl, write(Y), nl, fail.
>% hecho usado para concluir el ciclo "graba", una vez que se han grabado todos los valores
>graba.
>cierra :- told, !. % cierra el archivo de salida
>% invoca la apertura de archivo, acceso y grabación de hechos y el cierre del archivo
>abre(datos), graba, cierra?
YES % problema resuelto, archivo "datos" creado

```
%>  estos son los datos grabados
maria
juan
luisa
tomas

>% Serie 2, ejercicios de lectura, se limpia la base de datos, se lee el archivo con los términos
>% se insertan nuevos hechos con el predicado "amigos" y se despliegan
>retractall(amiga(_, _))?          % se borran todos los hechos para limpiar la base de datos
>listall?                          % se demuestra que está vacia la base de datos
YES
>abre (X):- see(X), !.             % abre el archivo X de "entrada"
>cierra :- seen, !.                % cierra el archivo de "entrada" X
>% ciclo encargado de leer la pareja de términos a grabar con el predicado "amiga" como un hecho
>ciclo :- read (V), V <> "end_of_file", read (W), W <> "end_of_file", asserta (amiga(W)), ciclo.
>ciclo.
>control (X) :- abre(X), ciclo, cierra, !.   % control de la apertura, lectura y clausura del archivo

>control(datos)?                   % se invoca al proceso de lectura y carga de un archivo
YES                                % proceso ejecutado
YES

>listall?                          %muestra los hechos generados
amiga (maria, juan).
amiga (luisa, tomas).
abre (X):- see(X), !.              % y las cláusulas previas
cierra :- seen, !.
ciclo :- read (V), V <> "end_of_file", read (W), W <> "end_of_file", asserta
(amiga(W)), ciclo.
ciclo.
control (X) :- abre(X), ciclo, cierra, !.
YES
```

Bibliografía Recomendada:

"Razonando con Haskell. Un curso sobre programación funcional" B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallard Ed. Thomson , 2004.

"Introducción a la programación funcional con Haskell" (2ed) R. Bird Ed. Prentice-Hall, 1999.

"The Craft of Functional Programming" cS. Thompson

Richard Bird. Introducción a la Programación Funcional con Haskell. Prentice-Hall, 2000.

Seif Haridi and Peter van Roy. Concepts, Techniques, and Models of Computer Programming. The MIT Press, 2004.

Paul Hudak. The Haskell School of Expression. Learning functional programming through multimedia. Cambridge University Press, 2000.

Ulf Nilsson and Jan Maluszynski Logic, Programming and Prolog (2ed) John Wiley & Sons Ltd, 2000.

Rinus Plasmeijer and Marko van Eekelen. Functional Programming and Parallel Graph Rewriting. Addison-Wesley, 1993.

Ed. Addison-Wesley, 1999. Adicional
K. R.. Apt. From Logic Programming to Prolog. Prentice Hall. 1997.
C. S. Clocksin, C. W. Mellish. Programacion en Prolog. Gustavo Gili. 1988.
J. Lloyd. Foundations of Logic Programming. Springer-Verlag. 1987.
L. Sterling, E. Shapiro. The Art of Prolog. MIT Press. 1986.

Graham Hutton; Programming in Haskell; Cambridge University Press, 2007;
* Alejandro Serrano Mena; Beginning Haskell: A Project-Based Approach; Apress, 2014;
* R. Bird; Introducción a la Programación Funcional con Haskell; Segunda edición, Prentice Hall, 2000;
* B.C. Ruiz, F. Gutiérrez, P. Guerrero, J.E. Gallardo; Razonando con Haskell: un curso sobre programación funcional; Thomson, 2004;
Libros de programación lógica
* L.Sterling, E.Shapiro; The Art of Prolog. Advanced Programming Techniques; The MIT Press, 2ª Edición, 1994;
* P. Julián, M. Alpuente; Programación Lógica, Teoría y Práctica; Pearson, 2007;
* W.F. Clocksin, C.S. Mellish; Programming in Prolog Using the ISO Standard; Springer Verlag, 5ª edición, 2003;

Sitios consultados

http://www.ida.liu.se/~ulfni/lpp/bok/bok.pdf

http://babel.ls.fi.upm.es/teaching/

http://djaramillo2dani.blogspot.mx/2011/04/guia-practica-uno-1-estructura-228106.html

http://marcelo-trabajo.blogspot.mx/

http://cursos.aiu.edu/Lenguages%20de%20Programacion/PDF/Tema%201.pdf

http://algoritmosylenguajes.blogspot.mx/2008/05/unidad-iii_31.html