El Lenguaje de Programación PROLOG

De M. Teresa Escrig:

A mi marido Pedro Martín,
que aunque físicamente ya no esté aquí,
sigue presente en mi corazón y en mi recuerdo.

A Daniel y a mis padres,
por su amor incondicional que me ha ayudado a superar momentos muy duros.

De Julio Pacheco:

A Sonia,
por la entrega que me hace poner en todo lo que vivimos juntos.

De Francisco Toledo:

A Sergio y Mónica

Prefacio

Este libro ha nacido como resultado de ocho años de experiencia en la enseñanza del lenguaje de programación PROLOG, impartida en cuarto curso de la carrera de Ingeniería Informática de la Universidad Jaume I de Castellón.

La primera vez que me puse a programar en PROLOG, para el desarrollo de mi tesis doctoral, recuerdo que me fascinó la forma de trabajar del lenguaje y los resultados que se podían obtener con él. Claro que rompía los esquemas de otros lenguajes de programación que conocía, como el Pascal y el C, lenguajes orientados al procedimiento a seguir para resolver el problema. PROLOG tiene un punto de vista más descriptivo o declarativo, es decir, especifica aquello que se quiere conseguir para resolver el problema, no cómo se va a resolver. En todos estos años que llevo en la enseñanza de este lenguaje, eso mismo les ha pasado a todos mis alumnos y por eso sigo disfrutando de su enseñanza. Al finalizar las tres horas de prácticas semanales, tengo que invitar a los alumnos, lo más amablemente posible y con una sonrisa de satisfacción y complicidad en la boca, a que abandonen la sala de ordenadores porque otro profesor con sus alumnos están esperando en la puerta.

PROLOG es un lenguaje de programación especialmente indicado para modelar problemas que impliquen objetos y las relaciones entre ellos. Está basado en los siguientes mecanismos básicos: unificación, estructuras de datos basadas en árboles y backtracking automático. La sintaxis del lenguaje incluye la declaración de hechos, preguntas y reglas. Con la definición de este pequeño conjunto de conceptos se consigue un lenguaje de programación muy potente y flexible, ampliamente utilizado (junto con el lenguaje de programación LISP) en aplicaciones que utilizan técnicas de Inteligencia Artificial.

PROLOG tiene sus fundamentos en la lógica matemática. En el plan de estudios vigente en nuestra Universidad desde su comienzo en 1991, tanto el lenguaje PROLOG como sus fundamentos lógicos, se estudian en la misma asignatura “Lenguajes de Programación III”, en cuarto curso de la Ingeniería Informática. Los fundamentos lógicos se ven en la parte teórica, en 25 horas, y el lenguaje PROLOG en la parte de laboratorio, en 50 horas. Ambas partes comienzan a la vez, aunque la parte teórica sea el fundamento de la parte práctica de la asignatura. Sin embargo, como Ivan Bratko apunta en su libro “PROLOG. Programming for Artificial Intelligence”: una introducción matemática fuerte no es conveniente si el propósito es enseñar PROLOG como una herramienta de programación práctica. Así que el presente libro sólo contiene los principios de enseñanza del PROLOG y sus principales aplicaciones, tal y como son utilizadas en parte práctica de dicha asignatura. Otro volumen de este mismo libro titulado “Principios de la Programación Lógica” contiene los fundamentos en lógica de primer orden, tal y como son utilizados en la teoría de la misma asignatura.

En este momento y hasta que se publique, este libro de teoría puede conseguirse en la página web del grupo de investigación “Intelligent Control Systems” de la

Universidad “Jaume I” (http://ics.uji.es). También están disponibles en esta página una selección de proyectos resueltos por los alumnos.

El contenido del libro se puede estructurar en tres partes. La parte I la forman los *capítulos 1, 2, 3* y *4* que constituyen la base para el aprendizaje del lenguaje PROLOG, incluyendo el manejo de listas y el corte en PROLOG. La parte II está compuesta por los cuatro capítulos siguientes; teniendo los *capítulos 5, 6* y *7* las principales aplicaciones de PROLOG, como las bases de datos, las gramáticas y los sistemas expertos, y en el *capítulo 8* aparece una extensión del PROLOG a la programación lógica basada en restricciones. La parte III contiene el *capítulo 9*, que consiste en el enunciado de varios proyectos de envergadura mediana, en el que se utilizará el PROLOG como lenguaje de programación; un *apéndice* con la solución a una selección de ejercicios; y la *bibliografía* referenciada en todo el libro. Cada capítulo contiene, además, una bibliografía específica a cada tema, para facilitar su estudio cuando es elegido como tema de proyecto.

En cada uno de los capítulos, la explicación del contenido está abundantemente intercalada con ejemplos, para que el alumno los pruebe y comprenda su funcionamiento, y ejercicios planteados que serán desarrollados por el alumno en el laboratorio.

La idea es que todos los capítulos sean vistos y practicados en una sesión de laboratorio de 3 horas, excepto los capítulos que hacen referencia al tratamiento de listas y los que se refieren a los campos de aplicación del PROLOG como las bases de datos, las gramáticas y los sistemas expertos, cuya duración prevista es de dos sesiones de prácticas de 3 horas cada una. El objetivo es que quede tiempo de clase guiada por el profesor para el desarrollo de un proyecto de mediana envergadura, ya que el paso de solucionar problemas pequeños a solucionar problemas de mediana envergadura no es trivial.

Este libro está orientado especialmente a los estudiantes de PROLOG en una asignatura de características similares a las descritas en este prefacio, es decir para alumnos que tengan unos conocimientos de ordenadores, bases de datos y gramáticas, que podrían adquirirse en tres años de docencia en una carrera de informática. Pero también está escrito para aquel lector que quiera aprender el lenguaje PROLOG por su cuenta, y que no quiera quedarse sólo con aprender a utilizarlo, sino que busque profundizar en su aplicación a áreas de Inteligencia Artificial.

En este libro se ha elegido como entorno de programación ECLiPSe, que contiene el PROLOG estándar y algunas extensiones para manejar bases de datos (incluyendo bases de datos declarativas y bases de conocimiento), programación lógica basada en restricciones (en concreto programación lógica basada en restricciones sobre dominios finitos), y programación concurrente, entre otras extensiones. La elección de ECLiPSe se hace por los siguientes motivos:

1) es compilado y por tanto más eficiente que otros entornos interpretados;
2) tiene un entorno agradable de trabajo, con un depurador de programas bastante bueno, que permite seguir paso a paso el funcionamiento del PROLOG, lo cual es necesario para comprender su funcionamiento y en definitiva para aprender a programar en PROLOG;

3) se dispone de una licencia gratuita para docencia;

4) existe una versión para LINUX y otra para Windows NT (con posibilidad de acoplarlo a Windows 95), que permite a los alumnos instalárselo en sus ordenadores personales para hacer prácticas en casa;

5) es un potente entorno profesional que se sigue desarrollado actualmente.

Sin embargo, casi todos los predicados predefinidos de ECLiPSe pueden encontrarse en otro entorno de programación, como puede ser SICStus Prolog.

Nuestro más ferviente deseo al escribir el libro es que el lector experimente el lenguaje PROLOG y su aprendizaje como un desafío intelectual excitante.

Francisco Toledo Lobo
Julio Pacheco Aparicio
M. Teresa Escrig Monferrer

Julio de 2001

Índice

1. Fundamentos de PROLOG

CONTENIDO

1.1. Introducción

PROLOG es un lenguaje de programación declarativo. Los lenguajes declarativos se diferencian de los lenguajes imperativos o procedurales en que están basados en formalismos abstractos (PROLOG está basado en la lógica de predicados de primer orden y LISP, otro lenguaje de programación declarativa, en lambda calculo), y por tanto su semántica no depende de la máquina en la que se ejecutan. Las sentencias en estos lenguajes se entienden sin necesidad de hacer referencia al nivel máquina para explicar los efectos colaterales. Por tanto, un programa escrito en un lenguaje declarativo puede usarse como una especificación o una descripción formal de un problema. Otra ventaja de los programas escritos en lenguajes declarativos es que se pueden desarrollar y comprobar poco a poco, y pueden ser sintetizados o transformados sistemáticamente.
PROLOG es un lenguaje de programación muy útil para resolver problemas que implican objetos y relaciones entre objetos. Está basado en los siguientes mecanismos básicos, que se irán explicando a lo largo de este capítulo:

- Unificación
- Estructuras de datos basadas en árboles
- Backtracking automático

La sintaxis del lenguaje consiste en lo siguiente:

- Declarar hechos sobre objetos y sus relaciones
- Hacer preguntas sobre objetos y sus relaciones
- Definir reglas sobre objetos y sus relaciones

Cada una de estas partes se explica en los siguientes apartados.

1.2. Los hechos PROLOG

Para explicar los fundamentos de PROLOG vamos a utilizar el típico ejemplo de las relaciones familiares.
Para decir que Laura es uno de los dos progenitores de Damián, podríamos declarar el siguiente hecho PROLOG:

```
progenitor(laura, damian).
```

"progenitor" es el nombre de la relación o nombre de predicado y "laura" y "damian" son los argumentos. Los hechos acaban siempre con punto. Nosotros interpretaremos que Laura, primer argumento de la relación, es la madre de Damián, segundo argumento de la relación. Sin embargo, este orden es arbitrario y cada programador puede darle su propio significado. Los nombres de las relaciones y los argumentos que se refieren a objetos o personas concretas se escribirán con minúscula.
Otros ejemplos de hechos pueden ser los siguientes:

```
le_gusta_a(juan,maria).
valioso(oro).
tiene(juan,libro).
da(juan,libro,maria).
```

Los nombres también son arbitrarios y el programador decidirá la interpretación que haga de ellos. La relación le_gusta_a(juan,maria) es equivalente a la relación a(b,c), aunque para que la interpretación sea más sencilla, se recomienda que los nombres se elijan de forma que ayuden a su interpretación.
Los hechos no tienen que reflejar el mundo real necesariamente, pero será única y exclusivamente lo que PROLOG tomará como verdadero. Un conjunto de hechos (también llamados cláusulas), junto con un conjunto de reglas, forman lo que se llama una base de datos PROLOG.

1.3. Las preguntas PROLOG

Sobre un conjunto de hechos se pueden realizar una serie de preguntas. Por ejemplo:

```
?- le_gusta_a(juan,maria).
```

PROLOG busca automáticamente en la base de datos si existe un hecho que se puede unificar (es decir, tiene el mismo nombre de predicado, el mismo número de argumentos —o aridad — y cada uno de los argumentos tiene el mismo nombre, uno a uno) con el hecho que aparece en la pregunta. PROLOG contestará "SI" si encuentra ese hecho y "NO" si no lo encuentra. La contestación "NO" no implica que el hecho sea falso (aunque sí lo sea para nuestra base de datos), sino que no se puede probar (en general) que sea verdadero con el conocimiento almacenado en la base de datos.

Para realizar preguntas más interesantes, como por ejemplo, qué le gusta a María o cuáles son los padres de Damián, se usarán las variables. En PROLOG las variables empiezan por mayúscula. Por ejemplo:

```
?-le_gusta_a(maria,X).
?-progenitor(Y,damian).
```

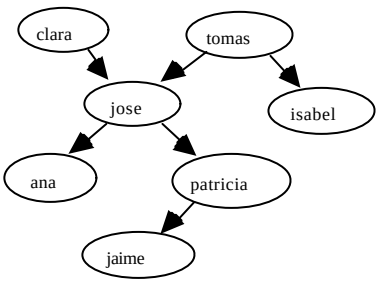
Para obtener la o las respuestas, PROLOG recorre la base de datos hasta encontrar el primer hecho que coincide con el nombre de la relación y su aridad y con los argumentos que no son variables. Marca esa posición para poder recordar dónde se quedó en el recorrido por la base de datos. La o las variables se instancian al valor que le corresponde según el lugar que ocupan en la relación, y ese valor es la respuesta que proporciona PROLOG. Si pulsamos RETURN no obtendremos más que la primera respuesta. Si se quieren obtener todas las respuestas (para el caso de que exista más de una) se teclea ";". Cuando pulsamos ";", PROLOG sigue automáticamente la búsqueda desde la marca de posición en la que se había quedado en la base de datos. Se dice entonces que PROLOG intenta resatisfacer la pregunta. Se desinstancian las variables que se habían instanciado, y sigue buscando otro hecho que coincida sintácticamente con la pregunta. A este mecanismo se le llama backtracking, y PROLOG lo hace automáticamente.
Para resolver preguntas más complejas, como por ejemplo, ¿se gustan Juan y María? o ¿tienen Ana y Damián un progenitor común, es decir, son hermanos? o ¿quién es el nieto(s) de Tomás?, se utilizan conjunciones de objetivos, es decir, preguntas separadas por comas, que en PROLOG corresponden a la "Y" lógica.

```
?-le_gusta_a(juan,maria), le_gusta_a(maria,juan).
?-progenitor(X,ana), progenitor(X,damian).
?-progenitor(tomas,X), progenitor(X,Y).
```

En una conjunción de objetivos correspondientes a la misma pregunta, la misma variable se refiere al mismo objeto (en el segundo ejemplo, X se refiere a la misma persona). En preguntas u objetivos distintos, el mismo nombre de variable se refiere a distintos objetos (la X del segundo y el tercer ejemplo se refieren a personas distintas).
Para buscar las soluciones a una conjunción de objetivos, PROLOG establece una marca de posición para cada objetivo, y recorre toda la base de datos en la búsqueda de cada objetivo.

Ejemplo 1.1
Suponemos definida la siguiente base de datos de relaciones familiares:

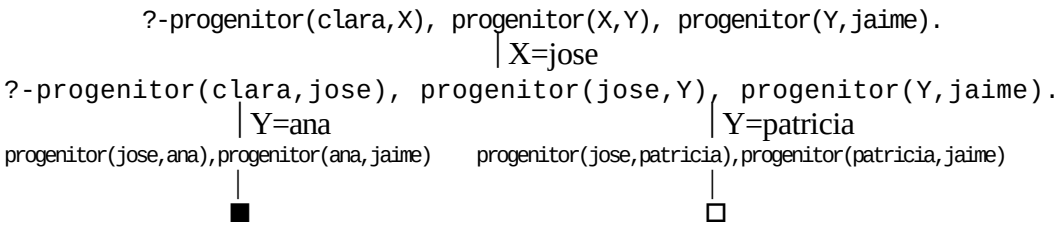
```
progenitor(clara,jose).
progenitor(tomas, jose).
progenitor(tomas,isabel).
progenitor(jose, ana).
progenitor(jose, patricia).
progenitor(patricia,jaime).
```



Para demostrar si Clara es bisabuela de Jaime, utilizaríamos la siguiente conjunción de objetivos:

```
?-progenitor(clara,X), progenitor(X,Y), progenitor(Y,jaime).
```

Para explicar el funcionamiento del backtracking que permite obtener todas las respuestas de la anterior conjunción de objetivos, vamos a utilizar el esquema de deducción natural.



Explicación: Se intenta satisfacer X del primer objetivo de la conjunción de objetivos, progenitor(clara,X). El objetivo coincide con el primer hecho de la base de datos, donde se quedará su marca de posición asociada, y la variable X se instancia a "jose". Dentro de una misma pregunta, las variables con el mismo nombre se refieren al mismo objeto, por tanto se sigue comprobando si se puede demostrar que "progenitor(jose,Y), progenitor(Y,jaime)". Para el objetivo progenitor(jose,Y) se empieza a buscar en la base de datos desde el principio, hasta encontrar un hecho que unifique. El primer hecho que coincide es progenitor(jose, ana), por tanto Y se instancia a "ana" y en este hecho se queda la marca de posición para este objetivo. Ahora se comprueba si se puede demostrar que es verdad progenitor(ana,jaime). Como este hecho no está en la base de datos, esta no es una solución, y se señala en el árbol de derivación natural, con el símbolo ■. La última variable que se había instanciado se desinstancia, y se sigue buscando en la base de datos, desde la marca de posición asociada, otro hecho que coincida con el objetivo actual. En este caso, progenitor(jose,patricia). Y se instancia a "patricia". Ahora se demostrará si se cumple progenitor(patricia,jaime). Como este sí que es un hecho de la base de datos, la instanciación X=jose e Y=patricia es una solución a la conjunción de objetivos del ejemplo, y se señala en el árbol con el símbolo □. Y no hay más soluciones, porque no hay otro hecho de la base de datos que coincida con progenitor(clara,X).

Ejercicio 1.1

Dada la base de datos familiar del ejemplo 1.1, se pide la respuesta de PROLOG y el enunciado verbal de las siguientes preguntas:

a) ?-progenitor(jaime,X).
b) ?-progenitor(X,jaime).
c) ?-progenitor(clara,X), progenitor(X,patricia).
d) ?-progenitor(tomas,X), progenitor(X,Y), progenitor(Y,Z).

Ejercicio 1.2

Dada la base de datos familiar del ejemplo 1.1, formula en PROLOG las siguientes preguntas:

a) ¿Quién es el progenitor de Patricia?
b) ¿Tiene Isabel un hijo o una hija?
c) ¿Quién es el abuelo de Isabel?
d) ¿Cuáles son los tíos de Patricia? (no excluir al padre)

1.4. Las reglas PROLOG

Existe en PROLOG la posibilidad de definir la relación "abuelo(X,Y)" o la relación "tio(X,Y)" como reglas, además de poderlo hacer como hechos o como conjunción de objetivos.

Ejemplo 1.2

```
abuelo(X,Y):- progenitor(X,Z), progenitor(Z,Y).
tio(X,Y):- progenitor(Z,Y), progenitor(V,Z), progenitor(V,X).
```

A la primera parte de la regla se le llama cabeza o conclusión, el símbolo ":-" es el condicional (SI), y a la parte de la regla que está después de ":-" es el cuerpo o parte condicional. El cuerpo puede ser una conjunción de objetivos separados por comas. Para demostrar que la cabeza de la regla es cierta, se tendrá que demostrar que es cierto el cuerpo de la regla.
Por lo visto hasta ahora, las cláusulas PROLOG son de tres tipos: hechos, reglas y preguntas. Las cláusulas PROLOG consisten en una cabeza y un cuerpo. Los **hechos** son cláusulas que tienen cabeza pero no tienen cuerpo. Las **preguntas** sólo tienen cuerpo. Las **reglas** tienen siempre cabeza y cuerpo. Los hechos son siempre ciertos. Las reglas declaran cosas que son ciertas dependiendo de una condición. El programa PROLOG (o base de datos PROLOG) está formado por hechos y reglas y para PROLOG no hay ninguna distinción entre ambas. Las preguntas se le hacen al programa para determinar qué cosas son ciertas.

Ejercicio 1.3

Dada la base de datos familiar del ejemplo 1.1, y suponiendo definidas las siguientes cláusulas:

```
hombre(X).
mujer(X).
progenitor(X,Y).
dif(X,Y):- X\=Y.
```

Donde las 3 primeras cláusulas se definirán como hechos (por tanto no se podrá poner una variable como argumento, ya que una variable haría que el hecho fuera cierto para cualquier objeto) y la última como una regla (donde el símbolo \= significa distinto). Escribir las reglas de PROLOG que expresen las siguientes relaciones:

a) es_madre(X).
b) es_padre(X).
c) es_hijo(X).
d) hermana_de(X,Y).
e) abuelo_de(X,Y) y abuela_de(X,Y).
f) hermanos(X,Y). Tened en cuenta que una persona no es hermano de sí mismo.
g) tia(X,Y). Excluid a los padres.

Con la definición del tipo de reglas anterior se pueden resolver problemas interesantes, sin embargo, la gran potencia del PROLOG está en la definición de reglas recursivas. Como hemos visto en ejemplos anteriores se puede definir la relación progenitor, y las reglas abuelo, bisabuelo, tatarabuelo, etc. En general, puede ser interesante definir la relación predecesor(X,Y). Un predecesor de X podrá ser el progenitor de X. También será predecesor si es abuelo/a, si es tatarabuelo/a, etc., es decir, necesitaríamos un conjunto de reglas como:

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

La definición de varias reglas con el mismo nombre de relación equivale en PROLOG a la "O" lógica o disyunción. Pero la definición de este conjunto de reglas es infinito, nunca terminaríamos de escribirlo. La siguiente regla recursiva nos permitirá definirlo.

Ejemplo 1.3

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

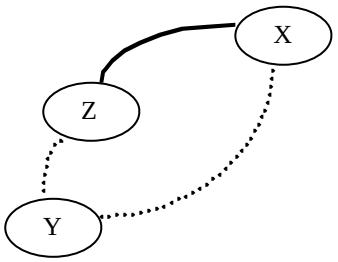


Figura 1.1 Diagrama que explica la definición de predecesor. La línea continua se refiere a un progenitor directo y la discontinua a un predecesor.

La primera definición de predecesor corresponde al caso más sencillo en el que el predecesor corresponde al progenitor. Es lo que llamaremos la regla de salida de la recursividad. Si al intentar comprobar si una persona es predecesora de otra, se halla en la base de datos que la primera persona es progenitora de la segunda, se habrá demostrado y PROLOG responderá que sí. En caso contrario, se intentará demostrar que es predecesor utilizando la segunda regla, que contiene la llamada recursiva.

Ejercicio 1.4

Dada la base de datos familiar del ejemplo 1.1:

a) Define una regla que permita obtener los sucesores de una persona.
b) Comprueba el funcionamiento de PROLOG para obtener los sucesores de Clara. Escribir el árbol de derivación natural.
c) ¿Es una alternativa válida a la definición de predecesor la siguiente?

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

Dibuja un diagrama que explique la definición.

1.5. La sintaxis PROLOG

Los objetos o términos PROLOG pueden ser objetos simples o estructuras (ver figura 1.2). Los objetos simples pueden ser constantes o variables. Las constantes serán átomos o números. Los átomos empiezan con letra minúscula (nunca con números), pueden contener caracteres especiales y pueden ser nombres entre comillas simples. Los números serán enteros o reales, sin una definición explícita de tipos. PROLOG se utiliza para una programación simbólica, no numérica, por eso los enteros se utilizarán por ejemplo para contar el número de elementos de una lista, pero los reales son poco utilizados. Las variables empiezan con mayúscula o con subrayado. Las variables anónimas son aquellas cuyo nombre es sólo el carácter subrayado (_). Se usan cuando no es importante el nombre de la variable o cuando la variable no puede unificar con otra, dentro de la misma cláusula.

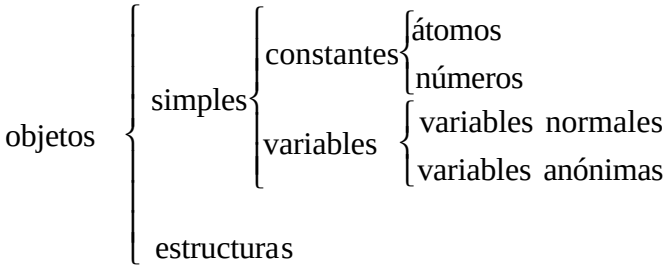


Figura 1.2 Términos PROLOG.

Por ejemplo:

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

"Y" no unifica con otra variable en la definición de la relación "tiene_un_hijo", por lo que es aconsejable sustituirla por una variable anónima (en algunos entornos de programación PROLOG se advierte sobre la conveniencia de ello).

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

Es importante señalar que el alcance de una variable es la cláusula donde aparece, y el alcance de una constante es todo el programa PROLOG.

La sintaxis de las estructuras es la misma que la de los hechos. Los **funtores** de las estructuras son los nombres de los predicados de hechos y reglas. Los argumentos de los hechos y las reglas son los componentes de las estructuras.

Ejemplo 1.4

```
tiene(juan,libro).
tiene(juan,libro(don_quijote,autor(miguel,cervantes),3256)).
```

Las estructuras se pueden representar mediante árboles. Por ejemplo, un punto en 2-dimensiones se puede representar con la estructura punto(X,Y) y un punto en 3-dimensiones con la estructura punto(X,Y,Z). Ambas estructuras son distintas porque tienen distinta aridad o número de argumentos. La estructura en árbol correspondiente sería:

Un segmento se puede representar con 2 puntos, segmento(punto(X1,Y1), punto(X2,Y2)); y un triángulo con 3 puntos, triangulo(punto(X1,Y1), punto(X2,Y2), punto(X3,Y3)).
La operación más importante sobre términos es la unificación. Dos términos pueden unificarse si son idénticos o las variables de ambos términos pueden instanciarse a objetos tales que después de la sustitución de las variables por esos objetos, los términos sean idénticos.

Ejemplo 1.5

```
fecha(D,M,1998) = fecha(D1,mayo,A)
piloto(A,Londres) = piloto(londres,paris)
punto(X,Y,Z) = punto(X1,Y1,Z1)
f(X,a(b,c)) = f(Z,a(Z,c))
```

En el caso de que "?-X=Y" sea una pregunta PROLOG, puede ocurrir varias cosas:

(1) que una de las dos variables no esté instanciada pero la otra sí, entonces el objetivo se satisface y la variable que no estaba instanciada queda instanciada al valor de la variable que estaba instanciada;

(2) que ambas variables estén sin instanciar, entonces el objetivo se satisface y además ambas variables quedan compartidas, es decir, que cuando una de las dos variables se instancie, la otra quedará instanciada a lo mismo;

(3) si X e Y son constantes ambas tiene que ser el mismo objeto, y si son estructuras, tienen que tener el mismo funtor, el mismo número de argumentos y que coincidan todos los argumentos.

Usando solamente la unificación se pueden resolver problemas interesantes.

Ejemplo 1.6

```
vertical(segmento(punto(X,Y1),punto(X,Y2)).
horizontal(segmento(punto(X1,Y),punto(X2,Y)).
```

Un segmento vertical en 2-D está formado por dos puntos, en los cuales la coordenada X coincide. Y un segmento horizontal en 2-D está formado por dos puntos, en los cuales la coordenada Y coincide.

Ejercicio 1.5

Decir si la unificación tiene éxito y cuál es el resultado de la instanciación de las variables en:

```
triangulo(punto(-1,0),P2,P3) = triangulo(P1,punto(1,0),punto(0,Y)).
```

¿A qué familia de triángulos da lugar la instanciación resultante?

Ejercicio 1.6

Con la siguiente definición de segmento:

```
segmento(punto(X1,Y1),punto(X2,Y2)).
```

Representar cualquier segmento línea vertical con X=5.

Ejercicio 1.7

Si representamos el rectángulo por rectángulo(P1,P2,P3,P4) donde Pi son vértices ordenados positivamente, define la relación

```
regular(R)
```

que será verdadero cuando los lados del rectángulo R sean verticales y horizontales.

Ejercicio 1.8

Dado el siguiente programa:

```
f(1,uno).
f(s(1),dos).
f(s(s(1)),tres).
f(s(s(s(X))),N):- f(X,N).
```

¿Cómo se comporta PROLOG ante las siguientes preguntas?

a) ?-f(s(1),A).
b) ?-f(s(s(1)),dos).
c) ?-f(s(s(s(s(s(s(1)))))),C).
d) ?-f(D,tres).

Ejercicio 1.9

Dada la siguiente base de datos familiar:

progenitor(clara,jose).
progenitor(tomas, jose).
progenitor(tomas,isabel).
progenitor(jose, ana).
progenitor(jose, patricia).
progenitor(patricia,jaime).

mujer(clara).
mujer(isabel).
mujer(ana).
mujer(patricia).

hermana_de(X,Y):- mujer(X), progenitor(Z,X), progenitor(Z,Y).

tia(X,Y):- hermana_de(X,Z), progenitor(Z,Y).

Construir el esquema de deducción natural para las siguientes preguntas:

a) ?-tia(isabel,ana).
b) ?-tia(clara,ana).
c) Si añado la cláusula progenitor(tomas, maria), ¿cómo quedarían a) y b) si pulsamos ;?

1.6. Significado declarativo y procedural de los programas

En un lenguaje declarativo puro, sería de esperar que el orden en el que aparecen los hechos y las reglas en la base fuera independiente de los datos, sin embargo en PROLOG no es así.

El significado declarativo tiene que ver sólo con las relaciones definidas por el programa. De esta manera, el significado declarativo determina **cuál** será la salida del programa. Por otro lado, el significado procedural determina **cómo** se ha obtenido esta salida; es decir, como evalúa las relaciones PROLOG.

Si tenemos un conjunto de hechos y reglas con el mismo nombre de relación y la misma aridad, puede ser conveniente que los hechos estén situados en la base de datos antes que las reglas, (sobre todo, si los hechos son excepciones de las reglas). Además también suele ser aconsejable poner la regla para salirse de la recursividad antes que la regla recursiva.

La habilidad de PROLOG para calcular de forma procedural es una de las ventajas específicas que tiene el lenguaje. Como consecuencia esto anima al programador a considerar el significado declarativo de los programas de forma relativamente independiente de su significado procedural.

Es decir, las ventajas de la forma declarativa de este lenguaje son claras (es más fácil pensar las soluciones y muchos detalles procedurales son resueltos automáticamente por el propio lenguaje) y podemos aprovecharlas.

Los aspectos declarativos de los programas son, habitualmente, más fáciles de entender que los procedurales. Esta es la principal razón por la que el programador debe concentrarse en el significado declarativo y evitar distraerse por los detalles de cómo se ejecutan los programas.

El acercamiento a la programación declarativa no es suficiente. Una vez se tenga el concepto claro, y cuando se trabaje con programas grandes, los aspectos procedurales no se pueden ignorar por completo por razones de eficiencia. No obstante, el estilo declarativo a la hora de pensar en PROLOG debe ser estimulado y los aspectos procedurales ignorados para favorecer el uso de las restricciones.

Ejercicio 1.10

Construir el árbol de resolución lineal para la pregunta:

```
?-predecesor(clara,patricia).
```

teniendo en cuenta las siguientes 4 definiciones de predecesor:

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

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

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

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

1.7. El entorno de programación ECLiPSe

ECLiPSe es un entorno de programación que contiene el lenguaje de programación PROLOG y algunas extensiones que permitirán manejar bases de datos (incluyendo bases de datos declarativas y bases de conocimiento), programación lógica basada en restricciones (en concreto programación lógica basada en restricciones sobre dominios finitos), y programación concurrente.

En nuestra Universidad, el software de ECLiPSe está cargado en `anubis` (anubis.uji.es; IP: 150.128.40.100), la máquina que gestiona las prácticas de todas las asignaturas de todos los alumnos. Por tanto para acceder al entorno basta con entrar en `anubis` con la cuenta que cada alumno tiene asignada enla máquina y ejecutar "`eclipse`".

ECLiPSe dispone de un editor interno al que se accede tecleando "[`user`]." desde la entrada de ECLiPSe, del cual se sale pulsando "^D". Sin embargo se recomienda que se abran dos sesiones, en una de ellas se utiliará un editor (el favorito de cada alumno) y en la otra sesión se tendrá abierto ECLiPSe. De esta manera el ciclo "cambiar código/compilar/ejecutar con debugger" se hace cambiando sólo de ventana, sin tener que salirse cada vez del entorno de ECLiPSe.

Un vez generado un fichero con el editor (por ejemplo lo llamaremos "prueba") y grabado, se carga y compila en ECLiPSe mediante el objetivo "[`prueba`]." (terminado en punto).
Para utilizar el debbuger podemos teclear "`trace.`" o "`spy(nombre del predicado y aridad).`". `trace` hace un debbuger minucioso. Tecleando "`c`" (de *creep*) o retorno de carro, pasaremos por todos los subobjetivos que definen nuestro programa PROLOG; con "`s`" (de *skip*) podemos saltar todo lo referente a la cláusula en donde lo tecleemos. "`spy(nombre del predicado y aridad).`" sólo hará el debbuger del predicado en el que hayamos puesto el "`spy`".
Cada vez que hacemos alguna cosa en el entorno ECLiPSe (compilar, ejecutar, hacer una traza, etc.), el prompt del eclipse va cambiando, incrementando de 1 en 1 el número del final de la entrada. Ese número puede utilizarse posteriormente como forma de escritura rápida del comando que contenía esa entrada.
Por ejemplo:

```
<eclipse1>[prueba].
<eclipse2>inversa([a,b,c],L).
<eclipse3>inversa([a,b,c],[c,b,a]).
```

Supongamos que en este punto queremos cambiar el código de "prueba". Después de grabar el fichero, compilaríamos y ejecutaríamos con las siguientes llamadas:

```
<eclipse4>1.
<eclipse5>2.
<eclipse6>3.
```

1.8. Bibliografía

[Clocksin 93] Clocksin, W.F., Mellish, C.S., Programación en PROLOG. Segunda edición. Colección Ciéncia Informática. Editorial Gustavo Gili S.A., 1993.

[Brisset 94] P. Brisset et al., ECLiPSe 3.4 Extensions User Manual, European Computer-Industry Research Center, Munich, Germany, 1994.

[Konigsberger & Bruyn 90] Konigsberger, H., Bruyn, F., PROLOG from the beginning. McGraw-Hill book Company, 1990.

2. Tratamiento de listas en PROLOG

CONTENIDO

2.1. Listas en PROLOG

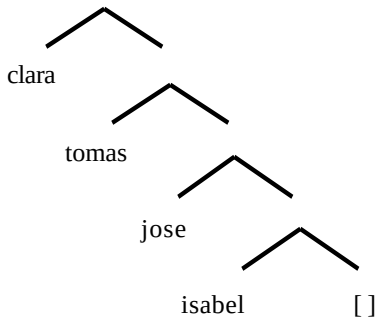
La lista es una estructura de datos simple, muy usada en la programación no numérica. Una lista es una secuencia de elementos tales como:

```
clara,tomas,jose,isabel
```

La sintaxis de PROLOG para las listas consiste en englobar las secuencias de elementos entre corchetes.

```
[clara,tomas,jose,isabel]
```

La representación interna de las listas en PROLOG es con árboles binarios, donde la rama de la izquierda es el primer elemento –o cabeza– de la lista y la rama de la derecha es el resto –o cola– de la lista. De la rama que contiene el resto de la lista también se distinguen dos ramas: la cabeza y la cola. Y así sucesivamente hasta que la rama de la derecha contenga la lista vacía (representado por [])



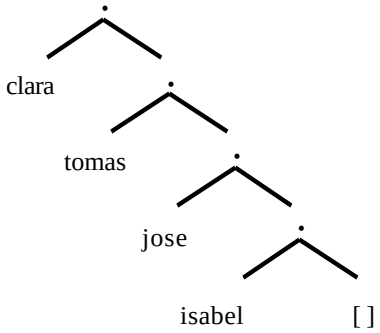
De cualquier forma consideraremos dos casos de listas: Las listas vacías y las no vacías. En el primer caso escribiremos la lista vacía como un átomo de PROLOG, []. Y en el segundo caso, la lista la consideraremos formada por dos partes; el

primer elemento, al que llamaremos la cabeza de la lista; y el resto de la lista al que llamaremos la cola de la lista.
En el ejemplo anterior, la cabeza será `clara` y la cola `[tomas,jose,isabel]`.

Aunque como hemos visto anteriormente no es necesario, en general la estructura lista se representa en PROLOG con el nombre de predicado o funtor ".".

Ejemplo 2.1
```
.(clara,.(tomas,.(jose,.(isabel,[]))))
```

Que PROLOG representará internamente como un árbol:

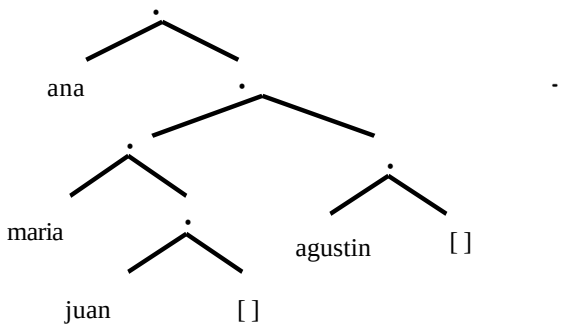


El último elemento siempre es la lista vacía ([]).

Como en PROLOG el manejo de listas es muy usual, se utiliza la notación simplificada con corchetes, que es equivalente a la anterior ([clara, tomas, jose, isabel]). Interiormente PROLOG lo interpretará con la notación árbol.
Los elementos de una lista pueden ser de cualquier tipo, incluso otra lista.

Ejemplo 2.2
```
[ana, [maria, juan], agustin]
```

PROLOG representará internamente como un árbol:



La cabeza y la cola de una lista se pueden separar con el símbolo "|".

Ejemplo 2.3
Las siguientes son expresiones válidas sobre listas y se refieren a la misma lista:

[a,b,c] [a|[b,c]] [a,b|[c]] [a,b,c|[]] [a|X],[Y|[b,c]]

El orden de los elementos en la lista importa y un elemento se puede repetir en una lista, de forma similar a como ocurre en las tuplas y pares ordenados.

2.2. Ejemplos de solución de problemas de listas en PROLOG

La práctica en el manejo de listas con PROLOG es básica para el aprendizaje del lenguaje. Ese es el objetivo de este capítulo.
Para resolver un problema de listas con PROLOG, primero pondremos varios ejemplos de su funcionamiento, para determinar los argumentos que son necesarios en la definición de la relación y la respuesta de PROLOG en distintos casos, que nos indicará el comportamiento deseado de la cláusula.

Ejemplo 2.4
Supongamos que queremos determinar si un elemento es miembro de una lista. Los siguientes serían ejemplos del funcionamiento de la relación "miembro".

```
miembro(b,[a,b,c]).  %PROLOG respondería sí.
miembro(b,[a,[b,c]]).  %PROLOG respondería no.
miembro([b,c],[a,[b,c]]).  %PROLOG respondería sí.
```

El símbolo "%" permite poner comentarios que ocupen sólo la línea donde aparece dicho símbolo. Para comentarios más largos se puede utilizar los símbolos "/*" para empezar y "*/" para terminar.
La siguiente podría ser una definición de la relación "miembro":

```
miembro(X,[X|_]).
miembro(X,[_|R]):-miembro(X,R).
```

El elemento será miembro de la lista si coincide con la cabeza de la lista (independientemente de lo que contenga el resto de la lista, por eso ponemos una variable anónima en su lugar). En este caso el predicado será cierto y PROLOG no comprobará nada más. Si no es así, podrá ser miembro si lo es del resto de la lista, en la llamada recursiva de la segunda definición del predicado. La cabeza de la lista es una variable anónima, porque cuando vaya a comprobar esta segunda regla, es seguro que la primera no se cumple (es decir, el elemento no coincide con la cabeza de la lista), ya que PROLOG busca la cláusula en el orden de aparición en la base de datos.

Ejemplo 2.5
Suponemos que queremos concatenar dos listas y devolver el resultado en una tercera lista, es decir:

```
concatenar([a,b,c],[d,e],[a,b,c,d,e]). % PROLOG respondería que sí.
concatenar([a,b],[1,2,3],X). % PROLOG respondería X=[a,b,1,2,3].
```

Solución

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

La lista resultante se construye (o se comprueba) en la cabeza de la regla recursiva. En la regla que corresponde a la salida de la recursividad se inicializa el argumento resultado a la lista que aparece en el segundo argumento.

Ejercicio 2.1

Utilizar la definición de concatenación para:

a) Descomponer una lista en dos, obteniendo todas las posibles descomposiciones.
b) Borrar algunos elementos de una lista.
Por ejemplo: considera la lista L1=[a,b,c,d,z,z,e,z,z,z,f,g] y borra todos los elementos que siguen a la secuencia z,z,z.
c) Borrar los tres últimos elementos de una lista.
d) Definir de nuevo la operación miembro.
e) Definir la relación para extraer el último elemento de la lista
    e.1) Utilizando la definición de concatenación.
    e.2) Sin utilizarla.

Ejercicio 2.2

Borrar un elemento de una lista.

Ejercicio 2.3

Añadir un elemento en una lista.

Ejercicio 2.4

Comprobar si dos elementos son consecutivos en una lista.

Ejemplo 2.6
Calcular la inversa de una lista.
Ejemplo de utilización:

```
inversa([a,b,c,d],[d,c,b,a]). %PROLOG devolvería sí.
```

Solución 1:

```
inversa([],[]).
inversa([X|L1],L):-
     inversa(L1,Resto),
     concatenar(Resto,[X],L).
```

Solución 2:

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

/* Para las dos últimas reglas es indistinto el orden, porque sólo una de las dos será cierta cada vez */

En esta segunda versión, se utiliza otro predicado inversa/3 (de aridad 3, y por tanto distinto al predicado inversa/2), en donde el segundo argumento se inicializa a lista vacía. En esta versión la lista se construye (o se comprueba) en el cuerpo de la regla de la segunda regla inversa/3 (al contrario de como sucedía en la definición del ejemplo 2.3). Para estos casos, la salida de la recursividad deberá realizar un "volcado" de la lista construida en el segundo argumento en la lista resultante, que aparece en este caso en el tercer argumento. De esta manera el resultado quedará almacenado en la lista resultante ya que el resultado obtenido en el segundo argumento se irá perdiendo a medida que se cierre la recursividad.
Muchas veces, para un mismo problema, se pueden plantear ambas versiones, la versión que construye o comprueba en la cabeza de la regla recursiva y la versión que construye o comprueba en el cuerpo de la regla. La primera solución es más elegante. La segunda solución, sin embargo, puede resultar más fácil para programadores de lenguajes procedurales, ya que la lista se construye "manualmente", desde la lista vacía. Es interesante conocer ambas formas de resolverlo para poder elegir la más conveniente a cada problema.

Ejercicio 2.5

Comprobar si una lista es un palíndromo.
(Un palíndromo es una palabra que se lee igual de izquierda a derecha que de derecha a izquierda)

Ejercicio 2.6

Escribir un conjunto de cláusulas que resuelvan las siguientes operaciones sobre conjuntos:

a) Subconjunto.
Ejemplo 1: ?- subconjunto([b,c],[a,c,d,b]). % El resultado sería verdadero

Ejemplo 2: ?- subconjunto([a,c,d,b],[b,c]). /* El resultado sería verdadero, pero se obtendría con un conjunto de cláusulas distintas a las de la solución para el ejemplo 1. Probad a escribir el conjunto de cláusulas necesarias para el caso del ejemplo 1 y para el caso del ejemplo 2. */

b) Disjuntos.
Ejemplo: ?- disjuntos([a,b,d],[m,n,o]). % El resultado sería verdadero

Ejercicio 2.7

Definir dos predicados:

par(Lista), impar(Lista)

que serán verdaderos si Lista tiene un número par e impar de elementos, respectivamente.

Ejercicio 2.8

Definir la relación

shift(L1,L2)

tal que L2 es la lista L1 después de una rotación de un elemento a la izquierda.
Ejemplo: ?- shift([a,b,c,d],L), shift(L,L1).
Da como resultado:

L=[b,c,d,a]
L1=[c,d,a,b]

Ejercicio 2.9

Definir la relación

trasladar(L1,L2)

que permita trasladar una lista de números (L1) a una lista de sus correspondientes nombres (L2).

Ejemplo: ?- trasladar([1,2,3,7],[uno,dos,tres,siete]). % El resultado sería veradero

Utiliza la siguiente relación auxiliar:

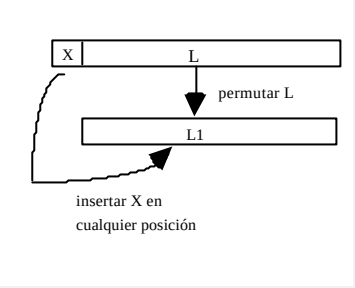
significa(0,cero).
significa(1,uno).
significa(2,dos).
etc.

Ejercicio 2.10

Calcular las permutaciones de una lista.
Ejemplo:

?- permutacion([a,b,c,],P).

P=[a,b,c];
P=[a,c,b];
P=[b,a,c];…



Pista: para insertar en cualquier posición se puede utilizar el predicado borrar.

Si queremos contar los elementos de una lista en PROLOG, necesitamos operadores para sumar. Los siguientes son nombres de predicados en PROLOG que son predefinidos (ya están definidos, sólo hay que utilizarlos) con un comportamiento distinto del explicado hasta ahora y que como efecto colateral nos permitirán sumar, etc.

2.3. Construcción de expresiones aritméticas en PROLOG

A continuación se muestran los operadores que se pueden utilizar para construir expresiones:

X+Y	/*suma*/
X-Y	/*resta*/
X*Y	/*multiplicación*/
X/Y ó X div Y	/*división real y entera*/
X mod Y	/*resto de la división de X entre Y*/
X^Y	/*X elevado a Y*/
-X	/*negación*/
abs(X)	/*valor absoluto de X*/
acos(X)	/*arco coseno de X*/
asen(X)	/*arco seno de X*/
atan(X)	/*arco tangente de X*/
cos(X)	/*coseno de X*/
exp(X)	/*exponencial de X; [e^X]*/
ln(X)	/*logaritmo neperiano de X*/
log(X)	/*logaritmo en base 2 de X*/
sin(X)	/*seno de X*/
sqrt(X)	/*raíz cuadrada de X*/
tan(X)	/*tangente de X*/
round(X,N)	/*redondeo del real X con N decimales*/

2.4. Comparación de términos en PROLOG.

Los siguientes operadores son los que permiten comparar términos (véase el apartado 1.5 para un repaso del concepto de término):

X<Y	/*X es menor que Y*/
X>Y	/*X es mayor que Y*/

```
X=<Y          /*X es menos o igual que Y*/
X>=Y          /*X es mayor o igual que Y*/
X=Y           /*X es igual que Y*/
X\=Y          /X es distinto que Y*/
```

2.5. Comparación de expresiones en PROLOG.

Una expresión es un conjunto de téminos unidos por operadores aritméticos (los introducidos en el apartado 2.3). Los siguientes predicados predefinidos comparan expresiones sin evaluarlas, mediante una comparación sintáctica siguiendo el siguiente orden:

- variables,
- enteros y reales,
- átomos en orden alfabético,
- términos complejos: aridad, nombre y orden según la definición recursiva.

```
X==Y          /*la expresión X es igual que la expresión Y*/
X\==Y         /*la expresión X es distinta que la expresión Y*/
X@<Y          /*la expresión X es menor que la expresión Y*/
X@>Y          /*la expresión X es mayor que la expresión Y*/
X@=<Y         /*la expresión X es menor o igual que la expresión Y*/
X@>=Y         /*la expresión X es mayor o igual que la expresión Y*/
```

El siguiente operador permite evaluar expresiones:

```
X is Y
```

Ejemplo 2.7

```
X is 3*5+3/2    /*PROLOG contestaría X=9*/
```

Para obtener un resultado equivalente a lo que en los lenguajes procedurales es la asignación, en PROLOG usaremos el predicado “is” para instanciar la variable que aparece a la izquierda a un valor concreto o al resultado de la evaluación de una expresión que aparece a la derecha. Sin embargo, esa asignación sólo podrá ser satisfecha si la variable que aparece a la izquierda del “is” no está instanciada, en caso contrario la operación fallará. Es decir, en PROLOG no se puede cambiar el valor de una variable sin desistanciarla mediante backtracking.

Los siguientes predicados predefinidos comparan términos haciendo una evaluación de expresiones:

```
X=:=Y         /* El resultado de evaluar la expresión X es igual al resultado
                 de evaluar la expresión Y*/
X=\=Y         /* El resultado de evaluar la expresión X es distinto al
                 resultado de evaluar la expresión Y*/
```

Ejemplo 2.8

Escribir un predicado PROLOG para determinar la longitud de una lista.

Ejemplo: `?-longitud([a,b,c,d,e],5).`
Solución:

```
longitud([],0).
longitud([_|Resto],N):-
     longitud(Resto,N1),
     N is N1+1.
```

Para la siguiente consulta al anterior predicado "?-longitud([e,w,f,v],X).", las variables N y N1 no tendrán ningún valor hasta que sean inicializadas en la salida de la recursividad, cuando la lista esté vacía. A partir de ese momento, se contabilizará la longitud de la lista en el cierre de la recursividad. Para una mejor comprensión del funcionamiento del PROLOG es aconsejable ejecutar los programas paso a paso, entendiendo cada uno de los pasos.

Ejercicio 2.11

a) Escribir un predicado PROLOG para que dada una lista de números, determine el número de elementos positivos (el 0 se considera como positivo) y el número de elementos negativos.

Ejemplo: ?-pos_y_neg([3,-1,9,0,-3,-5,-8],3,4). % PROLOG respondería sí.

b) Escribir un predicado PROLOG para que dada una lista de números, construya dos listas, una de números positivos y otra de negativos y que determine la cardinalidad de ambas listas.
Ejemplos:
?-pos_y_neg([3,-1,9,0,-3,-5,-8],3,[3,9,0],4,[-1,-3,-5,-8]). % PROLOG respondería sí.
?-pos_y_neg([3,-1,9,0,-3,-5,-8],Npos,Pos,Nneg,Neg). /* PROLOG respondería:
Npos=3, Pos=[3,9,0], Nneg=4, Neg=[-1,-3,-5,-8] */

Ejercicio 2.12

Dada una lista, escribir los predicados PROLOG necesarios para obtener dos sublistas, una que contenga los elementos de las posiciones pares y otra que contenga los elementos de las posiciones impares.
Ejemplo: ?-par_impar([a,b,c,d,e,f],[b,d,f],[a,c,e]). % PROLOG contestaría que sí.

Ejercicio 2.13

Dada una lista de números, escribir los predicados PROLOG necesarios para hallar el elemento mínimo y el máximo.
Ejemplo: ?-minimo([4,6,2,7,9,0],Min). % PROLOG respondería Min=0
Ejemplo: ?-maximo([4,6,2,7,9,0],Max). % PROLOG respondería Max=9

Ejercicio 2.14

Dada una lista, escribir los predicados PROLOG necesarios para contabilizar las apariciones de un determinado elemento en la lista.
Ejemplo: ?-aparicion(a,[a,b,c,a,b,a],X). % PROLOG contestaría X=3.

2.6. Bibliografía

[Clocksin 93] Clocksin, W.F., Mellish, C.S., Programación en PROLOG. Segunda edición. Colección Ciéncia Informática. Editorial Gustavo Gili S.A., 1993.

[Bratko 94] Bratko,I., PROLOG programming for Artificial Intelligence. Second edition. Addison-Wesley Publishing Company, 1994.

[Konigsberger & Bruyn 90] Konigsberger, H., Bruyn, F., PROLOG from the beginning. McGraw-Hill book Company, 1990.

[Starling & Shapiro 94] Starling, L., Shapiro, E., The art of PROLOG. Second edition. The MIT Press, 1994.

[Starling 90] Starling, L. editor, The practice of PROLOG. The MIT Press, 1990.

[O'Keefe 90] O'Keefe, R.A., The craft of PROLOG. The MIT Press, 1990.

[Stobo 89] Stobo, J., Problem solving with PROLOG. Pitman publishing, 1989.

3. El corte en PROLOG

CONTENIDO

3.1. Introducción

En este capítulo veremos un predicado predefinido para prevenir el backtracking: **el corte**. El corte también extiende el poder de expresividad de PROLOG y permite la definición de un tipo de negación: la negación como fallo y la asocia con "la asunción de un mundo cerrado".
PROLOG hace backtracking automáticamente, si es necesario para satisfacer un objetivo. El backtracking automático es un concepto de programación útil, porque alivia al programador de la carga de la programación con backtracking explícita. Por otro lado, el backtracking incontrolado puede causar ineficiencia en un programa. Además, algunas veces es preferible controlar el backtracking para no hacerlo. Podemos hacer esto utilizando **el corte**.
Veamos primero un ejemplo simple donde el backtracking es innecesario (e ineficiente).

Ejemplo 3.1

R1: si X<3 entonces Y=0
R2: si 3≤X y X<6 entonces Y=2
R3: si 6≤X entonces Y=4

Solución 1:

```
f(X,0):- X<3.
f(X,2):- 3=<X,X<6.
f(X,4):- 6=<X.
```

Problema: `?-f(1,Y), 2<Y.`

Este objetivo no se puede cumplir, porque para X=1, Y no puede ser >2. Sin embargo, con la solución 1 se necesita comprobar las 3 definiciones para demostrar que no se cumple.
Debemos intentar encontrar una solución para esta ineficiencia: **el corte**.

3.2. Comportamiento del corte en PROLOG

El corte es un predicado predefinido, cuya sintaxis es "!" y cuyo comportamiento es el siguiente. Suponemos definido el predicado "a" con las siguientes cláusulas:

```
a :- b, !, c.
a :- d.
```

Para comprobar o demostrar que el predicado *a* es cierto, pasamos a comprobar si es cierto el predicado *b*. Pueden ocurrir dos cosas:

1. Si *b* no es cierto, mediante backtracking, se comprobará si *d* es cierto. Si *d* es cierto, quedará demostrado que *a* es cierto. Hasta aquí el predicado a tiene un comportamiento normal en PROLOG.

2. Si *b* es cierto, se sobrepasa la "barrera" del corte que significa:
- Que no se podrá hacer backtracking para tomar otras posibilidades alternativas en el predicado *b*.
- Que no se pueden considerar otras posibles definiciones del predicado *a* (es decir, no podemos acceder a *a :- d.*).

Si *c* es cierto, entonces se ha demostrado *a*, pero si *c* no se cumple, entonces el predicado *a* falla.

Intentemos, utilizando el corte, ver si hay alguna solución buena para el problema anterior.

Solución 2: % En este caso el corte no modifica el significado declarativo

```
f(X,0):- X<3,!.
f(X,2):- 3=<X, X<6,!.
f(X,4):- 6=<X.
```

Solución 3: % Más eficiente, pero sí cambia el significado declarativo

```
f(X,0):- X<3,!.
f(X,2):- X<6,!.
f(X,4).          % Esta es la regla "recogetodo"
```

3.3. Usos comunes del corte

Hay 3 usos comunes del corte en PROLOG:
1. No buscar soluciones en predicados alternativos
2. Combinación de corte y fallo
3. Sólo es necesaria la primera solución

Veremos a continuación cada uno de ellos.

3.3.1. No buscar soluciones en predicados alternativos

Regla correcta para resolver un objetivo: "Si has llegado hasta aquí es que has escogido la regla adecuada para este objetivo, ya no hace falta que sigas comprobando las demás, no encontrarás otra solución".
Como se muestra en el ejemplo 3.1 y sus soluciones

Ejemplo 3.2
(max/3 nos indica en el tercer argumento el mayor de los dos primeros)

```
Solución 1:    max(X,Y,X):- X>=Y.
               max(X,Y,Y):- X<Y.
Solución 2:    max(X,Y,X):- X>=Y,!.      % Más eficiente.
               max(X,Y,Y).
```

3.3.2. Combinación de corte y fallo

Cuando queremos decirle a PROLOG que fracase inmediatamente, sin intentar encontrar soluciones alternativas. Se combina corte y fallo para conseguir la negación en PROLOG. La definición del predicado predefinido "not" es la siguiente:

```
not(P):- call(P), !, fail.
not(P).
```

Ejemplo 3.3
Para calcular la cantidad de impuestos que debe pagar una persona se necesita comprobar si dicha persona es un pagador de impuestos "medio", porque para este tipo de personas se utiliza un cálculo simplificado.

```
pagador_medio(X):-
     extranjero(X),
     !,fail.
pagador_medio(X):-
     conyuge(X,Y),
     ingresos_brutos(Y,Ing),
     Ing>2.000.000,
     !,fail.
pagador_medio(X):-
     ingresos_brutos(X,Ing),
     1.000.000<Ing,
     4.000.000>Ing,
     !,fail.
```

```
pagador_medio(X):-
     recibe_pension(X,P),
     P<800.000,
     !,fail.
ingresos_brutos(X,Y):-
     salario_bruto(X,Z),
     rentas(X,W),
     Y is Z+W.
...
```

Redefinición del programa "pagador_medio" usando not:

```
pagador_medio(X):-
     not(extranjero(X)),
     not((conyuge(X,Y), ingresos_brutos(Y,Ing), Ing>2.000.000)),
     not((ingresos_brutos(X,Ing1), 1.000.000<Ing1, 4.000.000>Ing1)),
     not(recibe_pension(X,P), P<800.000)).
```

Por cuestiones de legibilidad conviene utilizar la negación como fallo (not), en vez del corte-fallo. Siempre es posible sustituir el corte/fallo por el not.

3.3.3. Sólo es necesaria la primera solución

"Si has llegado hasta aquí es que has encontrado la única solución a este problema, no se necesita buscar alternativas".
Se usa para finalizar una secuencia de "generación y prueba".

Este tercer uso común del corte se diferencia del primero en que las distintas alternativas se obtienen haciendo uso del backtracking, ya que no hay mas que un predicado donde elegir.

Ejemplo 3.4
Juego de 3 en raya.

```
movimiento_forzoso(Tablero,Cas):-
     en_linea(Casillas),
     amenaza(Casillas,Tablero,Cas), !.

    enlinea([1,2,3]).
    enlinea([4,5,6]).
    enlinea([7,8,9]).
    enlinea([1,4,7]).
    enlinea([2,5,8]).
    enlinea([3,6,9]).
    enlinea([1,5,9]).
    enlinea([3,5,7]).

    amenaza([X,Y,Z],B,X):-
      vacio(X,B), cruz(Y,B), cruz(Z,B).
    amenaza([X,Y,Z],B,Y):-
      vacio(Y,B), cruz(X,B), cruz(Z,B).
    amenaza([X,Y,Z],B,Z):-
    vacio(Z,B), cruz(X,B), cruz(Y,B).

    vacio(Cas,Tablero):- arg(Cas,Tablero,Val), Val=v.
    cruz(Cas,Tablero):- arg(Cas,Tablero,Val), Val=x.
    cara(Cas,Tablero):- arg(Cas,Tablero,Val), Val=o.

    arg(Num,Lista,Valor):- arg(Num,1,Lista,Valor).

    arg(N,N,[X|_],X).
    arg(N,N1,[_|L],Valor):-
        N2 is N1 + 1,
        arg(N,N2,L,Valor).
```

1	2	3
4	5	6
7	8	9

Esta puede ser un ejemplo de disposición de fichas en el tablero:

Tablero=[x,v,o,v,v,v,x,o,x]

x		o
x	o	x

3.4. Problemas con el corte

Los cortes que no afectan al significado declarativo de los programas se llaman cortes "verdes". Esos programas se pueden leer sin tener en cuenta los cortes. Los cortes que sí cambian el significado declarativo de los programas, los cortes llamados "rojos", pueden ocasionar problemas. Es más fácil cometer errores con ellos. Cuando cambiamos el uso de un programa con uno de esos cortes debemos repasar su significado procedural.

Ejemplo 3.5

```
p:- a,b.            % p ⇔ (a∧b) ∨ c
p:- c.
```

/* El siguiente corte es "rojo", si invierto el orden de la cláusula cambia el significado declarativo */

```
p:- a, !, b.        % p ⇔ (a∧b) ∨ (¬a∧c)
p:- c.

p:-c.               % p ⇔ c ∨ (a∧b)
p:-a, !, b.
```

Los siguientes ejemplos no llevan cortes "rojos", pero sí ocasionan problemas con algunos usos concretos de las relaciones.

Ejemplo 3.6
Si se define "concatenar" haciendo uso del corte, para mejorar la eficiencia en su ejecución:

```
concatenar([],X,X):- !.
concatenar([A|B],C,[A|D]):- concatenar(B,C,D).
```

Al intentar buscar todas las posibles divisiones de una lista en dos sublistas (un uso posible del predicado "concatenar"), sólo podemos obtener una solución, ya que se satisface el predicado "!" de la primera definición de "concatenar" y no hay posibilidad de acceder a la segunda definición de "concatenar", como vemos a continuación.

```
?- concatenar(X,Y,[a,b,c]).
X=[],
Y=[a,b,c].          % Y no da más soluciones
```

3.5. La negación y sus problemas.

Dada la definición de los siguientes predicados:

```
bueno(juanluis).
bueno(francisco).

caro(juanluis).

razonable(Restaurante):-
not caro(Restaurante).   % El corte está implícito en el not
```

La siguiente conjunción de objetivos daría una solución:

```
?-bueno(X), razonable(X).
X=francisco.
```

Ya que el predicado “bueno” hace instanciar X a “juanluis” (que no cumple el segundo predicado “razonable”) y después a “francisco” (que sí cumple el segundo predicado “razonable”).
Pero si intercambiamos el orden de los objetivos a comprobar, “razonable” no permite instanciar ninguna variable sin que el objetivo completo falle, por lo que no podrá pasar al segundo objetivo “bueno”, y nunca obtendremos solución.

```
?-razonable(X), bueno(X).
no.
```

La negación utilizada en PROLOG no corresponde a la negación de la lógica clásica, la comunmente conocida en matemáticas. Por ejemplo a la pregunta:

```
?- not caro(francisco).
```

PROLOG contestaría sí, lo cual no quiere decir necesariamente que el restaurante francisco no sea caro, sino que no se ha podido comprobar que sea caro. Esto es debido a que PROLOG no intenta probar el objetivo “not caro(francisco).” directamente, sino que prueba “caro(francisco).” y si no se satisface, entonces asume que “not caro(francisco).” se satisface. Este razonamiento sería válido si asumimos lo que se llama “mundo cerrado”, es decir, lo que es cierto es todo aquello que está en el programa PROLOG o puede derivarse de él, y todo lo que no está se entiende como falso. Sin embargo, esta asunción no es generalmente cierta en el sentido de que lo que no está en el programa PROLOG ni se puede derivar de él, no necesariamente es falso, simplemente no se tiene constancia de ello.
Este uso de la negación en PROLOG causa una no monotonía en el sentido de que lo que antes se puede derivar como verdadero, tras añadir nueva información pasa a ser considerado como falso.
Por ejemplo, “?- not caro(francisco).” era verdad en el ejemplo anterior. Pero si se añade como hecho “caro(francisco).” pasa a ser falso.

3.6. Ejercicios

Ejercicio 3.1

Escribir un conjunto de cláusulas PROLOG que resuelvan las siguientes operaciones sobre conjuntos:

a) Intersección.
Ejemplo: ?- interseccion([a,b,c,d],[c,e,f,b],[b,c]). % PROLOG contestaría que sí.

b) Unión.
Ejemplo: ?- union([a,b,c,d],[d,e,f],[a,b,c,d,e,f]). % PROLOG contestaría que sí.

c) Diferencia de conjuntos. Dadas dos listas, L1 y L2, obtener otra lista cuyos elementos son los elementos de L1 que no están incluídos en L2.
Ejemplo: ?-diferencia([a,b,c,d,e,f],[b,d,e],[a,c,f]). % PROLOG contestaría que sí.

Ejercicio 3.2

Definir la relación

aplanar(L, Lplana)

en donde L, que puede ser una lista de listas, pase a ser el conjunto de todos los elementos de L pero en una lista plana en Lplana.

Ejemplo: ?- aplanar([a,b,[c,d],[],[[[e]]],f],L).
L=[a,b,c,d,e,f]

Ejercicio 3.3

Cambiar una frase por otra, según el siguiente ejemplo:

Usuario: tu eres un ordenador
PROLOG: yo no soy un ordenador

Usuario: hablas frances
PROLOG: no- hablo aleman

Para ello definimos la siguiente relación auxiliar en una base de datos:

```
cambiar(tu,yo).
cambiar(eres,[no,soy]).
cambiar(hablas,[no-,hablo]).
cambiar(frances,aleman).
cambiar(X,X).
```

/* Esta última se llama “recogetodo”, si una palabra no aparece como que tiene que cambiarse, se cambiará por ella misma */

Escribir el programa PROLOG para que se cambien frases enteras del estilo de los ejemplos anteriores.

Ejercicio 3.4

La definición de miembro de un elemento en una lista encuentra todas las ocurrencias del elemento en la lista, pulsando sucesivamente ";". Permitir que sólo encuentre la primera.

Ejercicio 3.5

Añadir un elemento a una lista sin admitir duplicados.

Ejercicio 3.6

Sea el programa:

```
p(1).
p(2):- !.
p(3).
```

Escribir las respuestas de PROLOG a las siguientes preguntas:

a) ?- p(X).
b) ?- p(X), p(Y).
c) ?- p(X), !, p(Y).

Ejercicio 3.7

La siguiente relación clasifica los números en 3 clases: positivos, cero y negativos. Define una relación equivalente que sea más eficiente.

```
clase(Numero,positivo):- Numero>0.
clase(0,cero).
clase(Numero,negativo):- Numero<0.
```

Ejercicio 3.8

Define la relación

divide(Numeros,Positivos,Negativos)

que divida la lista de Numeros en 2 listas, una de números Positivos y otra de Negativos, usando el operador corte. Compara la solución con el ejercicio 2.11.
Ejemplo: ?- divide([3,-1,0,5,-2],[3,0,5],[-1,-2]).

Ejercicio 3.9

Se desea evaluar expresiones lógicas dadas por listas PROLOG que contengan los conectores lógicos not y and. Por ejemplo, la evaluación de la expresión lógica definida por esta lista:

[not,not,not,0,and,1,and,1,and,not,0]

Daría como resultado 1.
Suponiendo que las expresiones a evaluar son siempre correctas y que la evaluación se hace de izquierda a derecha, se pide:

a) Implementar en PROLOG las tablas de verdad de los conectores lógicos not y and .

b) Implementar en PROLOG los predicados necesarios para que, dada una lista que empiece por un conector lógico not, evalúe la sublista que acaba con el primer valor lógico (0 ó 1) que se encuentre en la lista después de todos los not. En el ejemplo anterior evaluaría not,not,not,0, dando como resultado 1. Ese valor se quiere devolver como sustitución de la sublista en la lista original. Es decir, devolvería la lista:

[1,and,1,and,1,and,not,0]

c) Implementar en PROLOG los predicados necesarios para que, dada una lista que contiene una expresión lógica en la que sólo aparecen el conector lógico and (por ejemplo [1,and,1,and,0,and,1]) devuelva el valor de su evaluación (en este caso devolvería 0).

NOTA: Faltaría un cuarto apartado para solucionar completamente el problema planteado. Como ejercicio se podría intentar resolver el problema completo.

Ejercicio 3.10

Dada una lista de letras del alfabeto, por ejemplo: L=[a,b,c,d,e,f,g], se pide:

a) Implementar en PROLOG los predicados necesarios para que, dado un elemento de una lista y la lista, devuelva una sublista que tendrá los elementos de la lista original desde el elemento indicado (inclusive) hasta el final de la lista.

Ejemplo:

La llamada al predicado	?-desde(c,L,L1).
Dará como resultado:	L1=[c,d,e,f,g].

b) Implementar en PROLOG los predicados necesarios para que, dado un elemento de una lista y la lista, devuelva una sublista que tendrá los elementos de la lista original desde el primer elemento de la lista hasta el elemento indicado (inclusive).

Ejemplo:

La llamada al predicado	?-hasta(d,L,L1).
Dará como resultado:	L1=[a,b,c,d].

c) Utilizando los predicados "desde" y "hasta", implementar en PROLOG los predicados necesarios para que, dados dos elementos de una lista y la lista devuelva una sublista que tendrá los elementos de la lista original desde el primer elemento indicado hasta el segundo elemento indicado (ambos inclusive).

Ejemplo:

La llamada al predicado	?-desde_hasta(c,d,L,L1).
Dará como resultado:	L1=[c,d].

Ejercicio 3.11

Dadas dos listas de números ordenadas en orden creciente, se pide implementar en PROLOG los predicados necesarios para calcular la unión ordenada de las dos listas iniciales. La solución debe basarse en la característica de que las listas originales están ordenadas.

Ejemplo:

Dadas las listas:	L1=[1,5,7,9], L2=[4,5,6,9,10].
La llamada al predicado	?-union_ordenada(L1,L2,L3).
Dará como resultado:	L3=[1,4,5,6,7,9,10].

Ejercicio 3.12

Dada una lista cuyo contenido es un conjunto de nombres (p.e. [q0,q1,q2,q3,q4]) y un par de esos nombres (p.e. q1 y q3), escribir un predicado en PROLOG que permita obtener una sublista cuyos nombres serán los nombres de la lista original que van del primer nombre hasta el segundo (el resultado del ejemplo anterior sería [q1,q2,q3]).
Es decir, la llamada al predicado:

```
?- sublista(q1,q3,[q0,q1,q2,q3,q4],L).
```

Dará como resultado:

```
L=[q1,q2,q3]
```

Nota: Los dos nombres que se dan como entrada siempre pertenecen a la lista original y además el primer nombre siempre está antes que el segundo nombre en esa lista.

3.7. Bibliografía

[Clocksin 93] Clocksin, W.F., Mellish, C.S., Programación en PROLOG. Segunda edición. Colección Ciéncia Informática. Editorial Gustavo Gili S.A., 1993.

[Bratko 94] Bratko,I., PROLOG programming for Artificial Intelligence. Second edition. Addison-Wesley Publishing Company, 1994.

[Konigsberger & Bruyn 90] Konigsberger, H., Bruyn, F., PROLOG from the beginning. McGraw-Hill book Company, 1990.

[Starling & Shapiro 94] Starling, L., Shapiro, E., The art of PROLOG. Second edition. The MIT Press, 1994.

[Starling 90] Starling, L. editor, The practice of PROLOG. The MIT Press, 1990.

[O'Keefe 90] O'Keefe, R.A., The craft of PROLOG. The MIT Press, 1990.

[Stobo 89] Stobo, J., Problem solving with PROLOG. Pitman publishing, 1989.

4. Predicados predefinidos

CONTENIDO

4.1. El esquema condicional en PROLOG.

En PROLOG la "Y" lógica corresponde a la "," y la "O" lógica a la definición de varias cláusulas para el mismo predicado, como ya hemos visto en los anteriores capítulos. Equivalente a la definición de varias cláusulas para el mismo predicado es el uso del predicado predefinido ";", tal y como vemos en los siguientes ejemplos.

```
p :- a, b.
p :- c, d.
p :- e, f.
p :- g.
```

```
p :-
    (
    a, b
    ;
    c, d
    ;
    e, f
    ;
    g
    ).
```

Ambas definiciones son equivalentes. Para facilitar la lectura de programas PROLOG se recomienda el uso de la construcción de la izquierda, es decir, varias cláusulas para el mismo predicado. Si por cualquier motivo se usa la construcción de la derecha, es recomendable la separación de los paréntesis "(", ")" y el punto y coma";" en líneas distintas, para así distinguirlo rápidamente de la coma ",".

4.2. La notación operador.

A veces puede resultar interesante escribir predicados o funtores como operadores. Por ejemplo, las operaciones aritméticas suelen escribirse normalmente como operadores, por ejemplo de la forma "x+y×z" donde "x", "y" y "z" son argumentos y "+" y "×" son operadores.
Sin embargo, PROLOG utiliza por defecto notación prefija"+(x, × (y,z))" (que internamente corresponde a una representación en árbol.
PROLOG permite definir representaciones infijas utilizando la notación operador.
Para definir operadores necesitaremos determinar su ***precedencia*** (que establece un orden de evaluación del operador), su ***posición*** (infijo, prefijo o postfijo) y su ***asociatividad*** (cuando se combinan operadores de la misma precedencia).
Un programador puede definir sus propios operadores. Por ejemplo, podemos definir los átomos es y contiene como operadores infijos y escribir hechos de la siguiente manera:

```
juan es listo.
el_jarrón contiene flores.
```

Y esos hechos serán equivalentes a:

```
es (juan ,listo).
contiene (el_jarrón, flores).
```

Por ejemplo, el operador ":-" se define como infijo con la máxima precedencia:

```
:-op(1200,xfx,':-').
```

La ***precedencia*** va de 1 a 1200. La máxima precedencia corresponde a los últimos operadores que se evaluarán, es decir, los más "débiles".
La ***posición*** vendrá determinada en el segundo argumento por:

- Infijos: xfx, xfy, yfx
- Prefijos: fx, fy
- Postfijos: xf, yf

donde *x* e *y* se refieren a los argumentos y *f* al funtor o nombre de predicado declarado como operador.

La ***asociatividad*** viene representada por los caracteres *x* e *y*. *x* se refiere a un argumento cuya precedencia es estrictamente menor que la precedencia del operador e *y* se refiere a un argumento cuya precedencia es menor o igual que la precedencia del operador.

Por ejemplo, si la definición del operador not es:

```
:-op(500,fx,not).
```

Entonces no podremos utilizar **not not p**, porque el argumento de not es de la misma precedencia que not y no estrictamente menor. Tendremos que hacer uso de paréntesis, **not(not p)** o definir el operador not como :-op(500,fy,not).
En la tabla 4.1 puede verse algunos ejemplos de operadores predefinidos en ECLiPSe y SICStus:

en ECLiPSe	en SICStus

`:-op(1200,xfx,':-').`
`:-op(1200,fx,[:-,?]).`
	`:-op(1200,xfx,[:-, -->]).`
	`:-op(1200,fx,[:-, ?-]).`
	`:-op(1150,fx,[mode, public, dynamic,`
	`multifile, block,`
	`meta_predicate, parallel,`
	`sequential]).`

`:-op(1100,xfy,';').`	`:-op(1100,xfy,[;]).` `:-op(1050,xfy,[->]).`
`:-op(1000,xfy,',').`	`:-op(1000,xfy,[,]).` `:-op(900,fy,[\+, spy, nospy]).`

`:-op(700,xfx,[=,is,<,>,=<,>=`
`,==,=\=,\==,=:=]).`
	`:-op(700,xfx,[=,is,=..,\==,<,>,=<,`
	`>=,@<,@>,@=<,@>=,`
	`==,=\=,\==,=:=]).`

`:-op(500,yfx,[+,-]).`
`:-op(500,fx,[+,-,not]).`
`:-op(400,yfx,[*,/,div]).`
`:-op(300,xfx,mod).`
	`:-op(550,xfy,[:]).`
	`:-op(500,yfx,[+,-,#,/\,\/]).`
	`:-op(500,fx,[+,-]).`
	`:-op(400,yfx,[*,/,//,<<,>>]).`
	`:-op(300,xfx,[mod]).`
	`:-op(200,xfy,[^]).`

Tabla 4.1 Operadores predefinidos en ECLiPSe y SICStus.

4.3. Predicados predefinidos

Los predicados predefinidos son aquellos que ya están definidos en PROLOG, que no necesitamos especificarlos mediante cláusulas. Aunque algunos predicados predefinidos ya han sido introducidos en los capítulos anteriores, daremos un repaso a los más importantes en este capítulo. Existen básicamente dos tipos de predicados predefinidos[1]:

- Aquellos predicados de uso frecuente que ya los proporciona PROLOG, aunque podríamos definirlos nosotros.
- Predicados con un efecto colateral distinto a la instanciación de variables a valores (funcionamiento normal del PROLOG).

A continuación se da una lista de aquellos predicados predefinidos que son más utilizados, clasificados según su función.

4.3.1. Clasificación de términos

[1] Los predicados se suelen escribir de dos formas, o bien seguidos de tantas variables entre paréntesis como argumentos tengan (por ejemplo, `predicado(X,Y,Z)`), o bien con el nombre del predicado seguido de una barra invertida "/" y del número de argumentos que utiliza (por ejemplo, `predicado/3`), que es la aridad.

Este conjunto de predicados predefinidos permiten determinar el tipo de términos que estamos usando.

`var/1`
El objetivo `var(X)` se cumple si `X` es una variable no instanciada.

`novar/1`
El objetivo `novar(X)` se cumple si `X` es una variable instanciada.

`atom/1`
El objetivo `atom(X)` se cumple si `X` representa un átomo PROLOG.

`integer/1`
El objetivo `integer(X)` se cumple si `X` representa un número entero.

`atomic/1`
El objetivo `atomic(X)` se cumple si `X` representa un entero o un átomo.

4.3.2. Control de otros predicados

Los siguientes son predicados predefinidos que permiten controlar otros predicados.

`!/0 (cut)`
El símbolo de "`corte`" es un predicado predefinido que fuerza al sistema PROLOG a mantener ciertas elecciones que ha realizado.

`true/0`
Este objetivo siempre se cumple.

`fail/0`
Este objetivo siempre fracasa.

`not/1`
Suponiendo que `X` está instanciada a un término que pueda interpretarse como un objetivo. El objetivo `not(X)` se cumple si fracasa el intento de satisfacer `X`. El objetivo `not(X)` fracasa si el intento de satisfacer `X` tiene éxito.

`repeat/0`
El predicado predefinido `repeat` se da como una forma auxiliar para generar soluciones múltiples mediante el mecanismo de reevaluación.

`call/1`
Suponiendo que `X` está instanciada a un término que pueda interpretarse como un objetivo. El objetivo `call(X)` se cumple si tiene éxito el intento de satisfacer `X`.

`,/2`

El funtor "`,`" específica una conjunción de objetivos.

`;/2`
El funtor "`;`" específica una disyunción (es decir, una *o* lógica) de objetivos.

4.3.3. Introducción de nuevas cláusulas

`consult/1, reconsult/1` y lo que es equivalente en ECLiPSe, la compilación usando ***[nombrefichero]***.

`consult/1`
El predicado predefinido `consult` está pensado para situaciones en las que se quiera añadir las cláusulas existentes en un determinado fichero (o que se tecleen en el terminal) a las que ya están almacenadas en la base de datos. El argumento debe ser un átomo que dé el nombre del fichero del que se van a leer las cláusulas.

`reconsult/1`
El predicado `reconsult` es similar a `consult`, excepto que las cláusulas leídas sustituyen a todas las demás cláusulas existentes para el mismo predicado. (Muy útil para corregir errores de programación)

4.3.4. Construcción y acceso a componentes de estructuras

`arg/3`
El predicado `arg` debe utilizarse siempre con sus dos primeros argumentos instanciados. Se usa para acceder a un determinado argumento de una estructura. El primer argumento de `arg` especifica qué argumento se requiere. El segundo especifica la estructura donde debe buscarse el argumento. PROLOG encuentra el argumento apropiado y entonces intenta hacerlo corresponder con el tercer argumento. Es decir, `arg(N, E, A)` se cumple siel argumento número `N` de la estructura `E` es `A`.

`functor/3`
El predicado `functor` se define de tal manera que `functor(E, F, N)` significa que `E` es una estructura con funtor o nombre de predicado `F` y aridad `N`.

`name/2`
El predicado `name` se usa para manejar átomos arbitrarios. El predicado `name` relaciona un átomo con la lista de caracteres (códigos ASCII) que lo constituyen. El objetivo `name(A,L)` significa que *los caracteres del átomo* `A` *son los miembros de la lista* `L`.

`=../2`

El predicado "`=..`" (pronunciado "`univ`" por razones históricas) se utiliza para construir una estructura, dada una lista de argumentos. El objetivo `X=..L` significa que `L` *es la lista que consiste en el funtor* `X` *seguido de los argumentos de* `X`.
Ej. append([a],[b],[a,b])=..L, L=[append,[a],[b],[a,b]]

4.3.5. Supervisión de PROLOG en su funcionamiento

`trace/0`
El efecto de satisfacer el objetivo `trace` es activar un seguimiento exhaustivo. Esto significa que, a continuación, podrá verse cada uno de los objetivos generados por el programa en cada uno de los cuatro puertos principales.

`notrace/0`
El efecto de satisfacer el objetivo `notrace` es desactivar un seguimiento exhaustivo. Sin embargo, los seguimientos debidos a la presencia de puntos espía continuarán.

`spy/1`
El predicado `spy` se utiliza cuando se quiere prestar especial atención a objetivos que incluyan ciertos predicados específicos. Esto se hace fijando en ellos *puntos espía*. El predicado se define como operador prefijo, de forma que no hace falta poner el argumento entre paréntesis. El argumento puede ser: un átomo, una estructura de la forma `nombre/aridad` o una lista.

`debugging/0`
El predicado predefinido `debugging` permite ver qué puntos espía se han establecido hasta el momento. La lista de puntos espía se escribe como efecto colateral al satisfacerse el objetivo `debugging`.

`nodebug/0`
El objetivo `nodebug` retira todos los puntos espía activos en ese momento.

`nospy/1`
Como `spy`, `nospy` es un operador prefijo. `nospy` es más selectivo que `nodebug`, ya que puede especificar exactamente qué puntos espía se quieren retirar.

4.3.6. Lectura/escritura y manejo de ficheros

4.3.6.1. Lectura/escritura de términos

`write/1`
El objetivo `write(X)` escribe el término `X` en el canal de salida activo. `write` sólo se cumple una vez (no se puede resatisfacer).

`nl/0`

Escribe una secuencia de control al canal de salida activo que genera una "nueva línea". `nl` sólo se cumple una vez.

`read/1`

El objetivo `read(X)` lee el siguiente término del canal de entrada activo, y lo hace coincidir con `X`. `read` sólo se cumple una vez. La entrada proporcionada desde teclado debe terminar con ".".

`display/1`

El predicado `display` funciona exactamente igual que `write`, excepto que pasa por alto las declaraciones de operadores.

Ejemplo 4.1

El siguiente ejemplo escribe términos identados según están incluidos en sublistas.

```
pp([H|T],I):- !, J is I+3, pp(H,J), ppx(T,J), nl.
pp(X,I):- tab(I), write(X), nl./* para presentar algo que no es una lista */

ppx([],_).
ppx([H|T],I):- pp(H,I), ppx(T,I).

tab(0):- !.
tab(N):- put(32), M is N-1, tab(M).% put/1 se explicará más adelante
```

Ejemplo 4.2

El siguiente ejemplo escribe términos en la misma línea.

```
pth([]):- nl.
pth([H|T]):- write(H), tab(1), pth(T).
```

Ejemplo 4.3

Este ejemplo muestra la diferencia entre write y display.

```
?- write(1+2*4), nl, display(1+2*4), nl.
1+2*4
+(1,*(2,4))
yes
```

Ejemplo 4.4

El siguiente ejemplo solicita al usuario una fecha y muestra el hecho histórico ocurrido en esa fecha.

```
acontecimiento(1505,['Euclides',traducido,al,latin]).
acontecimiento(1523,['Chistian','II',huye,de,'Dinamarca']).

/* ponemos comillas simples porque las constantes empiezan con
mayúscula o cuando hay símbolos que no pueden tener normalmente*/
```

```
consulta:-
pth(['Que',fecha,desea,'consultar?   ']),
read(D),
acontecimiento(D,S),
pth(S).
```

4.3.6.2. Lectura/escritura de caracteres

`put/1`
Este objetivo escribe el entero `X` como carácter en el canal de salida activo. `put` sólo se cumple una vez. Si `X` no está instanciada, se produce un error.

`get/1`
Este objetivo se cumple si `X` puede hacerse corresponder con el siguiente carácter imprimible encontrado en el canal de entrada activo. `get` sólo se cumple una vez.

`get0/1`
Este objetivo se cumple si `X` puede hacerse corresponder con el siguiente carácter encontrado en el canal de entrada activo. `get0` sólo se cumple una vez.

Ejemplo 4.5

```
escribecadena(L):-name(L,Lascii), escribecadena1(Lascii).
escribecadena1([]).
escribecadena1([H|T]):- put(H), escribecadena1(T).

?-escribecadena("Escribecadena permite presentar listas en forma
de cadena").
```

Escribecadena permite presentar listas en forma de cadena.

4.3.6.3. Lectura /escritura en ficheros

`tell/1`
El objetivo `tell(X)` abre el fichero `X` si no estaba abierto, y define el fichero `X` como canal de salida activo. Si `X` no está instanciada se produce un error. La primera vez que se utiliza `tell` sobre un fichero no existente, se crea un fichero con ese nombre. Si se utiliza `tell` con un fichero ya existente, el contenido de dicho fichero se destruye.

`telling/1`
Este objetivo se cumple si `X` coincide con el nombre del canal de salida activo, fracasando en caso contrario.

`told/1`
Este objetivo cierra el canal de salida activo, haciendo que se escriba una marca de fin-de-fichero al final del fichero. El nuevo canal de salida activo pasa a ser la pantalla del ordenador del usuario.

`see/1`
El objetivo `see(X)` abre el fichero `X`, si todavía no está abierto, y define al canal de entrada activo de tal forma que su origen sea el fichero `X`. Si `X` no está instanciada, o su nombre corresponde a un fichero que no existe, se produce un error.

`seeing/1`
Este objetivo se cumple si el nombre del canal de entrada activo coincide con `X`, fracasando en caso contrario.

`seen/1`
Este objetivo cierra el canal de entrada activo, asignando el nuevo canal de entrada activo al teclado del ordenador del usuario.

4.3.7. Manipulación de Bases de Datos

4.3.7.1. Añadir cláusulas a la BD.

`assert/1`
Añade cláusulas al final del conjunto de cláusulas en la BD que tienen el mismo nombre de predicado.

`asserta/1`
Añade cláusula al principio del conjunto de cláusulas que tienen el mismo nombre de predicado.

4.3.7.2. Borrar cláusulas de la BD.

`retract/1`
Borra una cláusula de la BD. Haciendo backtracking podremos borrar todas las cláusulas y con cada una de ellas efectuar las operaciones que nos interesen.

`retract_all/1`
Borra todas las cláusulas de la BD con el nombre y aridad que aparece en el argumento.

Estas formas de añadir y borrar son lógicas, es decir, no se almacenan físicamente en ninguna base de datos.

4.3.7.3. Visualizar cláusulas de la BD.

`findall/3`

Busca en la base de datos todas las ocurrencias de término (primer argumento) que satisfacen el objetivo (incluido en el segundo argumento) y devuelve todas las instancias en una lista (tercer argumento) desordenada.

Ejemplo 4.6. Este ejemplo contiene los cláusulas PROLOG correspondientes al predicado predefinido findall (Buscatodos).

```
buscatodos(X,G,_):-
     asserta(encontrada(marca)),
     call(G),
     asserta(encontrada(X)),
     fail.
buscatodos(_,_,L):-
     recoger_encontrados([],M),
     !,
     L=M.

recoger_encontrados(S,L):-
     siguiente(X),
     !,
     recoger_encontrados([X|S],L).
recoger_encontrados(L,L).

siguiente(X):-
     retract(encontrada(X)),
     !,
     X\==marca.
```

current_predicate/1

Tiene éxito si el argumento es un predicado definido visible por el usuario o desde una biblioteca.

Ejemplo 4.7

```
?-current_predicate(X).
```

A través de backtracking sacará todos los predicados que hay en la BD.

```
?-current_predicate(append/X).
```

Podemos comprobar la aridad del predicado que tenemos almacenado en la BD.

listing
listing(Nombre/Aridad)
listing([Nombre/Aridad,Nombre/Aridad,...])

Si no se especifica el predicado listing sacará la lista de todos los predicados de la BD actual.

4.4. Otros predicados predefinidos

`atom_length(+Atom,?Length)`
Se satisface si length es la longitud del átomo Atom.

`concat_atom(+List,?Dest)`
Se satisface si Dest es la concatenación de los términos atómicos que aparecen en la lista List.

`concat_atoms(+Src1,+Src2,?Dest)`
Se satisfae si Dest es la concatenación de Src1 y Src2.

`Substring(+String1,+String2,?Posicion)`
Se satisface si String2 es una sublista de String1 que comienza en la posición Posicion.

`append(?Lista1,?Lista2,?Lista3)`
Se satisface si Lista3 es el resultado de añadir Lista2 a Lista1.

`checklist(+Pred,+Lista)`
Se satisface si Pred(Elem) es verdad para cada elemento de la lista Lista.

`delete(?Elemento,?Lista1,?Lista2)`
Se satisface si Lista2 es Lista1 menos una ocurrencia de Elemento en Lista1.

`intersection(+Lista1,+Lista2,?Comun)`
Se satisface si Comun se unifica con la lista que contiene los elementos en comun de las listas Lista1 y Lista2.

`length(?List,?N)`
Se satisface si la longitud de la lista Lista es N.

`member(?Term,?Lista)`
Se satisface si el término Term unifica con un miembro de la lista Lista.

`nonmember(+Elem,+Lista)`
Se satisface si Elem no es un elemento de la lista Lista.

`subset(?Sublista,+Lista)`
Se satisface si Lista contiene todos los elementos de Sublista en el mismo orden que en Sublista.

`substract(+Lista1,+Lista2,?Resto)`
Se satisface si Resto contiene aquellos elementos de Lista1 que no están en Lista2.

`union(+Lista1,+Lista2,?Union)`
Se satisface si Union es la lista que contiene la unión de los elementos de Lista1 y de Lista2.

Ejercicio 4.1

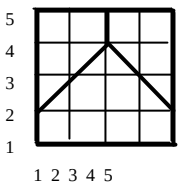
Escribir los predicados PROLOG que solucionen los predicados predefinidos citados en este párrafo (aunque no es necesario reescribirlos —son predicados predefinidos, sólo se necesita llamarlos— es recomendable el ejercicio de pensar cómo están escritos en PROLOG).

4.5. Ejemplo de uso de predicados predefinidos

Como ejemplo de uso de predicados predefinidos veremos la manipulación de árboles y grafos. La representación y acceso a árboles y grafos es muy utilizada en informática, y su manejo es muy simple y elegante en PROLOG. Básicamente existen dos estrategias de búsqueda de infromación en grafos: (1) primero en profundidad y (2) primero en anchura.

El siguiente es un ejemplo de cómo se puede resolver un problema representándolo como un grafo. Podemos representar un pequeño conjunto de calles con una cuadrícula superpuesta en ellas como un grafo, y el grafo representado como el siguiente conjunto de hechos en una base de datos.

```
Calle(1,1,2,1).     calle(4,5,5,5).
calle(1,1,1,2).     calle(2,1,3,1).
calle(1,2,1,3).     calle(3,1,4,1).
calle(1,2,2,3).     calle(4,1,5,1).
calle(1,3,1,4).     calle(5,1,5,2).
calle(2,3,3,4).     calle(5,2,5,3).
calle(1,4,1,5).     calle(5,3,5,4).
calle(1,5,2,5).     calle(5,4,5,5).
calle(2,5,3,5).     calle(5,2,4,3).
calle(3,4,3,5).     calle(4,3,3,4).
calle(3,5,4,5).
```



El significado es el siguiente: desde la esquina (1,1) se puede ir a la esquina (2,1) y la esquina (1,2), por eso hay dos hechos en la base de datos calle(1,1,2,1) y calle(1,1,1,2). Suponemos que los arcos no representan dirección.

Para hacer una búsqueda “primero en profundidad” desde el nodo (Sx,Sy) (esquina origen) al (Dx,Dy) (esquina destino) el programa sería el siguiente:

```
ir([Salidax,Saliday],[Destinox,Destinoy],Ruta):-
     ir0([Salidax,Saliday],[Destinox,Destinoy],[],R),
     inv(R,Ruta).

ir0([Dx,Dy],[Dx,Dy],T,[[Dx,Dy]|T]).
ir0([Sx,Sy],[Dx,Dy],T,R):-
     nodolegal([Sx,Sy],T,[Sigx,Sigy]),
     ir0([Sigx,Sigy],[Dx,Dy],[[Sx,Sy]|T],R).

nodolegal([Sx,Sy],Camino,[Sigx,Sigy]):-
          (
```

```
calle(Sx,Sy,Sigx,Sigy)
;
calle(Sigx,Sigy,Sx,Sy)
),
not member([Sigx,Sigy],Camino).
```

La lista T se construye para comprobar que no pasamos dos veces por el mismo sitio para evitar meternos en un bucle infinito.

El predicado “ir/3” encuentra una ruta, que no tiene que ser la mejor, y puede encontrar rutas alternativas mediante reevaluación.

Ejercicio 4.2

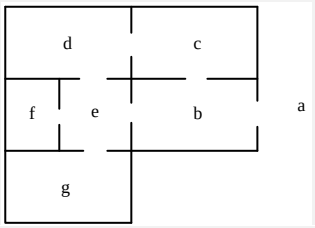
Escribir los predicados PROLOG necesarios para hacer la búsqueda primero en anchura para el ejemplo anterior.

Ejercicio 4.3

Escribir un programa en PROLOG que lea una base de datos donde se describan las calles, como en el problema ejemplo 2), y que utilice esa información para obtener la lista de esquinas por las que se tiene que pasar para ir desde una posición inicial [Xinicio,Yinicio] hasta cualquier punto de un área rectangular definida por 4 puntos [Xmin,Ymin,Xmax,Ymax].
Se han de respetar las calles y además se ha de obtener “el camino más corto”. El camino más corto se aproximará de la siguiente forma: cuando en una esquina haya una bifurcación de caminos, se elegirá como esquina siguiente dentro del camino más corto aquella cuya distancia a los vértices del rectángulo destino sea menor.

Ejercicio 4.4

Se quiere buscar, desde la calle, en qué habitación del palacio de la figura hay un teléfono, almacenando el camino recorrido desde la entrada para evitar pasar dos veces por la misma habitación.



a) ¿Cómo podríamos dercirle que buscara sin entrar en las habitaciones f y g?
b) Ampliar el programa para que PROLOG proporcione mensajes del tipo "entrando en la habitación Y", "He encontrado el teléfono en la habitación Y".
c) ¿Se pueden encontrar caminos alternativos con el programa que has hecho? Si es así, ¿dónde se podría poner un corte para evitarlo?
d) ¿Qué determina el orden en el que se buscan las habitaciones?

Ejercicio 4.5

Escribir un conjunto de predicados PROLOG para generar nuevos átomos a partir de una raiz dada.

Ejemplo: ?- gensim(estudiante,X).
X=estudiante1;
X=estudiante2;
X=estudiante3.

4.6. Bibliografía

[Clocksin 93] Clocksin, W.F., Mellish, C.S., Programación en PROLOG. Segunda edición. Colección Ciéncia Informática. Editorial Gustavo Gili S.A., 1993.

[Lazarev 89] Lazarev, G.L., Why PROLOG? Justifying logic programming for practical applications. Prentice Hall, 1989.

[Brisset 94] P. Brisset et al., ECLiPSe 3.4 Extensions User Manual, European Computer-Industry Research Center, Munich, Germany, 1994.

5. Programación lógica y Bases de Datos

CONTENIDO

5.1. Introducción al modelo relacional y al álgebra relacional.

El modelo relacional se ha establecido como el principal modelo de datos para las aplicaciones de procesamiento de datos. Una base de datos relacional consiste en un conjunto de tablas (o relaciones), como las que aparecen en el ejemplo 5.1. Cada fila (o tupla) de la tabla representa una relación entre un conjunto de valores. Cada columna de la tabla es un atributo. Para cada atributo hay un conjunto de valores permitidos, correspondiendo al dominio de dicho atributo. Cada relación tendrá una clave primaria, que será un subconjunto de sus atributos, y cuyo valor será único dentro de la misma relación. Por la clave primaria se podrá hacer referencia a cada una de las tuplas de la relación de manera unívoca.

Ejemplo 5.1 El ejemplo clásico de suministradores y partes

Sumin	Scodigo	Snombre	estatus	Ciudad
	s1	juan	20	madrid
	s2	pedro	10	castellon
	s3	raquel	30	alicante
	s4	maria	20	valencia
	s5	luis	30	castellon

Partes	Pcodigo	Pnombre	Color	Peso	Ciudad
	p1	mesa	verde	20	castellon
	p2	silla	verde	6	castellon
	p3	armario	azul	60	alicante
	p4	sofa	amarillo	55	valencia
	p5	cama	marrón	20	madrid
	p6	libreria	roja	70	castellon

S_P	Scodigo	Pcodigo	Cantidad
	s1	p1	300
	s1	p2	200
	s1	p3	400
	s1	p4	300
	s1	p5	700
	s1	p6	100
	s2	p1	300
	s2	p2	400
	s3	p2	200
	s4	p2	200
	s4	p4	300
	s4	p5	400

El álgebra relacional es un lenguaje de consulta procedimental. Consta de un conjunto de operaciones que toman como entrada una o dos relaciones y producen como resultado una nueva relación. Las operaciones fundamentales del álgebra relacional son selección, proyección, unión, diferencia de conjuntos, producto cartesiano y renombramiento. Además de las operaciones elementales hay otras operaciones, como la intersección de conjuntos, la reunión natural, la división y la asignación, que se definen en términos de las operaciones fundamentales. Las operaciones de selección, proyección y renombramiento son unarias, porque operan sobre una sola relación. Las operaciones de unión, diferencia de conjuntos y producto cartesiano son binarias, ya que operan sobre pares de relaciones.

- La operación de selección (denotada por σ), selecciona tuplas que satisfacen un predicado dado. Por ejemplo, para seleccionar las tuplas de la relación sumin en que la ciudad es “Castellón” hay que escribir:

$$\sigma_{\text{ciudad=castellon}} (\text{sumin})$$

Para seleccionar se pueden utilizar los operadores de comparación $=,\neq,>,\geq,<,\leq$, y también se pueden combinar varios predicados con las conectivas $\wedge,\vee$.

- La operación proyección (denotada por Π), selecciona columnas o atributos de una relación, eliminando las filas duplicadas. Por ejemplo, para obtener el nombre y el código de todos los suministradores hay que escribir:

$$\Pi_{\text{scodigo, snombre}} (\text{sumin})$$

También se puede hacer composición de operaciones. Así, para obtener el nombre y el código de aquellos suministradores de la ciudad de Castellón hay que escribir:

$$\Pi_{\text{scodigo, snombre}} (\sigma_{\text{ciudad=castellon}} (\text{sumin}))$$

- La operación de unión (denotada por $\cup$), permite unir las tuplas de dos relaciones iguales (misma aridad y los dominios de los atributos de ambas relaciones tienen que ser iguales), eliminando tuplas repetidas. Por ejemplo, para obtener el nombre de los suministradores de la ciudad de Castellón o de la ciudad de Valencia hay que escribir:

$$\Pi_{\text{snombre}} (\sigma_{\text{ciudad=castellon}} (\text{sumin})) \cup \Pi_{\text{snombre}} (\sigma_{\text{ciudad=valencia}} (\text{sumin}))$$

- La operación diferencia de conjuntos (denotada por -), permite obtener las tuplas de una relación que no están en la otra relación. Ambas relaciones tienen que ser iguales. Por ejemplo, para obtener los códigos de los suministradores que no han suministrado ninguna parte, hay que escribir:

$$\Pi_{\text{scodigo}} (\text{sumin}) - \Pi_{\text{scodigo}} (\text{s\_p})$$

- La operación producto cartesiano (denotada por $\times$), permite combinar información de cualesquiera dos relaciones. Para distinguir los atributos con el mismo nombre en las dos relaciones, se pone como prefijo el nombre de la relación de donde provienen, seguido de un punto. Por ejemplo, para obtener el nombre de los suministradores de la ciudad de Castellón que hayan suministrado partes alguna vez, hay que escribir:

$$\Pi_{\text{snombre}} (\sigma_{\text{sumin.scodigo=s\_p.scodigo}} (\sigma_{\text{ciudad=castellon}} (\text{sumin} \times \text{partes})))$$

El producto cartesiano (sumin $\times$ partes) asocia todas las tuplas de sumin con todas las tuplas de partes. Después se hace una selección de aquellas tuplas cuya ciudad es Castellón, y posteriormente se hace una selección de las tuplas en las que sumin.scodigo=s_p.scodigo. Por último se hace una proyección del atributo snombre.
Más eficiente resulta hacer el producto cartesiano del resultado de la selección de aquellas tuplas que nos interesan.

$$\Pi_{\text{snombre}} (\sigma_{\text{sumin.scodigo=s\_p.scodigo}} ((\sigma_{\text{ciudad=castellon}} (\text{sumin})) \times \text{partes}))$$

- La operación renombramiento (denotada por ρ) permite dar nombre a las relaciones obtenidas como resultado de expresiones del álgebra relacional.

Las operaciones no fundamentales no añaden potencia al álgebra, aunque simplifican las consultas habituales.

- La operación intersección de conjuntos (denotada por $\cap$) permite intersectar las tuplas de relaciones iguales. Por ejemplo, para obtener los nombres de los suministradores de Castellón y de Valencia, hay que escribir:

$$\Pi_{\text{snombre}} (\sigma_{\text{ciudad=castellon}} (\text{sumin})) \cap \Pi_{\text{snombre}} (\sigma_{\text{ciudad=valencia}} (\text{sumin}))$$

La equivalencia de esta operación no elemental con las operaciones elementales es la siguiente:

$$r \cap s = r - (r - s)$$

- La operación reunión natural (denotada por Ξ) es una operación binaria que permite combinar ciertas selecciones y un producto cartesiano en una sola operación. Las selecciones implícitas son aquellas que igualan los atributos con el mismo nombre de las dos relaciones. Así, el ejemplo anterior, para obtener el nombre de los suministradores de la ciudad de Castellón que hayan suministrado partes alguna vez, se reescribiría como:

$$\Pi_{\text{snombre}}((\sigma_{\text{ciudad=castellon}}(\text{sumin}))\ \Xi\ \text{partes}))$$

La equivalencia de esta operación no elemental con las operaciones elementales es la siguiente:

$$r\ \Xi_{\theta}\ s = \sigma_{\theta}\ (r \times s)$$

- La operación de división (denotada por ÷) resulta adecuada para las consultas que incluyen la expresión “para todos”. Por ejemplo, para obtener los suministradores que suministran todas las partes, hay que escribir:

$$r1 = \Pi_{\text{scodigo,pcodigo}}\ (s\_p)$$
$$r2 = \Pi_{\text{pcodigo}}\ (\text{partes})$$
$$r3 = r1 \div r2$$
$$r4 = \Pi_{\text{scodigo}}\ (r3 \times \text{sumin})$$

La equivalencia de esta operación no elemental con las operaciones elementales es la siguiente. Dadas r(R) y s(S) (siendo R y S los atributos de las relaciones r y s, respectivamente), con $S \subseteq R$:

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

5.2. Algebra relacional versus programas lógicos

La lógica es la base de los sistemas de bases de datos. Los lenguajes de consulta de las bases de datos relacionales (por ejemplo SQL), están basados en la lógica de predicados de primer orden, al igual que PROLOG.

5.2.1. Representación de relaciones en PROLOG.

Los datos, representados mediante tablas en las bases de datos relacionales, se pueden representar en PROLOG mediante un conjunto de hechos. Las tablas de la base de datos relacional de suministradores y partes del ejemplo 5.1, se representarán en PROLOG con los siguientes hechos:

```
sumin(s1,juan,20,madrid).
sumin(s2,pedro,10,castellon).
sumin(s3,raquel,30,alicante).
sumin(s4,maria,20,valencia).
sumin(s5,luis,30,castellon).

partes(p1,mesa,verde,20,castellon).
partes(p2,silla,verde,6,castellon).
partes(p3,armario,azul,60,alicante).
partes(p4,sofa,amarillo,55,valencia).
partes(p5,cama,marron,20,madrid).
partes(p6,libreria,roja,70,castellon).
```

```
s_p(s1,p1,300).
s_p(s1,p2,200).
s_p(s1,p3,400).
s_p(s1,p4,300).
s_p(s1,p5,700).
s_p(s1,p6,100).
s_p(s2,p1,300).
s_p(s2,p2,400).
s_p(s3,p2,200).
s_p(s4,p2,200).
s_p(s4,p4,300).
s_p(s4,p5,400).
```

5.2.2. Representación de operadores del álgebra relacional con PROLOG.

Las operaciones del álgebra relacional, que permiten generar nuevas relaciones a partir de las existentes, pueden ser reproducidas fácilmente en PROLOG.

- La selección se consigue en PROLOG utilizando la coincidencia sintáctica del PROLOG, si la comparación se realiza con "=", o con una cláusula añadida, si la comparación es distinta de "=". Por ejemplo:

Ejemplo 5.1.1
Proporcionar el nombre de los suministradores de Castellón con estatus mayor que 20.

```
consulta(Snombre):- sumin(_,Snombre,Estatus,castellon),
                    Estatus>20.
```

- La proyección se consigue, en PROLOG, escribiendo como argumento(s) de la cabeza de la regla que definirá la consulta, el nombre del atributo(s) que interesa proyectar. En el ejemplo anterior se ha proyectado el atributo Snombre.

- La unión se obtiene en PROLOG definiendo predicados con el mismo nombre que resuelvan cada una de las relaciones que se deben unir (es decir, utilizando la $\vee$ lógica del PROLOG). Por ejemplo, la consulta para obtener el nombre de los suministradores de la ciudad de Castellón o de la ciudad de Valencia, se escribiría en PROLOG:

```
consulta(Snombre):- sumin(_,Snombre,_,castellon).
consulta(Snombre):- sumin(_,Snombre,_,valencia).
```

- La operación diferencia de conjuntos se obtiene en PROLOG utilizando el predicado predefinido "not". Por ejemplo, la consulta para obtener el código de los suministradores que no hayan suministrado ninguna parte, se escribiría en PROLOG:

```
consulta(Scodigo):- sumin(Scodigo,_,_,_),
                    not s_p(Scodigo,_,_).
```

- La operación producto cartesiano se obtiene en PROLOG mediante la coincidencia sintáctica. Por ejemplo, para obtener los nombres de los suministradores de la ciudad de Castellón que hayan suministrado partes alguna vez, se escribiría en PROLOG:

```
consulta(Snombre):- sumin(Scodigo,Snombre,_,castellon),
                    s_p(Scodigo,_,_).
```

- La operación intersección de conjuntos se obtiene en PROLOG utilizando también la coincidencia sintáctica. Por ejemplo, para obtener el nombre de los suministradores que han suministrado la pieza con código "p1" y la pieza con código "p2", se escribiría en PROLOG:

```
consulta(Snombre):- sumin(Scodigo,Snombre,_,_),
                    s_p(Scodigo,p1,_),
                    s_p(Scodigo,p2,_).
```

- La operación de división se obtiene en PROLOG utilizando el predicado predefinido “findall”. Por ejemplo, para obtener el nombre de los suministradores que suministran todas las partes.

```
consulta3(Snombre):-
     findall(Pcodigo,partes(Pcodigo,_,_,_,_),L_todas),
     sumin(Scodigo,Snombre,_,_),
     findall(Pcodigo,s_p(Scodigo,Pcodigo,_),L_sumin),
     subcto_partes(L_todas,L_sumin).

subcto_partes([],_).
subcto_partes([H|T],Lista):-
     miembro(H,Lista),
     subcto_partes(T,Lista).
```

Ejercicio 5.1

Resolver en PROLOG las siguientes consultas a la base de datos suministradores-partes.

a) Nombre de los suministradores que suministran al menos una parte roja.
b) Nombres de los suministradores que no suministran la parte p2.
c) Nombre de los suministradores que suministran al menos todas las partes que suministra el suministrador s2.
d) Nombre de los suministradores que suministran solo partes de Castellón.
e) Nombre de las partes que hayan sido suministradas por todos los suministradores.
f) Nombre de los suministradores "al por mayor" (que nunca suministran partes en una cantidad menor a 400 piezas).
g) Nombre de los suministradores y la cantidad total de partes que han suministrado.
h) Calcular la media de las cantidades totales que han suministrado todos los suministradores.
i) Nombre de los suministradores cuya cantidad total suministrada supere la media de cantidades totales suministradas.

Ejercicio 5.2

Se supone definida una base de datos de familias, con una sola relación que tiene la siguiente estructura:

```
familia (
        persona(Nombre, Apellido1, Apellido2, fecha(Dia, Mes, Ano),
                trabaja(Compania, Salario) ),
        persona(Nombre, Apellido1, Apellido2, fecha(Dia, Mes, Ano),
                trabaja(Compania, Salario) ),
        [
        persona(Nombre, Apellido1, Apellido2, fecha(Dia, Mes, Ano), desempleada )
        |
        Lista_otros_hijos
        ]
        ).
```

Donde la primera persona corresponde al marido, la segunda a la mujer y la lista de personas del tercer argumento de "familia" corresponde a los hijos de la pareja.
Ejemplo:

```
familia (
        persona(gustavo, gimenez, marques, fecha(25,9,66), trabaja(drt, 3),
        persona(maria, martinez, gil, fecha(15,3,65), trabaja(tte, 3),
        [
        persona(ramon, gimenez, martinez, fecha(3,5,93), desempleada)
        ]
        ).
```

Se pide escribir los predicados en PROLOG que permitan obtener:

a) Todos los datos del marido de la familia.
b) Todos los datos de la mujer de la familia.
c) La lista con todos los hijos de la familia.
d) El primer hijo de la familia. El segundo hijo de la familia.
e) Todos los datos del hijo n-ésimo de la familia.

Definir este predicado para que pueda utilizarse sin necesidad de saber que los hijos están incluidos en una lista en el predicado "familia", es decir, que se pueda llamar al predicado:

```
n_esimo_hijo(N,Familia, Persona).
```

f) Todos los datos de todas las personas de la base de datos.
g) La fecha de nacimiento de las personas.
h) El salario de las personas.
i) Nombre y apellidos de las mujeres que trabajan.
j) Nombre y apellidos de las personas desempleadas que nacieron antes de 1978.
k) Nombre y apellidos de las personas que nacieron antes de 1961 que cobren menos de 3.
l) Primer apellido del padre y primer apellido de la madre de las familias que tienen al menos 3 hijos.
m) Calcular el total de ingresos por familia.
n) Calcular los ingresos por miembro de familia en cada familia.
o) Calcular la media de ingresos de todas las familias.
p) Nombre y apellidos de las personas de familias sin hijos.
q) Nombre y apellidos de las personas de familias cuyos hijos no tengan empleo.
r) Nombre y apellidos de las personas de familias cuyo marido esta desempleado y la mujer trabaje.
s) Nombre y apellidos de los hijos cuyos padres se diferencian en edad más de 15 años.
t) Definir la relación gemelos(Hijo1, Hijo2).

5.3. Bases de datos relacionales utilizando ECLiPSe

ECLiPSe utiliza también el modelo relacional de bases de datos. Las bases de datos relacionales convencionales sólo permiten almacenar conocimiento con hechos explícitos. Sin embargo, ECLiPSe permite también almacenar conocimiento más

complejo (como por ejemplo cláusulas PROLOG), proporcionando un entorno de programación lógica para construir bases de datos de gran escala.
Para cargar la biblioteca de bases de datos de ECLiPSe, basta teclear en el prompt de ECLiPSe *lib(db)* e *import database_kernel*. Para construir una base de datos se necesita ejecutar las siguientes operaciones:

a) Crear la base de datos
b) Definir las relaciones
c) Insertar los datos en las relaciones
d) Acceder a los datos
e) Borrar datos y borrar relaciones

Para insertar, borrar y acceder a los datos de la base de datos, ECLiPSe proporciona operaciones para acceder a una tupla (utilizando el backtracking se podrá acceder a todas las tuplas) y operaciones de conjunto.

5.3.1. Creación de bases de datos

`createdb('ejemplo')` ó **`createdb('/informatica/ejemplo')`**

Si la base de datos ya está creada en otra sesión anterior, sólo es necesario abrirla con:

`opendb('ejemplo')` ó **`opendb('/informatica/ejemplo').`**

Solamente una base de datos puede estar activa cada vez. Al abrir otra base de datos, se cerrará la anterior. También se cierra la base de datos actual al salir de ECLiPSe. Para cerrar una base de datos en otras circustancias, se puede utilizar **`closedb`**.

5.3.2. Definición de relaciones

Para definir la estructura de las relaciones se utiliza el predicado predefinido <=>, con la siguiente sintaxis:

```
nombre_relacion <=> [Atributo1,...,AtributoN].
```

donde cada atributo es de la forma:

```
Tipo(Nombre, Longitud, Indice)
```

Tipo puede ser **integer, real, atom** y **term**. Cualquier cadena de caracteres se representa como atom y los términos PROLOG como term.
El nombre es truncado a 31 caracteres.
La longitud de los atributos *enteros* puede ser de **1, 2** o **4** bytes; los *reales* de **4** bytes; los *atoms* pueden ser de cualquier longitud; y para el tipo *término* este argumento se ignora. El número máximo de bytes por relación es de 1000, teniendo en cuenta que un término se cuenta como 12 bytes.
Si en el argumento índice ponemos un +, ECLiPSe incluirá el atributo como índice de la relación. Si ponemos –, no lo considerará. Los atributos de tipo término no se pueden incluir como índice. Ya que toda relación debe incluir por lo menos un índice, tendrá que tener por lo menos un atributo que no sea de tipo término.
El máximo número de atributos permitidos en la relación es 50.

Ejemplo 5.2

```
sumin <=> [atom(scodigo,2,+),
     atom(snombre,12,-),
     integer(estatus,4,-),
     atom(ciudad,12,-)],

partes <=> [atom(pcodigo,2,+),
     atom(pnombre,12,-),
     atom(color,12,-),
     integer(peso,4,-),
     atom(ciudad,12,-)],

s_p <=>   [atom(scodigo,2,+),
     atom(pcodigo,2,+),
     integer(cantidad,4,-)].
```

Una forma muy rápida de borrar una relación es destruir su estructura:

```
     nombre_relacion<=>[].
```

5.3.3. Insertar datos en la base de datos

5.3.3.1. Utilizando operaciones de conjunto

El operador <++ añade a una relación cuyo nombre aparece a la izquierda del operador, una lista de tuplas que se relacionan a la parte derecha del operador y que son resultado de una expresión relacional (la forma de construir expresiones relacionales se verán en el apartado siguiente).

Ejemplo 5.3

```
partes <++ [
                [p1,mesa,verde,20,castellon],
                [p2,silla,verde,6,castellon],
                [p3,armario,azul,60,alicante],
                [p4,sofa,amarillo,55,valencia],
                [p5,cama,marron,20,madrid],
                [p6,libreria,roja,70,castellon]
           ].
```

Si los datos están introducidos como hechos PROLOG (como puede ocurrir si se ha resuelto el primer ejercicio), se pueden pasar a relaciones de la base de datos de ECLiPSe utilizando el predicado predefinido *findall*, como se puede ver en el siguiente ejemplo.

Ejemplo 5.4

```
insertar:-
     findall([A,B,C,D],sumin(A,B,C,D),L_sumin),
```

```
findall([A,B,C,D,E],partes(A,B,C,D,E),L_partes),
findall([A,B,C],s_p(A,B,C),Lista_todas_s_p),
sumin <++ L_sumin,
partes <++ L_partes,
s_p <++ Lista_todas_ s_p.
```

5.3.3.2. Utilizando operaciones de tupla

Los predicados predefinidos **inst_tup/1** (`ins_tup(predicado(terminos))` e **inst_tup/2** `ins_tup(predicado,[terminos]))`, añaden una tupla cada vez a la base de datos, como se muestra a continuación.

Ejemplo 5.5

```
ins_tup(partes(p1,mesa,verde,20,castellon)).
ins_tup(partes,[p1,mesa,verde,20,castellon]).
```

5.3.4. Acceso a los datos de la base de datos

Para examinar el estado de la base de datos se dispone de los siguientes predicados predefinidos (muy útiles para comprobar si las operaciones realizadas se han llevado a cabo como es de esperar).

`helpdb/0`
Lista los nombres de todas las relaciones de la base de datos

`helprel/1`
Da información de la relación cuyo nombre se pasa como argumento

`printrel/1`
Lista todas las tuplas almacenadas en la relación cuyo nombre se pasa como argumento

`arity/2`
Devuelve en el segundo argumento la aridad (el número de atributos) de la relación relación cuyo nombre se pasa como argumento

`cardinality/2`
devuelve en el segundo argumento la cardinalidad (el número de tuplas) de la relación cuyo nombre se pasa como argumento

5.3.4.1. Utilizando operaciones de conjunto

De manera similar a como el predicado "is" permite escribir expresiones aritméticas en PROLOG, el predicado predefinido "isr" permite incluir expresiones del álgebra relacional en los programas ECLiPSe. La sintaxis es la siguiente:

```
Nombre_relacion isr Expresion_relacional
```

Las relaciones construidas con isr son temporales y se destruirán cuando se cierre la base de datos. Para cualquier relación que deseemos que sea permanente se tiene que crear su estructura, con el predicado <=>.
Para construir expresiones del álgebra relacional, existen operadores que permiten:

5.3.4.1.1 **Seleccionar tuplas**

Para seleccionar tuplas se utiliza el operador **`where.`** Para la comparación simple entre atributos se pueden utilizar los operadores ==, \==, >=, =<, < y >. Para conjunción de comparaciones simples se utiliza el operador **and**.

Ejemplo 5.5
Elegir las tuplas de aquellos sumunistradores cuyo estatus es mayor que 20.

```
mayor_20 isr sumin where estatus>20.
```

Elegir las tuplas de las partes cuyo color sea verde y pesen más de 15.

```
verde_y_azul isr partes where color==verde and peso>15.
```

Elegir las tuplas de las partes cuyo color sea verde o azul.

```
verde_o_azul isr partes where color==verde,
verde_o_azul isr partes where color==azul.
```

1) Operador **join**

La sintaxis del operador **`join`** es la siguiente:

```
Rel isr Rel1 :*: Rel2 where Condicion
```

El nombre de la relación puede ser una constante, como en el ejemplo 5.5, o puede ser una variable, en cuyo caso PROLOG le asignará un nombre. La condición del join es opcional, si no se incluye se consigue realizar el producto cartesiano.

2) Operador **proyección**

La sintaxis del operador **`proyección`** es la siguiente:

```
Rel isr Lista_a_proyectar :^: Expresion_relacional
```

Si en la elección de atributos en la condición hay ambigüedad, se puede especificar la relación a la cual se refiere utilizando el operador (^), como en el anterior ejemplo (sumin^scodigo).

Ejemplo 5.6
Escribir el nombre de los suministradores de Castellón que han suministrado alguna parte.

```
R1 isr [nombre] :^:
sumin :*: s_p where sumin^scodigo==s_p^scodigo and
ciudad==castellon.
```

Para obtener el resultado que se pide, se puede utilizar más de una relación auxiliar. Muchas veces será de utilidad para aumentar la claridad de la expresión y otras será necesario para el correcto funcionamiento de la expresión.

Ejemplo 5.7

```
R1 isr sumin :*: s_p where
sumin^scodigo==s_p^scodigo and ciudad==castellon,
```

```
R2 isr [nombre] :^: R1.
```

3) Operador **diferencia**

La sintaxis del operador **`diferencia`** es la siguiente:

```
Rel isr Rel1 :-: Rel2 where Condicion
```

La condición es opcional. Para el operador diferencia, los atributos de las dos relaciones que participan tienen que ser los mismos.

Ejemplo 5.7

```
X isr a :-: b.
```

Crea una relación temporal que contiene las tuplas de a que no están en b.

4) Operador **unión**

La sintaxis del operador **`unión`** es la siguiente:

```
Rel isr Rel1 :+: Rel2 where Condicion
```

La condición también es opcional.Para el operador unión, los atributos de las dos relaciones que participan tienen que ser los mismos.
El operador unión está incluido por completitud sintáctica. Se puede sustituir por la siguiente secuencia:

```
Rel isr Rel1 where Condicion,
Rel2 isr Rel where Condicion.
```

5) Operador ++>

El operador **`++>`** permite recuperar un conjunto de tuplas como una lista PROLOG, para su posterior manejo con predicados PROLOG. Su sintaxis es:

```
Lista_de_proyeccion :^: Expresion_relacional ++> Lista
```

Ejemplo 5.8

Almacenar en una lista todas las tuplas de las partes excepto de aquellas que sean verdes.

```
?-partes where color \==verde  ++> Lista.
     Lista=[[p3,armario,azul,60,alicante],
     [p4,sofa,amarillo,55,valencia],
     [p5,cama,marron,20,madrid],
     [p6,libreria,roja,70,castellon]].
```

5.3.4.2. Cálculo de agregados

Un atributo de una relación de la base de datos puede tener una propiedad agregada, es decir, cuando se inserte una nueva tupla en la Base de Datos se ejecuta una operación agregada entre las tuplas de la relación y la nueva tupla insertada. En ECLiPSe existen cuatro operaciones agregadas:

`min` necesita atributos de tipo entero, real o átomo y se instanciará al menor valor de la tupla insertada.

max necesita atributos de tipo entero, real o átomo y se instanciará al mayor valor de la tupla insertada.

sum necesita atributos de tipo real o entero y se instancia a la suma de todas las tuplas insertadas.

count necesita atributos de tipo entero y se instanciará al número de tuplas insertadas.

Una relación puede contener sólo atributos agregados (en cuyo caso contendrá al menos una tupla) o atributos agregados y no agregados. En este último caso, los atributos no agregados se llaman de agrupación ya que permiten calcular los agregados sobre grupos de tuplas.

Ejemplo 5.9

Se quiere calcular la media salarial de los empleados de cada departamento de una empresa. Para ello se definiranlas relaciones empleado y salarios_por_departamento de la siguiente forma:

```
empleado <=>
       [atom(nombre,30,+),
       atom(departamento,30,+),
       integer(salario,4,-)],

salarios_por_departamento <=>
        [atom(departamento,30,+),
        sum(integer(suma_salarios,4,-)),
        count(integer(empleados_por_departamento,4,-))],

salarios_por_departamento <++
       [departamento,salario,salario]:^:empleado,

retr_tup(salarios_por_departamento,[Dep,Sum,Count]),
Media is Sum/Count.
```

5.3.4.3. Utilizando operaciones de tupla

Los predicados predefinidos **retr_tup/1, retr_tup/2 y retr_tup/3**, recuperan una tupla cada vez de la base de datos, como se muestra a continuación.

Ejemplo 5.10

```
retr_tup(partes (A,B,C,D,E)).
retr_tup(partes, [A,B,C,D,E]).
retr_tup(partes, [A,B,C,D,E], peso >20).
```

Para hacer trasparente la recuperación de tuplas de la base de datos, se podría definir el siguiente predicado:

```
partes(A,B,C,D,E):- retr_tup(partes (A,B,C,D,E)).
```

Tanto el acceso a la base de datos por tuplas como por conjuntos pueden usarse para resolver las mismas consultas.

Ejemplo 5.11

```
temp isr [a1,b3] :^: a :*: b where a2==b1,
result isr [a1,c1] :^: temp :*: c where b3==c2.
```

La anterior solución estaría orientada a conjuntos.

```
retr_tup(a(A1,A2,_)), retr_tup(b(A2,_,B3)), retr_tup(c(C1,B3)).
```

La anterior solución estaría orientada a tuplas.

Ambas serían equivalentes. En la mayoría de las ocasiones, las operaciones orientadas a conjuntos son más eficientes, ya que se pueden realizar joins más inteligentes. Por otra parte, los accesos por tuplas permiten una integración más fácil con otros objetivos PROLOG, y por tanto sepueden expresar condiciones más complejas. Además, la solución orientada a conjuntos utiliza relaciones auxiliares, que si tienen un tamaño muy grande puede llegar a compensar el uso de la recuperación por tuplas, un poco más lento.

5.3.5. Borrado de datos de la base de datos

5.3.5.1. Utilizando operaciones de conjunto

El operador **<--** borra las tuplas de una relación que se le indican con una expresión relacional o con una lista, como en los siguientes ejemplos.

Ejemplo 5.12

```
sumin <-- sumin where ciudad \==castellon.
sumin <-- [[s1,juan,20,madrid], [s4,maria,20,valencia]].
```

5.3.5.2. Utilizando operaciones de tupla

Los predicados predefinidos **del_tup/1, del_tup/2 y del_tup/3** borran una tupla cada vez de la base de datos, como se muestra a continuación.

Ejemplo 5.13

```
del_tup(partes (p1,mesa,verde,20,castellon)).
del_tup(partes, [p1 ,_,_,_,_]).
del_tup(partes, [_,Nombre,_,_,_], peso>20).
```

> ***Ejercicio 5.3***
>
> Resolved el ejercicio 5.1 con los predicados predefinidos que ofrece la biblioteca de ECLiPSe lib(db).

5.4. Bibliografía

[ECLiPSe, 94] ECLIPSE 3.4 (ECRC Common Logic Programming System). User Manual and ECLIPSE DataBase and Knowledge Base. July 1994.

[Clocksin 93] Clocksin, W.F., Mellish, C.S., Programación en PROLOG. Colección Ciencia Informática, 2ª edición. Editorial Gustavo Gili, S.A., 1993.
(Capítulo 7, puntos 7.8 y 7.9)

[Lazarev,89] Lazarev. Why PROLOG? Justifying logic programming for practical applications. Prentice Hall, 1989.
(Capítulo 4, punto 4.4)

[Bratko, 90] I. Bratko. PROLOG. Programming for Artificial Intelligence. Second Edition. Addison- Wesley. 1990.
(puntos 4.1 y 4.2)

[Silberschatz et al. 98] A. Silberschatz, H.F.Korth, S. Sudarshan, Fundamentos de bases de datos, Tercera edición. Ed. McGraw Hill, 1998.

[Date, 93] C.J.Date, Introducción a los Sistemas de bases de datos. Volumen I. Quinta edición. Ed. Addison-Wesley Iberoamericana, 1993.

[Lucas, 88] R. Lucas, Database Applications using PROLOG. Ellis Horwood books in Computer Science. Halsted Press: a division of Jonh Wiley & Sons, 1988.

[Nussbaum, 92] M.Nussbaum, Building a deductive database. Ablex Publishing Corporation Norwood, New Jersey, 1992.

[Kumer-Das, 92] S. Kumer-Das, Deductive databases and logic programming. Addison-Wesley publishing company, 1992.

[Mizoguchi, 91] F. Mizoguchi, editor. PROLOG and its applications. A Japanese perspective. Chapman & Hall Computing, 1991. (Capítulo 5)

[Gray et al., 88] P.M.D.Gray, R.J.Lucas, editors. PROLOG and databases. Implementations and new directions. Ellis Horwood Series in AI. 1988.

6. Programación lógica y gramáticas

CONTENIDO

6.1. El procesamiento del lenguaje natural

El término procesamiento del lenguaje natural se refiere a una forma restringida del lenguaje humano. El lenguaje usado por los humanos en su totalidad es demasiado complicado y ambiguo para ser tratado por un ordenador. En el caso de los humanos la ambigüedad es resuelta por el contexto y por muchos años de experiencia.
Como PROLOG tiene su origen en la lógica, es el lenguaje de programación más apropiado para el tratamiento del lenguaje natural. De hecho, PROLOG fue diseñado originalmente para traducir lenguaje natural.
Básicamente la tarea de procesar el lenguaje natural consiste en las siguientes tres fases:

1. Análisis léxico
2. Análisis sintáctico
3. Análisis semántico

Un lenguaje puede verse como un conjunto (normalmente infinito) de frases de longitud finita. Cada frase está compuesta de símbolos de algún alfabeto, según una combinación determinada para formar frases correctas. Para especificar cómo construir frases correctas en cualquier lenguaje se utiliza como formalismo las gramáticas, que constituyen un conjunto de reglas que definen la estructura legal en un lenguaje.

6.1.1. Gramáticas libres de contexto

Se dice que una gramática es libre de contexto si sus constituyentes son estructuralmente mutuamente independientes, es decir, la estructura de una parte no influye en la estructura de otra parte.
Formalmente una gramática libre de contexto es una 4-tupla (N,T,P,S) donde N y T son un conjunto finito y disjunto de identificadores del alfabeto no terminales y terminales, respectivamente. S es un símbolo no terminal que se denomina símbolo inicial. P es un conjunto finito de reglas de producción del estilo:

<oracion> → <sintagma_nominal> <sintagma_verbal>
<sintagma_nominal> → <determinante> <nombre>
<sintagma_verbal> → <verbo> <sintagma_nominal>

<sintagma_verbal> → <verbo>
<determinante> → el
<determinante> → la
<nombre> → hombre
<nombre> → manzana
<verbo> → come

Donde los no terminales se escriben entre paréntesis angulados (esta notación para describir gramáticas se denomina BNF —Backus-Naur form—). La derivación de una frase a partir del no terminal inicial de la gramática (que en el ejemplo anterior puede ser <oración>) puede describirse mediante un árbol de derivación (o de análisis) como el que indica la figura 6.1:

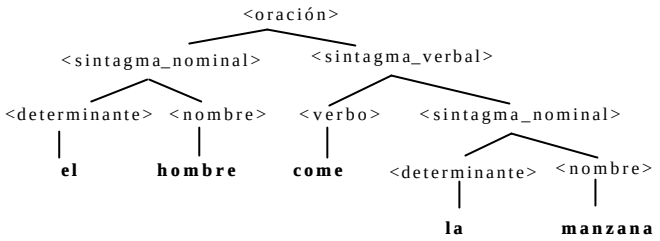


Figura 6.1 Arbol de derivación.

Al problema de construir un árbol de análisis para una oración se llama problema de análisis sintáctico. Al programa que construye árboles de análisis para las oraciones de una lengua se llama analizador (parser).
Existe un gran parecido entre programas lógicos y gramáticas libres de contexto, por lo que resulta sencillo construir un analizador en PROLOG para la gramática libre de contexto del ejemplo anterior, que sería el siguiente:

Ejemplo 6.1

```
oracion(Z):- conc(X,Y,Z), sintagma_nominal(X),
             sintagma_verbal(Y).

sintagma_nominal(Z):- conc(X,Y,Z), determinante(X),
             nombre(Y).

sintagma_verbal(Z):- conc(X,Y,Z), verbo(X),
             sintagma_nominal(Y).
sintagma_verbal(X):- verbo(X).

determinante([el]).
determinante([la]).

nombre([hombre]).
nombre([manzana]).

verbo([come]).
```

El programa anterior puede comprobar si una frase es correcta de acuerdo a la gramática libre de contexto que se ha definido. Por ejemplo:

```
?-oracion([el,hombre,come,la,manzana]).
```

El programa anterior es bastante ineficiente, ya que conc/3 generará todas las particiones posibles de la lista original Z y se necesitará tiempo para encontrar la partición correcta. Una versión del programa anterior más eficiente sería la siguiente (se utiliza lo que se denomina listas diferenciadas).

Ejemplo 6.2

```
oracion(S0,S):- sintagma_nominal(S0,S1),
                sintagma_verbal(S1,S).

sintagma_nominal(S0,S):- determinante(S0,S1),
                nombre(S1,S).

sintagma_verbal(S0,S):- verbo(S0,S).
sintagma_verbal(S0,S):- verbo(S0,S1),
                sintagma_nominal(S1,S).

determinante([el|S],S).
determinante([la|S],S).

nombre([hombre|S],S).
nombre([manzana|S],S).

verbo([come|S],S).
```

Para que el programa analizador anterior compruebe si una frase es correcta deberemos introducirle la siguiente pregunta:

```
?-oracion([el,hombre,come,la,manzana],[]).
```

Y para que el analizador genere todas las sentencias legales de la gramática usaremos la siguiente pregunta:

```
?- oración(S,[]).
```

Aunque muchas de las frases que son legales en la gramática, no son correctas semánticamente, por ejemplo “la manzana come el hombre”.
Una de las deficiencias de las gramáticas libres de contexto es que no se pueden utilizar para describir la concordancia de número, entre el nombre y el verbo o el artículo y el nombre, ya que ignoran la interdependencia estructural de los constituyentes de la sentencia.
Para superar estas deficiencias, Colmerauer en 1978, extiende las gramáticas libres de contexto a las gramáticas de cláusulas definidas.

6.1.2. Gramáticas de cláusulas definidas

Las gramáticas de cláusulas definidas son una extensión de las gramáticas libres de contexto en donde se permite a los no terminales contener argumentos que representen la interdependencia de los componentes de una frase.
Por ejemplo, se puede añadir a los terminales del ejemplo anterior los argumentos "singular" o "plural" para asegurar la concordancia de número (los no terminales también tendrían un argumento de más para definir una frase en singular o plural, etc.).

Ejemplo 6.3

```
determinante(singular,[el|S],S).
determinante(plural,[los|S],S).
```

Ejercicio 6.1

Probar el programa definido en los ejemplos 6.1 y 6.2, permitiendo demostrar que otras frases con la misma estructura sintáctica también pertenecen a la gramática.

Ejercicio 6.2

Para el programa anterior, resolver el problema de la concordancia de número.

Ejercicio 6.3

Basándose en la solución al ejercicio anterior, escribir otro conjunto de predicados PROLOG que analicen frases y devuelvan la estructura de dichas frases.
Por ejemplo:

?-oracion(S,[el,hombre,come,la,manzana],[]).

S = oracion(sintagma_nom(det(el), nombre(manzana)), sintagma_verb(verbo(come), sintagma_nom(det(la), nombre(manzana)))).

(La solución a este ejercicio se puede encontrar en [Van Lee 93], C10, pag. 483)

Ejercicio 6.4

Escribir un programa PROLOG que permita analizar las siguientes frases:

"Juan ama a María"
"todo hombre que vive ama a una mujer"

Ejercicio 6.5

Escribir un programa PROLOG que comprenda frases sencillas en castellano, con el siguiente formato:

_____ es un(a) _____
Un(a) _____ es un(a) _____
¿Es _____ un(a) _____?

El programa debe dar una de las siguientes respuestas (si, no, ok, desconocido), sobre la base de oraciones previamente dadas. Por ejemplo:

```
Juan es un hombre.
Ok
Un hombre es una persona.
Ok
¿Es Juan una persona?
Sí
¿Es María una persona?
desconocido
```

Cada oración debe traducirse a una cláusula PROLOG, que se incluye en la base de datos o se ejecuta, según sea lo apropiado. Así, las traducciones de los ejemplos anteriores son:

```
hombre(juan).
persona(X):- hombre(X).
?-persona(juan).
?-persona(maria).
```

Úsense reglas gramaticales si se considera apropiado. La cláusula superior para controlar el diálogo podría ser:

```
habla:-
        repeat,
        read(Oracion),
        analizar(Oracion,Clausula),
        responer_a(Clausula),
        Clausula=stop.
```

Ejercicio 6.6

Escribir un programa, utilizando reglas gramaticales, para analizar oraciones de la forma:

```
Pedro vio a Juan.
María fue vista por Juan.
Pedro dijo a María que viera a Juan.
Se cree que Juan ha sido visto por Pedro.
¿Se sabe si Juan ha dicho a María que vea a Luís?
```

Ejercicio 6.7

Escribir un programa que analice oraciones sobre acontecimientos de un edificio de oficinas, como por ejemplo, "Pérez estará en su oficina a las 3 p.m. para una reunión". Puede resultar conveniente utilizar reglas gramaticales para reconocer el "castellano comercial". El programa deberá seguidamente escribir un resumen de la frase, diciendo quién, qué, dónde y cuándo. Por ejemplo:

```
        quien:          perez
```

	donde:	oficina
	cuando:	3 pm
	que:	reunion

6.2. Desarrollo de un compilador de PASCAL (versión reducida)

El objetivo de un compilador es traducir un programa en lenguaje fuente a un programa en lenguaje ensamblador, para una arquitectura de computador en particular. En el proceso de compilación se puede distinguir las etapas mostradas en la figura 6.2: análisis léxico, análisis sintáctico, preprocesado, generación de código, optimización y ensamblado.
Vamos a fijarnos en una secuencia de instrucciones en un lenguaje similar al PASCAL, que asigna a la variable *r* el factorial de un número *n*.

```
c:=1;
r:=1;
while c<n do (c:=c+1; r:=r*c)
```

(En PASCAL los paréntesis se sustituirían por las palabras reservadas BEGIN END y aquí los puntos y comas se utilizan para separar instrucciones, no al final de las mismas).

El árbol sintáctico correspondiente al fragmento de programa anterior, representado en PROLOG (donde cada instrucción se representa en PROLOG con un término) sería:

```
assign(c,1); assign(r,1); while(c<n, (assign(c,c+1); assign(r,r×c))).
```

En este apartado de la práctica se verá cómo PROLOG puede utilizarse para la tarea de compilación correspondiente a la generación de código, para una arquitectura de ordenador RISC (Reduced Instruction Set Computers)[2]. En una máquina RISC el número de instrucciones básicas permitidas se reduce al mínimo y todas las operaciones aritméticas son realizadas a través de registros, cuyo acceso es muy rápido comparado con el acceso a memoria principal. Este diseño es utilizado en máquinas potentes actualmente.

[2] En [Cloksin 97] se puede encontrar dos compiladores reducidos para una arquitectura de ordenador de acumulador único y para una máquina de pila, respectivamente.

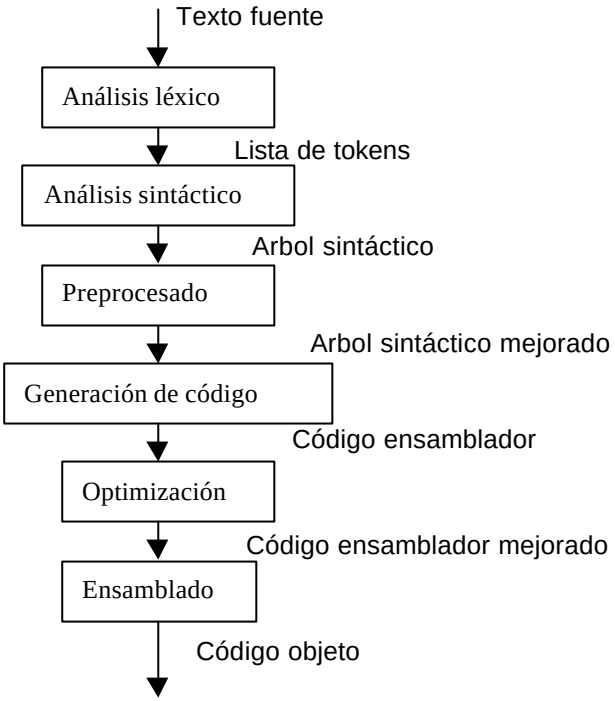


Figura 6.2 Etapas del proceso de compilación

Consideramos un conjunto reducido de instrucciones que se pueden ejecutar con esta máquina:

CODIGO DE LA OPERACIÓN	NOMBRE	DESCRIPCION
MOVC x, r	Mover constante	Se inserta la constante x en el registro r
MOVM x, r	Mover de memoria	Se inserta en el registro r el contenido de la posición de memoria x
STM r, x	Almacenar en memoria	Se almacena en la posición de memoria x el contenido del registro r
ADD ra, rb	Sumar	Suma el contenido del registro ra y el contenido del registro rb y el resultado lo inserta en rb
MUL ra, rb	Multiplicar	Multiplica el contenido de los registros ra y rb y deja el resultado en rb
CMP ra, rb	Comparar	Compara los contenidos de los registros ra y rb, y restablece los bits de condición
BR x	Ramificación incondicional	Va a la posición especificada con la etiqueta x
BGE x	Ramificación si mayor o igual que	Si el bit de condición N es mayor o igual que el bit de condición V, ir a la posición indicada por la etiqueta x
x:	Etiqueta con un único identificador	Indica el destino de una ramificación

El código en lenguaje ensamblador que resulta después de la compilación de la secuencia de instrucciones que calculaba el factorial podría ser el siguiente:

	MOVC 1, R1 STM R1, c	c:=1
	MOVC 1, R1 STM R1, r	r:=1
L1:		

	MOVM c, R1
	MOVM n, R2
	CMP R1, R2
	BGE L2
		If c>=n, go to L2

	MOVM c, R1
	MOVC 1, R2
	ADD R2, R1
	STM R1, c
		c:= c+1

	MOVM r, R1
	MOVM c, R2
	MUL R2, R1
	STM R1, r
		r:=r × c

	BR L1	Go to L1
L2:		

El siguiente programa en PROLOG (llamado code generator “cg”) serviría para generar el código anterior en lenguaje ensamblador para la secuencia que calcula el factorial.

```
cg(I,L,[movc(I,r(L))]):- integer(I).
cg(A,L,[movm(A,r(L))]):- atom(A).
cg(X+Y,L,[CX,CY,add(r(L1),r(L))]):-
          cg(X,L,CX), L1 is L+1, cg(Y,L1,CY).
cg(X*Y,L,[CX,CY,mul(r(L1),r(L))]):-
          cg(X,L,CX), L1 is L+1, cg(Y,L1,CY).
cg(assign(X,Y),L,[CY,stm(r(L),X)]):-
          cg(Y,L,CY).
cg(while(X,S),L,[label(R1),CX,SX,br(R1),label(R2)]):-
          ct(X,L,CX,R2), cg(S,L,SX).
cg((A;B),L,[CA,CB]):-
          cg(A,L,CA), cg(B,L,CB).

ct(X<Y,L,[CX,CY,cmp(L,L1),bge(R)],R):-
          cg(X,L,CX), L1 is L+1, cg(Y,L1,CY).
```

Si el programa fuente estubiera almacenado como la siguiente cláusula:

```
ex((  assign(c,1);
      assign(r,1);
      while((c<n); (asssign(c,c+1); assign(r,
r*c))))).
```

La siguiente pregunta PROLOG generaría el código en lenguaje ensamblador.

```
?-ex(X), cg(X,1,C).
```

En C devolvería una lista sin aplanar de instrucciones en lenguaje ensamblador.

Ejercicio 6.8

Modificar el generador de código anterior para que proporcione como salida una lista aplanada de código en lenguaje ensamblador, utilizando el concepto de listas diferenciadas.

Ejercicio 6.9

Añadir las cláusulas necesarias al generador de código anterior para que compile código para la instrucción "if...then...else". El término PROLOG if(x,y) puede denotar el nodo "if x then y" del árbol sintáctico. El término if(x,y,z) puede representar el nodo "if x then y else z".

6.3. Bibliografía

[Bratko 94] Bratko, I., PROLOG. Programming for Artificial Intelligence. 2ª edición. Ed. Addison-Wesley, 1994.
(Capítulo 17)

[Clocksin 93] Clocksin, W.F., Mellish, C.S., Programación en PROLOG. Colección Ciencia Informática, 2ª edición. Editorial Gustavo Gili, S.A., 1993.
(Capítulo 9)

[Clocksin 97] Clocksin, W.F., Clause and effect: PROLOG programming for the working programmer. Ed. Springer, 1997.
(El Capítulo es de lectura recomendada para el apartado 6.2 de la práctica)

[Nilson 94] Nilson, U., Matuszynski, J., Logic, Programming and Prolog. Ed John Wiley & Sons, 1994.
(Capítulo 10)

[Van Le 93] Van Le, T., Techniques of PROLOG programming with implementation of logical negation and quantified goals. Ed. John Wiley & Sons, Inc., 1993.
(El Capítulo 10 es de lectura recomendada para el apartado 6.1 de esta práctica. Además este tema contiene muchos ejercicios interesantes que están resueltos, y otro conjunto de ejercicios que no están resueltos y que pueden servir de base para el desarrollo de proyectos de la asignatura).

[Walker 90] Walker, A., et. al., Knowledge Systems and Prolog: Developing Expert, Database, and Natural Languaje Systems. 2ª edición. Addison-Wesley, 1990.

(El Capítulo 5 hace un tratamiento muy completo del tema, conveniente para aquellos alumnos que deseen profundizar o quieran desarrollar un proyecto de la asignatura sobre este tema)

[Mizoguchi, 91] F. Mizoguchi editor, PROLOG and its applications. A Japanese perspective. Chapman and Hall Computing, 1991. (Capítulo 4)

[Pereira, 94] F.Pereira, S.M.Shieber, PROLOG and Natural-language Analysis. CSLI Publications, 1987.

[Covington, 94] M.A.Covington, Natural Language processing for PROLOG programmers. 1st edition. Prentice Hall Inc. 1994.

[Matthews, 98] C. Matthews, An Introduction to Natural Language Processing through PROLOG. 1st edition. Computer Science, linguistics. Series learning about language. 1998.

http://www.landfield.com/faqs/natural-lang-processing-faq
Página Web de la “frequently asked questions and aswers” para el procesamiento del lenguaje natural. En ella se puede encontrar bibliografía actualizada, completa y reciente sobre este tema.

7. Programación lógica y Sistemas Expertos

CONTENIDO

7.1. ¿Qué es un Sistema Experto?

Un sistema experto es un programa que se comporta como un experto para algún dominio de aplicación, normalmente reducido. Debe ser capaz de explicar las decisiones que ha ido tomando y el razonamiento subyacente. Algunas veces es interesante que los sistemas expertos manejen incertidumbre y conocimiento incompleto.
Para el desarrollo de un sistema experto se distinguen tres módulos:

1. la base de conocimiento,
2. el motor de inferencia y
3. la interfaz con el usuario.

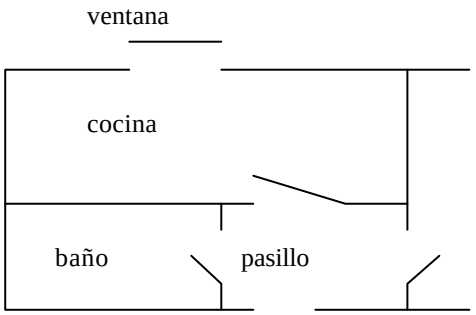
La base de conocimiento contiene el conocimiento específico del dominio de la aplicación, es decir, un conjunto de hechos, un conjunto de reglas que definen relaciones en el dominio, y métodos, heurísticos e ideas para resolver problemas en el dominio. El motor de inferencia contiene los programas necesarios para manejar el conocimiento de la base. La interfaz de usuario permitirá una comunicación fácil y agradable entre el usuario y el sistema, proporcionando detalles del proceso de resolución del problema. El motor de inferencia y la interfaz de usuario se pueden ver como un módulo, que le llamaremos *shell.* Teóricamente, el *shell* es independiente de la base de conocimiento. Sin embargo, para sistemas expertos complejos, un mismo *shell* no funciona todo lo bien que sería deseable para distintas bases de conocimiento, a no ser que los dominios de las aplicaciones sean muy similares. Aunque se necesiten algunas modificaciones en el *shell* cuando se cambia de dominio, los principios del *shell* permanecen invariables para distintas bases de conocimiento.

7.2. Representación del conocimiento

Las reglas "si-entonces", también llamadas reglas de producción, constituyen el formalismo más popular para representar conocimiento. En los sistemas expertos tienen las siguientes características deseables:

- *Modularidad*: cada regla define una parte de conocimiento pequeña y relativamente independiente.
- *Incremental*: se pueden añadir nuevas reglas a la base de conocimiento, que son relativamente independientes de las otras reglas.
- *Modificable*: se pueden cambiar reglas por otras nuevas.
- *Transparente*: es capaz de explicar las decisiones tomadas y las soluciones planteadas.

Para desarrollar un sistema experto que resuelva un problema real en un dominio concreto, el primer paso es consultar con humanos expertos en el dominio y después aprender de ese dominio uno mismo. Este paso supone un gran esfuerzo.
Para los ejemplos de este capítulo, se va a considerar una base de conocimiento juguete [Bratko 90] que trata de diagnosticar averías de agua en un piso. La avería puede darse en el baño o en la cocina. En ambos casos, la avería provoca que haya agua en el suelo del pasillo. Solamente puede haber avería en el baño o en la cocina. La base de conocimiento se muestra como una red de inferencia en la figura 7.1.
Las líneas que unen flechas significan "Y" lógica. Si las flechas no están unidas significa "O" lógica.



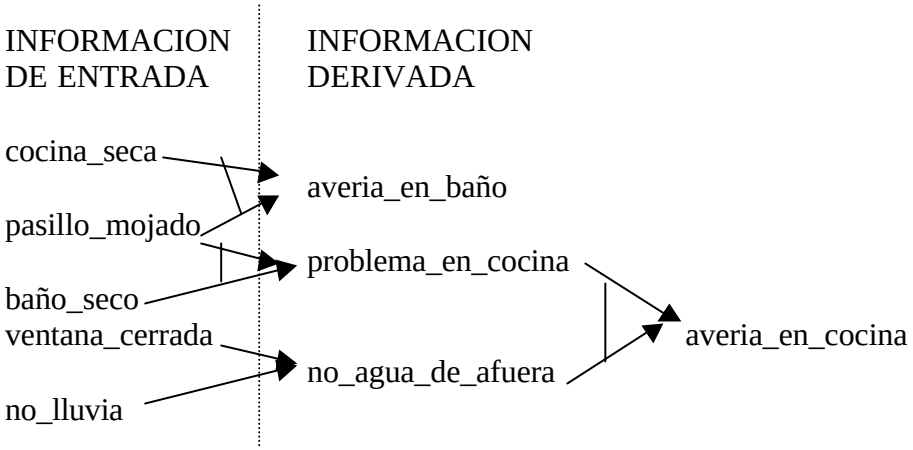


Figura 7.1 Ejemplo de una base de conocimiento para diagnosticar averías de agua en un piso.

7.3. Mecanismos de razonamiento

Cuando el conocimiento se ha representado, se necesita un mecanismo de razonamiento para obtener conclusiones a partir de la base de conocimiento. Para las reglas "si-entonces" existen básicamente dos formas de razonamiento:

1. el mecanismo de razonamiento encadenado hacia detrás y
2. el mecanismo de razonamiento encadenado hacia delante.

7.3.1. El mecanismo de razonamiento encadenado hacia detrás

El razonamiento encadenado hacia detrás parte de una hipótesis e intenta demostrarla, buscando hechos que evidencien la hipótesis. Este mecanismo de razonamiento se implementa en PROLOG de manera trivial, como muestra el siguiente ejemplo.

Ejemplo 7.1 Mecanismo de razonamiento encadenado hacia detrás en PROLOG.

```
hecho(averia_en_baño):-
     hecho(pasillo_mojado),
     hecho(cocina_seca).
hecho(problema_en_cocina):-
     hecho(pasillo_mojado),
     hecho(baño_seco).
hecho(no_agua_de_afuera):-
     (
     hecho(ventana_cerrada)
     ;
     hecho(no_lluvia)
     ).
hecho(averia_en_cocina):-
```

```
        hecho(problema_en_cocina),
        hecho(no_agua_de_afuera).
```

Los hechos reales en cada ocasión, es decir, las evidencias, se implementarán como hechos de la base de datos PROLOG. Por ejemplo:

```
hecho(pasillo_mojado).     %si no pongo hecho da problemas de
hecho(baño_seco).          % predicado no definido para aquellos
hecho(ventana_cerrada).    %predicados que no son hechos
```

Con esto se podría demostrar si es cierta la hipótesis:

```
?-hecho(averia_en_cocina).
```

Los usuarios de un sistema experto, sin embargo, no tienen por qué conocer PROLOG. Además, no hay distinción entre la base de conocimiento (que para nosotros es el conjunto de hechos) del resto del programa (las reglas), ya que para PROLOG es lo mismo. Sin embargo, sería deseable esa distinción.
El uso de la notación operador en PROLOG puede hacer que la sintaxis del sistema experto sea más amigable para un usuario que no conoce PROLOG.

Ejemplo 7.2 Razonamiento encadenado hacia detrás utilizando la notación operador.

```
:-op(800,fx,si).
:-op(700,xfx,entonces).
:-op(300,xfy,o).
:-op(200,xfy,y).

si
     pasillo_mojado y cocina_seca
entonces
     averia_en_baño.
si
     pasillo_mojado y baño_seco
entonces
     problema_en_cocina.
si
     ventana_cerrada o no_lluvia
entonces
     no_agua_de_fuera.
si
     problema_en_cocina y no_agua_de_fuera
entonces
     averia_en_cocina.
```

Las evidencias se podrían representar con los siguientes hechos:

```
hecho(pasillo_mojado).
```

```
hecho(baño_seco).
hecho(ventana_cerrada).
```

Los siguientes predicados interpretarán las reglas anteriores:

```
es_verdad(P):- hecho(P).
es_verdad(P):-
     si Condicion entonces P,
     es_verdad(Condicion).
es_verdad(P1 y P2):-
     es_verdad(P1),
     es_verdad(P2).
es_verdad(P1 o P2):-
     (
     es_verdad(P1)
     ;
     es_verdad(P2)
     ).
```

Y se podría comprobar si es cierta la hipótesis:

```
?-es_verdad(averia_en_cocina).
```

7.3.2. El mecanismo de razonamiento encadenado hacia delante

El mecanismo de razonamiento encadenado hacia delante parte de las evidencias y busca las conclusiones de esas evidencias, añadiéndolas a su vez a la base de conocimiento. Suponiendo que las reglas están almacenadas con la fórmula "si-entonces" del ejemplo 7.2, los siguientes predicados constituyen la forma de razonamiento encadenado hacia delante.

Ejemplo 7.3 Razonamiento encadenado hacia delante en PROLOG.

```
hacia_delante:-
     findall(P,hecho(P),L),     % recoge todas las evidencias en una lista L
     hacia_delante(L).

hacia_delante(L):-
     (
     nuevo_hecho_derivado(L,P),
     !,
     write('Derivado: '), write(P), nl,
     append([P],L,L1),    %append/3 es el predicado predefinido
     hacia_delante(L1)    %correspondiente a concatenar
     ;
     write('No hay mas hechos')
     ).

nuevo_hecho_derivado(L,Conclusion):-
     si Condicion entonces Conclusion,
```

```
     not miembro(Conclusion, L),
     hecho_compuesto(L,Condicion).

hecho_compuesto(L,Condicion):-
     miembro(Condicion,L).     % Es un hecho simple, está incluido en la lista
hecho_compuesto(L,Cond1 y Cond2):-
     hecho_compuesto(L,Cond1),
     hecho_compuesto(L,Cond2).
hecho_compuesto(L,Cond1 o Cond2):-
     (
     hecho_compuesto(L,Cond1)
     ;
     hecho_compuesto(L,Cond2)
     ).
```

7.3.3. Cuándo utilizar cada mecanismo de razonamiento

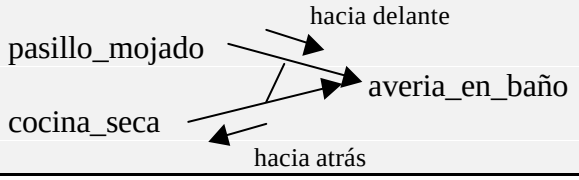
Si solamente se tiene una hipótesis a demostrar, el mecanismo de razonamiento a utilizar es el encadenamiento hacia detrás. Si hay muchas hipótesis y no hay ninguna razón para empezar con alguna de ellas en particular, es mejor utilizar el encadenamiento hacia delante. Si hay muchas evidencias y pocas hipótesis o conclusiones será más conveniente utilizar el encadenamiento hacia detrás; si hay pocas evidencias y muchas hipótesis será más útil el encadenamiento hacia delante. En algunas tareas es interesante una combinación de ambos mecanismos de razonamiento, hacia delante y hacia detrás. Por ejemplo, los diagnósticos médicos. Las evidencias iniciales de la enfermedad de un paciente pueden proporcionar una hipótesis inicial, utilizando un razonamiento hacia delante. La hipótesis inicial será confirmada o rechazada, utilizando evidencias adicionales y el razonamiento hacia detrás.

Ejercicio 7.1

Escribir los predicados PROLOG de los ejemplos anteriores y comprobar su funcionamiento para el ejemplo de las averias de agua en un piso, mediante el uso del debugger.

Ejercicio 7.2

Escribir los predicados PROLOG necesarios para combinar el razonamiento con encadenamiento hacia delante y hacia detrás. Comprobar su funcionamiento para el ejemplo de diagnosis de averias de agua en los siguientes pasos de inferencia:



7.4. Generación de explicación

El razonamiento encadenado hacia delante va escribiendo la explicación de cómo se ha ido generando la respuesta. Para el razonamiento encadenado hacia detrás, se generará un árbol de prueba de cómo la conclusión final deriva de los hechos y las reglas de la base de conocimiento. Se define "<=" como un operador infijo y se modifica el predicado "es_verdad" del ejemplo 7.2, devolviendo como segundo argumento la prueba de su veracidad, como se muestra en el siguiente ejemplo.

Ejemplo 7.4 Razonamiento encadenado hacia detrás generando explicación

```
:-op(800,xfx,<=).

es_verdad(P,P):-hecho(P).%La demostración de que el hecho(P) es verdad es P
es_verdad(P,P<=CondPrueba):-
     si Condicion entonces P,
     es_verdad(Condicion,CondPrueba).
es_verdad(P1 y P2,Prueba1 y Prueba2):-
     es_verdad(P1,Prueba1),
     es_verdad(P2,Prueba2).
es_verdad(P1 o P2, Prueba):-
     (
     es_verdad(P1,Prueba)
     ;
     es_verdad(P2,Prueba)
     ).
```

Ejercicio 7.3

Escribir los predicados PROLOG anteriores para el razonamiento encadenamdo hacia detrás generando explicación y comprobar su funcionamiento para el ejemplo de las averias de agua en un piso, mediante el uso de trazas.

7.5. Introducción de incertidumbre

Hasta aquí se ha asumido para la representación del conocimiento, que los dominios de los problemas son categóricos, es decir, que las respuestas a todas las preguntas son verdadero o falso. Sin embargo, algunos dominios de expertos son más inciertos y la respuesta puede tener más matices (por ejemplo la respuesta puede ser verdadera, altamente probable, probable, improbable o imposible). Una forma de expresar esta incertidumbre es mediante la teoría de la certeza, que permite asignar un número real a cada hecho o regla, p.e. entre 0 y 1, que determine el grado de certeza o la medida de creencia de dicho hecho o regla. Esta teoría fue desarrollada en los años 70 durante la construcción de MYCIN, un sistema experto para diagnóstico médico en enfermedades de infecciones de sangre.

En el siguiente ejemplo se muestra cómo la teoría de la certeza es aplicada para un uso efectivo.

Ejemplo 7.5 Razonamiento con encadenamiento hacia detrás y grado de certeza.

Las evidencias se escribirán utilizando otro argumento que indicará el grado de certeza. Por ejemplo:

```
hecho(pasillo_mojado,1).      % El pasillo está mojado
hecho(baño_seco,1).           % El baño está seco
hecho(cocina_seca,1).         % La cocina no está seca
hecho(no_lluvia,0.8).         % Probablemente no ha llovido
hecho(ventana_cerrada,0).     % La ventana no está cerrada
```

Las reglas se escribirán con su certeza asociada de la siguiente forma:

si Condicion entonces Conclusion : Certeza

Por ejemplo:

si
    pasillo_mojado y baño_seco
entonces
    problema_en_cocina : 0.9.

Los predicados PROLOG que interpreten estos hechos y reglas se definen como sigue.

```
:-op(800,xfx,:).

certeza(P,Cert):-hecho(P,Cert).  % La certeza de los hechos va incluida en ellos
certeza(Cond1 y Cond2,Cert):-%La certeza de una condicion compuesta por Y
     certeza(Cond1,Cert1),     % es el minimo de las condiciones simples
     certeza(Cond2,Cert2),
     minimo(Cert1,Cert2,Cert).
certeza(Cond1 o Cond2,Cert):-%La certeza de una condicion compuesta por O
     certeza(Cond1,Cert1),     % es el máximo de las condiciones simples
     certeza(Cond2,Cert2),
     maximo(Cert1,Cert2,Cert).
certeza(P,Cert):-
     si Condicion entonces P: C1, % La certeza de una conclusión es la
     certeza(Condicion,C2),       % certeza de la regla multiplicada por la
     Cert is C1*C2.               % certeza de la condición
```

Existen más aproximaciones al tratamiento del conocimiento inexacto, entre las que destacan las siguientes [Durkin 96]:

- Aproximación bayesiana o probabilística. Esta metodología se beneficia de que está fundada sobre unas bases estadísticas bien conocidas. Sin embargo, sólo podrá ser aplicada si se dispone de información sobre hechos pasados de los eventos y modelos

probabilísticos disponibles para que sean fiables las conclusiones proporcionadas; la suma de las probabilidades a favor y en contra debe ser igual a 1, y se debe asumir la independencia de los datos.

- Lógica difusa. Esta metodología se define como una rama de la lógica clásica, que utiliza grados de pertenencia a conjuntos, en vez de la categórica pertenencia verdadero/falso.

Ejercicio 7.4

Escribir los predicados PROLOG anteriores para el razonamiento encadenado hacia detrás con certeza y comprobar su funcionamiento, mediante el uso del debugger, para el ejemplo siguiente:

```
?-certeza(averia_en_cocina,C).
```

Nota: suponer las evidencias del ejemplo anterior.

Ejercicio 7.5

Se trata de constuir un sistema experto que identifica animales. El ejemplo contempla la existencia de 7 animales: onza, tigre, jirafa, cebra, avestruz, pingüino y albatros.
Las reglas son las siguientes:

Si X tiene pelo y da leche, entonces es un mamífero.
Si X tiene plumas, vuela y pone huevos, entonces es un ave.
Si X es mamífero y come carne, entonces es un carnívoro.
Si X es mamífero, tiene dientes agudos, tiene garras y tiene ojos que miran hacia delante, entonces es un carnívoro.
Si X es un mamífero y tiene pezuñas, entonces es un ungulado.
Si X es un mamífero y rumia, entonces es un ungulado.
Si X es carnívoro, de color leonado y con manchas oscuras, entonces es una onza.
Si X es un carnívoro, de color leonado y tiene franjas negras, entonces es un tigre.
Si X es ungulado, tiene patas largas, cuello largo, es de color leonado y tiene manchas oscuras, entonces es una jirafa.
Si X es ungulado, de color blanco y con franjas negras, entonces es una cebra.
Si X es un ave, no vuela, tiene patas largas, cuello largo y es blanca y negra, entonces es un avestruz.
Si X es un ave, no vuela, nada, y es blanca y negra, entonces es un pingüino.
Si X es un ave y nada muy bien, entonces es un albatros.

a) Para el siguiente conjunto de hechos:

Estirada tiene pelo.
Estirada rumia.
Estirada tiene patas largas.
Estirada tiene cuello largo.
Estirada es de color leonado.
Estirada tiene manchas oscuras.

¿Cuál es el mecanismo de razonamiento necesario para llegar a saber qué animal es Estirada? Escribir las reglas y los hechos anteriores de la forma más conveniente, y los predicados PROLOG necesarios para llegar a saber qué animal es Estirada y por qué se llega a esa conclusión.

b) Para el siguiente conjunto de hechos:

Ligera tiene pelo.
Ligera tiene dientes agudos, garras y ojos que miran hacia delante.
Ligera tiene color leonado.
Ligera tiene manchas oscuras.

¿Cuál es el mecanismo de razonamiento necesario para poder comprobar que Ligera es una onza? Escribir los predicados PROLOG necesarios para comprobar si Ligera es una onza.

Ejercicio 7.6

Diseñar un sistema experto que permita decidir automáticamente si a un cliente de una sucursal bancaria se le va a conceder un préstamo o no. 1,a cantidad del préstamo no puede superar, los 10 millones de pesetas y la utilización del préstamo no es relevante, para su concesión. La decisión de la concesión depende de la edad, el sueldo (si es estable o no y la cantidad) y si tiene avales o no.

- Si el sueldo es estable (trabajador fijo o funcionario,) no hace falta que tenga avaladores, si gana más de 150.000 ptas. mensuales.
- Si el sueldo es estable pero inferior o igual a 150.000 ptas. necesita, al menos, un avalador.
- Si el sueldo es estable pero inferior a 90.000 ptas. necesita, al menos, dos avaladores.
- Si el sueldo no es estable ha de tener al menos 25 años y más de tres avaladores.

El sistema debe permitir demostrar que la contestación a la petición de un préstamo de 5 millones de ptas. por una persona de 31 años, funcionaria, que gana 155.000 ptas. mensuales y no tiene avalador es afirmativa.

Ejercicio 7.7

Diseñar un sistema experto que permita decidir automáticamente si un médico ha de ser sancionado o no por el colegio de médicos de Castellón, después de producirse una denuncia.

Los motivos de sanción son:

- Si ha respetado o no las tarifas por visita.
- Si ha comunicado o no en un tiempo máximo de 21 días un caso de enfermedad infecciosa grave y en menos de 5 días un caso de enfermedad infecciosa no grave.
- Si ha respetado o no los protocolos establecidos por los organismos sanitarios competentes en 5 enfermedades. Consideramos que "respeta el protocolo" si lo hace al menos en la mitad de los protocolos de las enfermedades que trata.

Las reglas que rigen las sanciones son:

- Consideramos que si ha respetado el protocolo y ha comunicado un caso de enfermedad infecciosa a tiempo, no será sancionado.
- Si respeta el protocolo pero ha cobrado excesivamente y- ha comunicado una enfermedad infecciosa leve fuera de tiempo, debe ser seccionado.
- No será sancionado siempre que respete los protocolos de todas las enfermedades que trata. Nuestros médicos sólo pueden tratar 5 enfermedades cada uno.
- Las tarifas por tratamiento han de ser inferiores a 15.000 ptas. si el paciente tiene como máximo 2 enfermedades y más de 25.000 ptas. si tiene más de 4.

El sistema experto ha de poder demostrar que la contestación a la demanda de sanción es afirmativa en el caso que se pruebe que ha atendido a un paciente con 3 enfermedades, de las cuales en una se ha respetado el protocolo y en las otras 2 no; le ha cobrado 18.000 ptas; y una de ellas es infecciosa grave y ha tardado 2 días en comunicarlo a las autoridades sanitarias.

7.6. Bibliografía

[Bratko 94] Bratko, I, PROLOG. Programming for Artificial Intelligence. 2ª edición. Ed. Addison-Wesley, 1994.
(Capítulo 14 y Capítulo 15)

[Nilson 94] Nilson, U., Matuszynski, J., Logic, Programming and Prolog. Ed John Wiley & Sons, 1994.
(Capítulo 9)

[Walker 90] Walker, A., et. al., Knowledge Systems and Prolog: Developing Expert, Database, and Natural Languaje Systems. 2ª edición. Addison-Wesley, 1990.
(El Capítulo 4 hace un tratamiento muy completo del tema, conveniente para aquellos alumnos que deseen profundizar o quieran desarrollar un proyecto de la asignatura sobre este tema)

[Durkin 96] J. Durkin, Expert Systems. Design and development. Ed. Prentice-Hall, 1996.

[Jackson 90] P. Jackson, Introduction to expert systems. 2nd edition. Ed. Addison-Wesley, 1990.

[Giarratano & Riley 98] J.Giarratano, G.Riley, Expert Systems. Principles and programming. 3rd edition. Ed. PWS Publishing Company, 1998.

8. Programación lógica basada en restricciones

CONTENIDO

8.1. El problema de Satisfacción de Restricciones

Muchos problemas en Inteligencia Artificial y en otras áreas de informática pueden tratarse como problemas de combinatoria discreta [Hentenryck, 89] que intentan satisfacer un conjunto de restricciones en un espacio discreto y finito.
El problema de satisfacción de restricciones (Constraint Satisfaction Problem —CSP—) se caracteriza en términos generales como sigue:

Dado un conjunto de *n* variables:

$$\{X_1,\ldots,X_n\}$$

un dominio finito y discreto para cada variable:

$$\{D_1,\ldots,D_n\}$$

y un conjunto de restricciones sobre algún subconjunto especificado de dichas variables X_i, ..., X_j ($1 \leq i < \ldots < j \leq n$):

$$\{C_{i,\ldots,j}(X_i, \ldots, X_j)\}$$

el conjunto de soluciones es el subconjunto más grande del producto cartesiano de todos los dominios de las variables, tal que cada n-tupla de ese conjunto satisfaga todas las restricciones dadas.
Un ejemplo de un CSP es el problema de las N-reinas, que consiste en colocar N reinas en un tablero de ajedrez de N×N casillas de tal manera que las reinas no se ataquen entre sí. Para la resolución del problema se asocia una variable a cada columna del

tablero y también a cada reina, ya que dos reinas no pueden estar situadas en la misma columna. De este modo, el problema se reduce a asignar valores a las variables asociadas a cada reina. Este valor será la fila dentro de cada columna en la que la reina correspondiente está situada, teniendo en cuenta que dos reinas se atacarán si están situadas en la misma diagonal o en la misma línea horizontal. Asumiendo que las columnas están numeradas de izquierda a derecha y que X_i denota la columna de la i-ésima reina, el problema consiste en dar valores a una lista de variables $\{X_1,...,X_n\}$, tal que se satisfagan las siguientes restricciones:

- $1 \leq X_i \leq N$ para $1 \leq i \leq N$. Esta restricción unaria permite asignar valores a las variables dentro de su dominio (de 1 a N).
- $X_i \neq Xj$ para $1 \leq i < j \leq N$, corresponde a la restricción "dos reinas no pueden estar en la misma fila o línea horizontal".
- $|X_i - Xj| \neq |i-j|$ para $1 \leq i < j \leq N$, corresponde a la restricción "dos reinas no pueden estar en la misma diagonal".

Para N=8, el problema de las 8-reinas puede verse en la figura 8.1.

8.2. Métodos de resolución del problema de satisfacción de restriciones

Este tipo de problemas se resuelven utilizando métodos de búsqueda que, en general, tienen una complejidad temporal exponencial. Es decir, teóricamente un CSP es un problema NP-completo.
En los apartados siguientes se explica brevemente distintos algoritmos que resuelven el CSP.

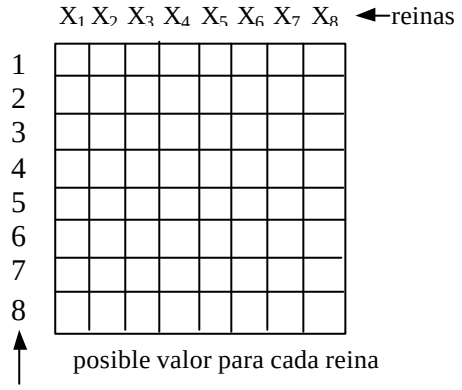


Figura 8.1. El problema de las 8-reinas.

8.2.1. Generación y prueba

El algoritmo más simple que resuelve el CSP es el que lleva a cabo una búsqueda sistemática de todo el espacio de búsqueda: es el paradigma de generación y prueba. En este paradigma, cada combinación posible de todas las variables se genera sistemáticamente y después se prueba si satisface todas las restricciones. La primera combinación que satisface todas las restricciones es una solución. Si se buscan todas las soluciones, el número de combinaciones consideradas en este método es el producto cartesiano de todos los dominios de las variables. Este es el método más ineficiente.

8.2.2. Backtracking

Un método más eficiente que el paradigma de generación y prueba es el paradigma backtracking. En este paradigma las variables se instancian secuencialmente y tan pronto como todas las variables relevantes a una restricción están instanciadas, se comprueba la validez de esa restricción. Si una instanciación parcial viola una restricción, se hace una marcha atrás (backtracking) reconsiderando la última decisión tomada en el proceso de búsqueda, es decir, la variable que se ha instanciado más recientemente y que aún dispone de valores alternativos disponibles, se instancia a otro valor posible.
El paradigma backtracking es mejor que el paradigma de generación y prueba, ya que cuando una instanciación parcial viola una resticción, el paradigma backtracking es capaz de eliminar un subespacio de todos los valores del producto cartesiano de los dominios de las variables. Aún así el paradigma backtracking tiene un coste exponencial.
La principal causa de ineficiencia del paradigma de backtracking es que la búsqueda en diferentes partes del espacio falla por las mismas razones (en inglés se denomina *trashing*), debido a la nodo-inconsistencia o/y a la arco-inconsistencia.
La nodo-inconsistencia se da cuando un valor de un dominio de una variable no satisface la restricción unaria de la variable, con lo cual la instanciación de esa variable a ese valor siempre resulta en un fallo inmediato. La nodo-consistencia puede conseguirse eliminando los valores del dominio de la variable que no satisfagan la restricción unaria.
La arco-inconsistencia se da en la siguiente situación: sea un conjunto de variables $V_1,V_2,...,V_i,...,V_j,...,V_n$ V_i y V_j, tal que para el valor V_i=a, ningún valor de V_j satisface la restricción binaria. En el árbol de búsqueda del backtracking, cuando V_i se instancie a *a*, la búsqueda fallará para todos los valores de V_j. Y ese fallo se repetirá para todos los valores que las variables V_k (i<k<j) puedan tomar. La arco-inconsistencia se evitará haciendo consistente cada arco (V_i,V_j) antes de que comience el proceso de búsqueda.
El objetivo de la investigación en este campo es mejorar la eficiencia del algoritmo de backtracking. Para ello se establecen, básicamente, dos líneas generales [Meseguer, 89]:

- Modificar el espacio de búsqueda para hacer el proceso de búsqueda más fácil.
- Utilizar heurísticos para guiar el proceso de búsqueda.

8.2.3. Algoritmos que modifican el espacio de búsqueda para hacer el proceso de búsqueda más fácil

Dentro de los algoritmos que modifican el espacio de búsqueda para hacer el proceso de búsqueda más fácil, se encuentra un grupo de algoritmos que modifican el espacio de búsqueda antes de que comience el proceso de búsqueda. Se denominan algoritmos de mejora de la consistencia. Entre ellos se encuentra el algoritmo de propagación de restricciones, y los algoritmos de arco-consistencia y camino-consistencia.
Otro conjunto de algoritmos modifican el espacio de búsqueda durante el proceso de búsqueda. Se denominan algoritmos híbridos, ya que en ellos se mezcla el proceso de búsqueda con el proceso de reducción del espacio de búsqueda. En esta categoría se incluyen los algoritmos de anticipación (*look-ahead*) [Hentenryck, 89] tales como *forward-checking, partial look ahead, full look ahead* y *really look ahead*.

8.2.3.1. Algoritmo de propagación de restricciones

Las restricciones entre pares de variables pueden almacenarse explícitamente o pueden ser implícitas. Las restricciones implícitas son efectos causados por las restricciones explícitas. La inconsistencia provocada por las restricciones implícitas se detecta

cuando actúa el conjunto de restricciones explícitas, varios pasos después de que se produzca la asignación de valores a variables, sin posibilidad de conocer cuál ha sido la restricción violada. Para evitar esto, se hacen explícitas todas las restricciones implícitas, utilizando el proceso de propagación de restricciones.
Un grafo en el que todas las restricciones han sido propagadas (y por tanto no contiene restricciones implícitas) se denomina *grafo mínimo de restricciones*. El grafo mínimo de restricciones es equivalente al grafo original, por lo que tiene las mismas soluciones. En un grafo mínimo de restricciones cada tupla de valores permitida por una restricción explícita pertenece, al menos, a una solución al problema. Para generar todas las soluciones de un grafo mínimo de restricciones se necesita combinar todos los valores definidos por las restricciones explícitas.
Encontrar el grafo mínimo de soluciones de un CSP es un problema NP-completo, y el algoritmo que lo resuelve tiene un coste exponencial [Freuder, 85]. Versiones menos estrictas que la propagación de restricciones exhaustiva consisten en una propagación de restricciones local, y son los algoritmos que aseguran la arco-consistencia y la camino-consistencia. Diferentes algoritmos que consiguen ambas consistencias tienen un coste polinomial. Es posible calcular una aproximación al grafo mínimo de restricciones (con algunas restricciones primitivas superfluas), aplicando técnicas de consistencia local [Fruehwirth 94a,94b].

8.2.3.2. Algoritmo de arco-consistencia

Como se indica previamente (sección 2.3.2), la arco-consistencia entre dos variables (V_i,V_j), se consigue eliminando los valores del dominio de V_i para los cuales ningún valor del dominio de V_j satisface alguna restricción entre las variables V_i y V_j. La eliminación de esos valores no elimina ninguna solución del problema original.
El algoritmo óptimo que asegura la arco-consistencia en un grafo de restricciones fue desarrollado por Mohr y Henderson [Mohr, 86] con una complejidad de $O(e \times d^2)$, donde e es el número de arcos en el grafo de restricciones y d es el tamaño de los dominios de las variables (se asume que todos los dominios de las variables son del mismo tamaño).
Sin embargo, en un grafo de restricciones que es arco-consistente no se asegura que una instanciación de las variables a valores de sus dominios sea una solución al CSP. Para ello se necesita una consistencia más fuerte, como la camino-consistencia.

8.2.3.3. Algoritmo de camino-consistencia

Un grafo de restricciones se dice que es k-consistente si para cada subconjunto de k-1 variables del grafo con valores que satisfacen todas las restricciones entre las k-1 variables, es posible encontrar un valor para una nueva variable del grafo tal que se satisfacen todas las restricciones entre las k variables. Si el conjunto de variables es k'-consistente para todo $k' \leq k$, entonces se dice que el grafo de restricciones es fuertemente k-consistente.
La nodo-consistencia es equivalente a la 1-consistencia y la arco-consistencia es equivalente a la fuerte 2-consistencia. La fuerte 3-consistencia es equivalente a la arco-consistencia más la camino-consistencia [Mackworth, 77].
Un grafo de restricciones es camino-consistente si para cualquier par de nodos (i,j) y todos los caminos entre ellos $i\text{-}i_1\text{-}i_2\text{-}\ldots\text{-}i_n\text{-}j$, la restricción directa entre los nodos c_{ij} es más precisa que la restricción indirecta a lo largo del camino (es decir, la composición de restricciones a lo largo del camino $c_{i,i1}\otimes\ldots\otimes c_{in,j}$) [Fruehwirth 94a,94b]. Una restricción disyuntiva es más precisa si contiene menos disyunciones.
La camino-consistencia puede utilizarse para aproximar el grafo mínimo de restricciones [Fruehwirth 94a,94b].

Un grafo de restricciones es completo si existe un eje (un par de arcos, uno en cada dirección) entre cada par de nodos. Si el grafo de restricciones es completo, para asegurar la camino-consistencia, solamente es necesario calcular repetidamente caminos de longitud dos como mucho [Fruehwirth 94a,94b], lo cual significa que para cada grupo de tres nodos (i,k,j) se calcula repetidamente la siguiente operación hasta que se alcanza un punto fijo:

$$c_{ij} := c_{ij} \oplus c_{ik} \otimes c_{kj}$$

La complejidad del algoritmo es $O(n^3)$, donde *n* es el número de nodos del grafo de restricciones [Mackworth, 85].

El algoritmo clásico para calcular la camino-consistencia de un grafo de restricciones puede encontrarse en [Mackworth, 77].

Tanto la arco-consistencia como la camino-consistencia eliminan valores de los dominios de las variables que nunca aparecerán en una solución. Sin embargo, para encontrar todas las soluciones del CSP, todavía es necesario un algoritmo de búsqueda con backtracking. De hecho, para que se puedan encontrar todas las soluciones en un grafo de restricciones, sin necesidad de utilizar un algoritmo de búsqueda con backtracking, se tendría que asegurar que es fuerte k-consistente, con k>w, siendo *w* la anchura del grafo de restricciones.

8.2.3.4. Algoritmos de anticipación (look-ahead)

Un CSP con *n* variables puede ser resuelto sin necesidad de utilizar un algoritmo de búsqueda con backtracking si el grafo de restricciones asociado es n-consistente. Una consistencia de menor grado no elimina la necesidad de backtracking en el proceso de búsqueda, salvo para ciertos tipos de problemas. Sin embargo el cálculo de la n-consistencia es incluso más costoso que el propio backtracking.

Existen un conjunto de algoritmos que combinan el proceso de búsqueda con backtracking con diferentes grados de propagación de restricciones, denominados genéricamente algoritmos de anticipación (*look-ahead).* Estos algoritmos están basados en la idea de que cada paso hacia una solución tiene que tener la evidencia de que el camino seguido no desembocará a una situación de no-solución (dead-end). Cada vez que se asigna un valor a la variable actual, se comprueban todas las variables del problema y se eliminan aquellos valores de las variables que son consideradas más tarde, si esos valores nunca aparecen en una solución.

En la categoría de algoritmos de anticipación pueden encontrarse los siguientes: *forward checking* (FC), *partial look ahead* (PL), *full look ahead* (FL) y *really full look ahead* (RFL), que se diferencian unos de otros en el grado de consistencia ejecutado en los nodos del árbol de búsqueda. La figura 2 muestra la combinación del proceso de búsqueda con diferentes grados de propagación de restricciones para cada uno de esos algoritmos, tal y como se explica a continuación.

Backtracking incorpora un grado limitado de arco-consistencia: cualquier valor de una variable considerado para instanciación que es inconsistente con instanciaciones anteriores, causa fallo inmediato.

Por su parte, el algoritmo FC filtra los dominios de todas las variables no instanciadas, cuando se va a hacer una nueva instanciación de una variable, de modo que contenga valores consistentes con la nueva instanciación. Si algún dominio de las variables no instanciadas se vuelve vacío, se produce el backtracking.

Los algoritmos PL, FL y RFL son versiones extendidas del algoritmo FC en donde se asegura la arco-consistencia incluso entre las variables no instanciadas.

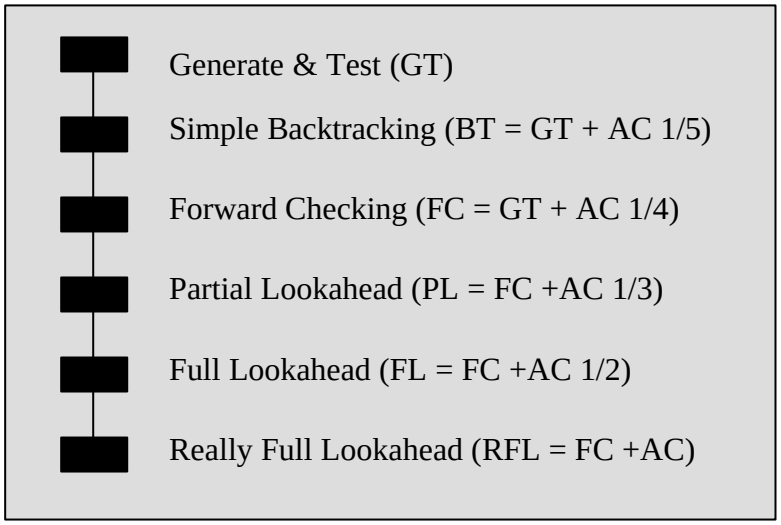


Figura 8.2. Diferentes algoritmos de satisfacción de restricciones vistos como una combinación de árbol de búsqueda y diferentes grados de propagación de restricciones (adaptado de [Kumar,92])

Un estudio comparativo de esos algoritmos puede encontrase en [Ibañez, 94] y una definición formal puede verse en [Hentenryck, 89]. Las conclusiones más importantes de los estudios comparativos realizados en [Ibañez, 94], [Hentenryck, 89] y [Nadel, 88] son los siguientes:

- El algoritmo más eficiente es FC.
- Es mejor reducir el trabajo de la propagación de restricciones, ya que la eliminación de todas las inconsitencias del grafo de restricciones no compensa el coste de realizarlo.

Para resolver un CSP eliminando los problemas del backtracking se utilizan también los esquemas de backtracking inteligente, que se realizan directamente sobre la variable que causa el fallo, y sistemas de mantenimiento de la verdad. Pero estos algoritmos tienen que ser suficientemente simples para ser eficientes.

8.2.4. Algoritmos que utilizan heurísticos para guiar el proceso de búsqueda

En general, en un proceso de búsqueda de soluciones para un CSP, no está fijado el orden en el que se eligen las variables, ni el orden en el que se asignan valores a las variables.
Existen dos criterios básicos para guiar el proceso de búsqueda:

1. Si se buscan todas las soluciones a un CSP, entonces se eligen primero los caminos que son más probables que fallen, ya que la pronta identificación de un camino que no va a dar una solución al CSP permite podar el espacio de búsqueda, y por tanto reducir su tamaño, cuanto antes.
2. Si se busca sólo una solución el criterio es el opuesto: se comprueban primero los caminos que tienen una mayor probabilidad de éxito.

Se han desarrollado heurísticos que aplican ambos criterios, junto con algunos algoritmos de consistencia, obteniendo mejoras sustanciales sobre el método de backtracking estándar. Estudios sobre este tema puede encontrarse en [Meseguer, 89] y [Kumar, 92].

8.3. El paradigma de programación lógica basada en restricciones

La forma común de resolver problemas de restricciones es escribir programas especializados, en lenguajes procedurales, que requieren mucho esfuerzo en su desarrollo y que después son difíciles de mantener, modificar y extender [Fruehwirth, 90].
La Programación Lógica Basada en Restricciones (Constraint Logic Programming CLP) es un nuevo paradigma que intenta superar esos problemas, combinando la declaratividad de la programación lógica con la eficiencia de los resolvedores de restricciones [Smolka, 94]. Consiste en reemplazar la unificación de términos de la programación lógica, por el manejo de restricciones en un dominio concreto. Esta tarea es realizada por un módulo denominado Resolvedor de Restricciones, que notifica si una restricción o un conjunto de restricciones es satisfecho. El esquema se denomina CLP(X) [Jaffar, 87], donde el argumento X representa el dominio de computación. El dominio X ha sido instanciado a los reales (CLP(R) [Jaffar, 92], al dominio de los racionales (CLP(Q)), a restricciones booleanas que incluyen funciones de verdad, a dominios finitos (CLP(FD) [Hentenryck, 91b] y a intervalos temporales (CLP(Temp) [Ibañez, 94]), entre otros.
El esquema CLP(X) permite resolver problemas de restricciones de manera eficiente, debido principalmente a dos razones:

- El Resolvedor de Restricciones es un algoritmo especializado e incremental. Es incremental porque cuando se añade una nueva restricción *C* a un conjunto de restricciones ya resuelto *S*, el resolvedor no tiene que resolver el conjunto de restricciones S {C} desde el principio.
- Las restricciones son usadas activamente (no pasivamente como en el esquema de generación y prueba), podando el espacio de búsqueda a priori.

Sin embargo, para utilizar el esquema CLP(X) se necesita conocer la semántica de las restricciones implicadas [Hentenryck, 89].
Una introducción informal a CLP puede consultarse en [Fruehwirth, 93], y una explicación formal puede verse en [Jaffar, 87] y [Hentenryck, 91a].

8.4. La programación lógica basada en restricciones sobre dominios finitos CLP(FD)

La Programación Lógica basada en Restricciones sobre Dominios Finitos (Constraint Logic Programming over Finite Domains CLP(FD)) proporciona una aritmética completa para variables restringidas a valores enteros o elementos atómicos. Restricciones sobre dominios finitos son la parte central de muchos problemas de búsqueda que se aplican, entre otras cosas, a la planificación de tareas. Debido a la importancia de las aplicaciones, las técnicas de restricciones sobre dominios finitos han sido ampliamente estudiadas en el campo de la Inteligencia Artificial desde 1970 [Hentenryck, 91b].
La terminología más importante de este paradigma abarca los siguientes conceptos [Brisset, 94]:

- Una variable dominio es una variable que puede ser instanciada sólo a valores de un conjunto finito.

- Una variable de dominio entero es una variable dominio cuyo dominio contiene sólo números enteros. Estas son las únicas variables que son aceptadas en los términos lineales. El dominio finito de enteros se escribe como Min..Max, con Min≤Max, que representa el conjunto {Min, Min+1,...,Max}.
- Un término lineal es una combinación entera lineal de variables de dominio entero. Los términos lineales pueden escribirse incluso en forma no canónica. Por ejemplo (este ejemplo corresponde al puzzle "send more money" [Brisset, 94]):

$$1000\times S+100\times E+10\times N+D + 1000\times M+100\times O+10\times R+E \#= \#=10000\times M+1000\times O+100\times N+10\times E+Y$$

Estas restricciones se resuelven de manera eficiente razonando con los valores de las variables máximos y mínimos [Hentenryck, 89]. Este hecho se muestra en el siguiente ejemplo: Dadas las variables X, Y y Z cuyos dominios son {1,2,...,10} y la restricción

$$2\times X+3\times Y \#< Z,$$

es posible deducir

$$2\times X+3\times Y \#< 8,$$

ya que el valor máximo para Z es 10. Teniendo en cuenta que el valor mínimo para Y es 1, deducimos que

$$2\times X+3\#< 8$$

y entonces

$$2\times X\#< 5$$

Con lo que el valor de X se reduce a {1,2}.
De forma similar, teniendo en cuenta que el valor mínimo de X es 1, se obtiene que

$$3\times Y\#< 6$$

con los que el valor de Y se reduce a {1}.
Razonando con los valores mínimos de X e Y tenemos que

$$2\times 1+3\times 1+2 \#< Z,$$

por lo que

$$7 \#< Z$$

es decir, el dominio de Z se reduce a {8,9,10}.
Si se intenta procesar un término no lineal, este término se "duerme" hasta que se convierta en lineal.
CLP(FD) se proporciona automáticamente con la biblioteca de dominios finitos (fd.pl) que está incluida como una extensión de ECLiPSe [Brisset, 94] (en otros sistemas basados en PROLOG, como Sisctus PROLOG u Oz, también se incluyen bibliotecas que manejan CLP(FD)).
Para cargar la biblioteca se escribe al principio del módulo:

```
:-use_module(library(fd)).
```

Los predicados predefinidos de esta biblioteca más utilizados para la resolución de problemas son los siguientes:

- **Variables :: Dominio**

Este predicado asocia un dominio a una variable de dominio finito o a un conjunto de variables de dominio finito.
Ejemplos:
1. [X,Y,Z]::1..10
2. Lista=[S,E,N,D,M,O,R,Y], Lista::0..9

- **T1#=T2, T1##T2, T1#<T2, T1#<=T2, T1#>T2, T1#>=T2**

Este conjunto de predicados comprueban si el valor del término lineal T1 es igual, distinto, menor que, menor o igual que, mayor que y mayor o igual que, el valor del término lineal T2, respectivamente.

- **alldistinc(Lista)**

Este predicado predefinido asegura que todos los elementos de la lista (que pueden ser variables dominio o términos atómicos) son distintos dos a dos.

- **indomain(VarDominio)**

Este predicado instancia la variable dominio VarDominio a elementos de su dominio. Haciendo uso de backtracking se instancia a todos los valores del dominio. Este predicado se utiliza para asignar valores a la variable dominio que sean consistentes con las restricciones impuestas sobre dicha variable.

- **maxdomain(VarDominio,Max), mindomain(VarDominio,Min)**

Estos predicados predefinidos devuelven en Max y Min los valores máximo y mínimo del dominio de la variable de dominio entero VarDominio, respectivamente.

8.5. Ejemplos de resolución de problemas usando CLP(FD).

El ejemplo anterior, cuya restricción era 2×X+3×Y+2 #< Z puede codificarse en ECLiPSe utilizando la biblioteca fd.pl de la siguiente forma:

Ejemplo 1:

```
:-use_module(library(fd)).
ejemplo(X,Y,Z):-
     [X,Y,Z]::1..10,
     2×X+3×Y+2 #< Z,
     indomain(X),
     indomain(Y),
     indomain(Z).
```

Suponiendo que el código anterior está en un fichero cuyo nombre es NomFichero, se compilaría en ECLiPSe:

```
     [eclipse 1]: [NomFichero].
```

Y se ejecutaría con la pregunta:

```
[eclipse 2]: ejemplo(X,Y,Z).
```

que proporcionaría el resultado explicado previamente (X={1}, Y={1,2}, Z={8,9,10}). Indomain proporcionará todos los valores consistentes con las restricciones impuestas a la variable, por medio del backtracking.
La estructura estándar de la solución a problemas que pueden ser resueltos utilizando variables de dominio finito es la siguiente:

Ejemplo 2: El problema de las N reinas.

El enunciado del problema, la definición de variables y dominios y el planteamiento de las restricciones ya se comentó en la sección 2.2 de este capítulo. Vamos a ver ahora la solución utilizando la biblioteca fd.pl (este ejemplo ha sido sacado de [Brisset, 94]).

```
:-use_module(library(fd)).

nreinas(N,Lista):-
     hacer_lista(N,Lista),    % Crea una lista con N
variables
Lista::1..N,
     restricciones(Lista),
     labeling(Lista).

hacer_lista(0,[]):-!.
hacer_lista(N,[_|Resto]):-
     N1 is N-1,
     hacer_lista(N1,Resto).

restriciones([]).
restriciones([X|R]):-
     a_salvo(X,R,1),
     restricciones(R).

a_salvo(X,[Y|R],K):-
     no_ataque(X,Y,K),
     K1 is K+1,
     a_salvo(X,R,K1).

no_ataque(X,Y,K):-
     X≠≠Y,
     X+K≠≠Y,
     X-K≠≠Y.

labeling([]).
labeling([X|R]):-
     indomain(X),
     labeling(R).
```

En la figura 3 puede verse el árbol de búsqueda para el problema de las N-reinas, con N=4, utilizando la solución anterior.

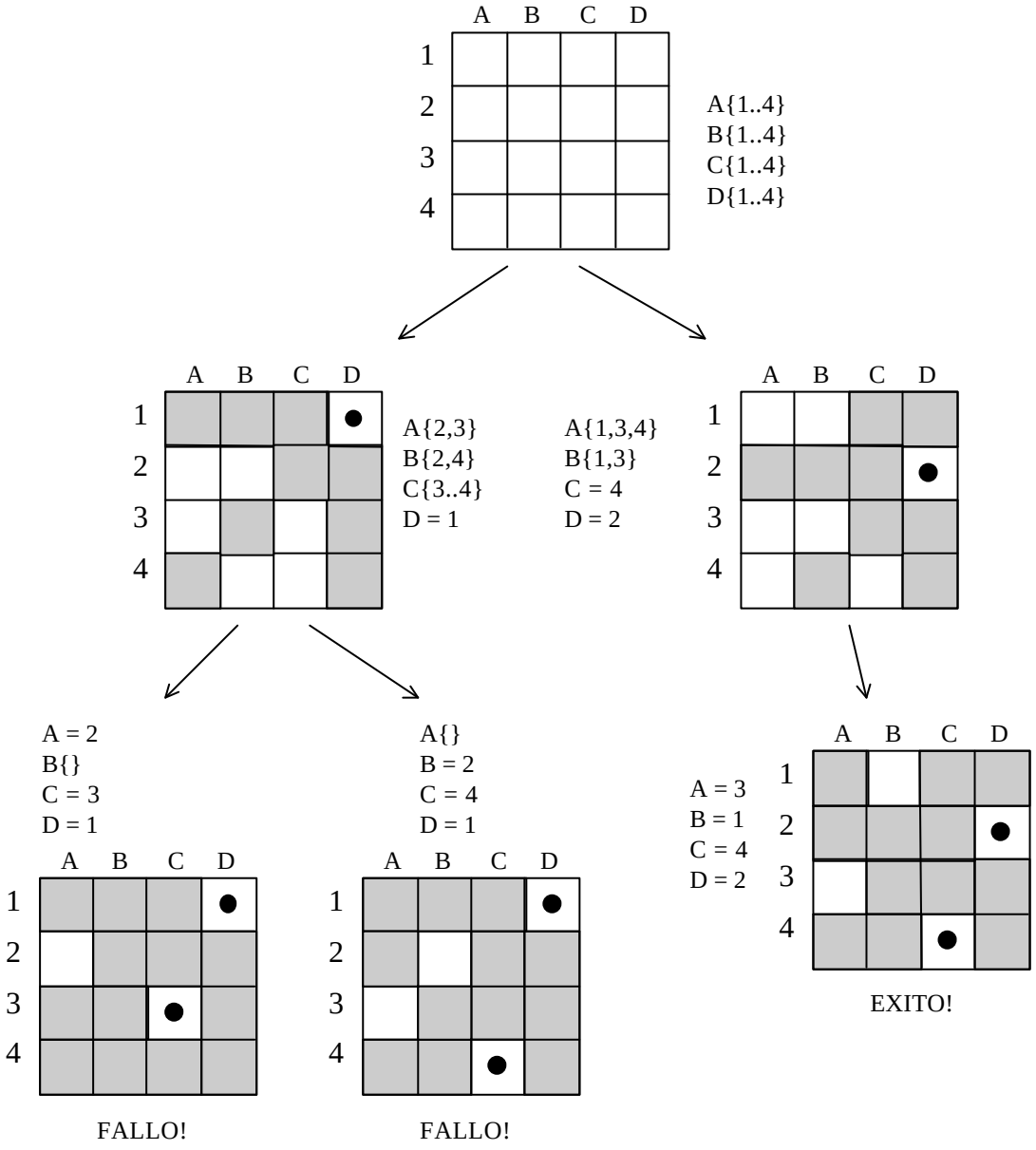


Figura 8.3. Árbol de búsqueda para el problema de la 4-reinas utilizando CLP(FD).

Utilizando el backtracking estándar la solución sería:

posicion(1).
posicion(2).
posicion(3).
posicion(4).
% Etc. tantos hechos como reinas hubiera.

/* Si sólo hay 4 reinas, la llamada se realiza mediante reinas([A,B,C,D]) */
reinas([]).

```
reinas([X|L]):-
     reinas(L),
     posicion(X),
     comprueba(X,L,0).
```

% Comprobación de que las reinas no se amenacen entre sí

```
comprueba(_,[],_).
comprueba(X,[X1|L],N):-
     N1 is N+1,
     X =\= X1,      % Distinta fila
     X =\= X1+N1,   % Distinta diagonal superior
     X =\= X1-N1,   % Distinta diagonal inferior
     comprueba(X,L,N1).
```

En la figura 4 puede verse el árbol de búsqueda para el problema de las N-reinas, con N=4, utilizando el backtracking estándar.

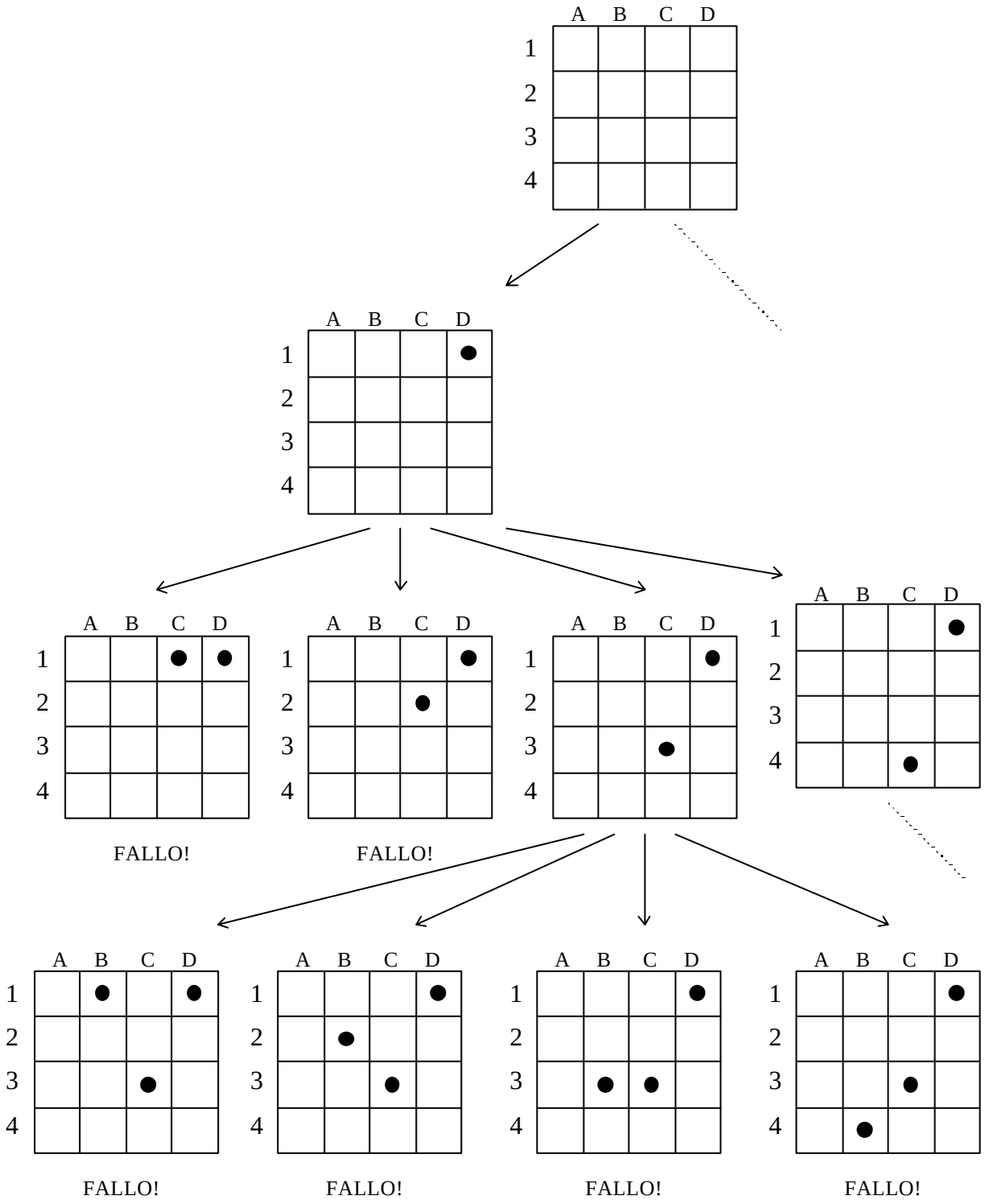


Figura 8.4. Árbol de búsqueda para el problema de la 4-reinas utilizando backtracking estándar.

Ejemplo 3: El problema de colorear mapas.

El problema de coloreado de mapas es asignar cada país de un mapa un color de tal forma que no haya vecinos que estén pintados del mismo color.
Asumimos que un mapa está especificado por la relación de vecindad

```
vecino(Comarca,Vecinos)
```

Donde `Vecinos` es la lista de comarcas limítrofes a `Comarca`. Así el mapa de la provincia de Castellón (figura 8.5) se podría representar como: (ordenado alfabéticamente)

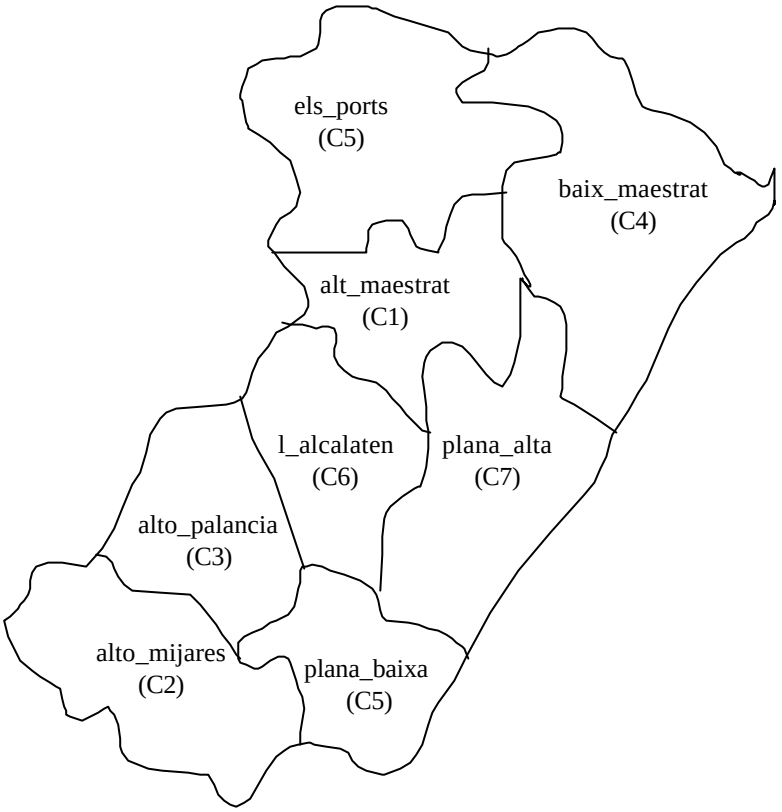


Figura 8.5. Mapa de las comarcas de Castellón.

```
vecino(alt_maestrat, [baix_maestrat, els_ports,
l_alcalaten, plana_alta]).
vecino(alto_mijares, [alto_palancia, plana_baixa]).
vecino(alto_palancia,
[alto_mijares,l_alcalaten,plana_baixa]).
vecino(baix_maestrat, [alt_maestrat, els_ports,
plana_alta]).
vecino(els_ports, [alt_maestrat, baix_maestrat]).
vecino(l_alcalaten, [alt_maestrat, alto_palancia,
plana_alta, plana_baixa]).
vecino(plana_alta, [alt_maestrat, baix_maestrat,
l_alcalaten, plana_baixa]).
vecino(plana_baixa, [alto_mijares, alto_palancia,
l_alcalaten, plana_alta]).
```

Solución 1:
La solución se representará de la forma

```
:- use_module(library(fd)).

Colores(L):-
          L=[C1,C2,C3,C4,C5,C6,C7,C8],
          L :: [amarillo, azul, rojo, verde],
          C5##C4, C5##C1,
          C4##C1, C4##C7,
          C1##C6, C1##C7,
          C6##C7, C6##C8, C6##C3,
          C7##C8,
          C3##C8, C3##C2,
          C8##C2,
          labeling(L).

labeling(L):-
          indomain(X),
          labeling(L).

labeling([]).
```

Solución 2: (sin utilizar dominios finitos)
La otra solución se representará de la forma

```
Comarca/Color
```

Que especifica el color de cada comarca en el mapa. Para un mapa dado, se fijan los nombres de las comarcas primero; y el problema es encontrar los valores para los colores. Así, para Castellón, el problema es encontrar una instancia adecuada a las variable `C1`, `C2`, `C3`, etc. En la lista:

```
[alt_maestrat/C1, alto_mijares/C2, alto_palancia/C3,
baix_maestrat/C4, els_ports/C5, l_alcalaten/C6,
plana_alta/C7, plana_baixa/C8]
```

Definimos el predicado

```
colores(Listadecoloresdecomarcas).
```

Que se cumplirá si `Listadecoloresdecomarcas` satisface el mapa de restricciones de colores con respecto a la relación vecino. Supongamos que tenemos suficiente con cuatro colores y sean los cuatro colores `amarillo`, `azul`, `rojo` y `verde`. La condición de no tener dos vecinos pintados del mismo color se puede expresar en PROLOG por:

```
colores([]).
colores([Comarca/Color | _ ] :-
     colores (_),
     miembro(Color, [amarillo, azul, rojo, verde]),
     not (miembro(Comarca/Color, _ ), vecindad (Comarca,
Comarca1)).
```

```
vecindad(Comarca, Comarca1) :-
     vecino(Comarca,Vecinos),
     miembro(Comarca1,Vecinos).
```

Este programa funciona bien para mapas con pocos países, pero sin embargo puede ser problemático para mapas grandes. Asumiendo que el predicado **`setof`** está disponible, un intento de colorear Castellón puede ser el siguiente. Primero definimos una relación auxiliar:

```
     pais(C):- vecino(C,_).
```

Entonces la pregunta para colorear Castellón se puede formular como:

```
 setof(Comarca/Color,pais(Comarca),Listadecoloresdecomarcas
),
     colores(Listadecoloresdecomarcas).
```

Ejercicio 8.1

Reescribir la solución anterior para una versión más simple del mismo problema, las 8-reinas, donde no es necesario un predicado para construir la lista.

Ejercicio 8.2

En una fiesta de amigos se quiere preparar un baile divertido y juntar parejas con miembros no necesariamente de distinto sexo, pero sí que cumplan las siguientes relaciones:

- Si los dos miembros de la pareja son de la misma altura tienen que tener distinto color de ojos.
- Si los dos miembros de la pareja tienen el mismo color de ojos, no tienen que llevar los dos gafas y tienen que tener distinto color de pelo.
- Si los dos miembros de la pareja tienen el mismo color de pelo, tienen que tener distinta altura (al menos 10 cm de diferencia).

Implementar en PROLOG los predicados necesarios para conseguir las parejas que cumplan las anteriores relaciones, suponiendo que los datos de las personas se almacenan como hechos de la base de datos PROLOG con el siguiente formato:

```
persona(Nombre,Altura,Color_pelo,Color_ojos,Lleva_gafas).
```

Un ejemplo puede ser el siguiente:

```
persona(maria,70,negro,marrones,no_gafas).
```

Ejercicio 8.3

Dada una habitación rectangular, escribir un algoritmo que permita obtener todas las distribuciones de muebles que cumplan las siguientes restricciones:

- La ventana (V) está delante de la puerta (P).
- La cama (C) nos gustaría tenerla en la pared adyacente a la de la ventana.
- En la misma pared que está la cama es conveniente tener la mesita de noche (MN).
- No cabe el armario (A) en la misma pared en donde están la cama, la ventana ni la puerta.
- La mesa de estudio (M) necesita toda una pared, es decir, no puede compartir la pared con otro mueble, ni estar en la misma pared que la puerta.
- El cuadro (CU) puede estar en cualquier pared excepto donde esté el armario.

Ejercicio 8.4

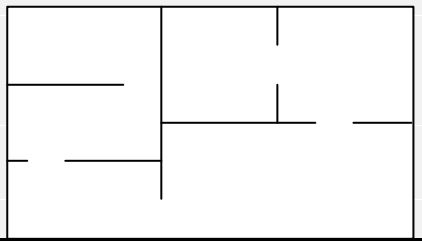
Tenemos 4 regalos (coche, casa, viaje, barco) a repartir entre 6 personas (Pepe, Juan, Rafael, Ana, Concha, Eva). Entre los 6, hay parejas que son incompatibles y nunca aceptarían tener el mismo regalo. La lista de parejas incompatibles es la siguiente:

Pepe es incompatible con las otras 5 personas
Juan es incompatible con Rafael, Ana y Concha
Rafael es incompatible con Concha y Eva
Ana es incompatible con Concha
Concha es incompatible con Eva

Escribir un programa PROLOG y el objetivo adecuado para obtener las asignaciones de regalos aceptables por las 6 personas.

Ejercicio 8.5

Para un experimento de robótica se necesita pintar cada una de las paredes de las habitaciones que aparecen en la figura de colores distintos a las paredes adyacentes. Se considera que una pared es adyacente a otra cuando las dos paredes forman una esquina o cuando están en la misma línea, separadas por una puerta. Escribir un programa PROLOG para que genere automáticamente posibles distribuciones de colores para las paredes.



Ejercicio 8.6

Existen dos grupos de piezas, las redondeadas, que comprenden el círculo, la elipse, el trébol y el corazón, y las puntiagudas, que comprenden el triángulo, el cuadrado, el rombo y el trapecio. Escribir un programa PROLOG que asigne de manera automática piezas a tres personas, A, B y C, tal que se cumplan las siguientes condiciones:

- Cada persona debe recibir 3 piezas.
- Al menos una pieza de cada persona debe ser original (ninguna otra persona tendrá esa pieza).

- Cada persona recibirá piezas redondeadas o puntiagudas, pero no una mezcla de ambas.
- Las personas A y C no están dispuestas a compartir ninguna figura.

Ejercicio 8.7

Dada una matriz de 3 4 elementos, escribir los predicados PROLOG necesarios para colorear cada elemento de la matriz de un color distinto al color de todos los elementos vecinos, utilizando el mínimo número de colores.

Ejercicio 8.8

Escribir un programa PROLOG para la composición artística automática sobre una matriz de 4 5 elementos, que se comporte de la siguiente manera. Los elementos de la periferia de la matriz serán pintados con colores fríos (azul o verde) y ningún elemento tendrá el mismo color que el elemento adyacente. Cada elemento de la periferia se supone que solamente tiene dos elementos adyancentes. Los elementos centrales de la matriz se pintarán con colores cálidos (rojo, marrón, naranja, rosa o magenta) y ningún elemento tendrá el mismo color que los elementos adyacentes. Se considerarán elementos adyacentes de un elemento central, cuyos índices en la matriz son (i,j), aquellos elementos de la matriz que no estén en la periferia y cuyos índices sean (i+1,j), (i-1,j), (i,j-1), (i,j+1), (i-1,j-1), (i-1,j+1), (i+1,j-1), (i+1,j+1).

Ejercicio 8.9

Los nuevos edificios del departamento de Informática están a punto de terminarse. Para asignar despachos a todos los profesores del departamento se pide escribir un programa PROLOG que tenga en cuenta las siguientes restricciones:

"Hay 7 despachos y 10 profesores, de los cuales 5 profesores tienen dedicación a tiempo completo y 5 profesores tienen dedicación a tiempo parcial. Los profesores a tiempo completo prefieren un despacho individual, excepto 2 profesores a tiempo completo que prefieren compartir despacho".

Ejercicio 8.10

Existen 5 tareas que se tienen que ejecutar como máximo en 30 segundos. Las tareas tienen además las siguientes restricciones temporales:

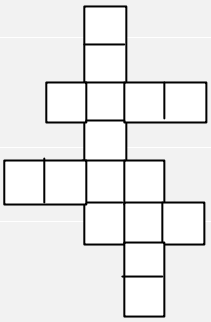
- La duración de la tarea 1 es de 2 segundos; la tarea 2 dura 3 segundos; la tarea 3 dura 2 segundos; la tarea 4 dura 10 segundos y la tarea 5 dura 15 segundos.
- La tarea 2 no puede comenzar antes de que la tarea 1 termine.
- la tarea 5 no puede comenzar antes de que las tareas 1, 2, 3 y 4 hayan terminado.
- Ninguna tarea puede comenzar antes de que las tareas con número inferior hayan comenzado.

Ejercicio 8.11

Tenemos una caja con 4 4 compartimentos iguales y hay que colocar figuras en ellos. Se dispone de 3 tipos de figuras: esfera, pirámide y cubo. La distribución que se desea no permite que la misma figura quede a los lados, aunque sí en la diagonal. Escribir los predicados PROLOG que sean necesarios para resolver el problema.

Ejercicio 8.12

Se quiere rellenar el crucigrama que aparece en la figura con las siguientes palabras: CAMISA, AMOR, CASA, ALMA, ALA. Escribir los predicados PROLOG necesarios para resolver el problema.



8.6. Bibliografía

[Brisset, 94] Brisset, P. et al., "ECLiPSe 3.4. ECRC Common Logic Programming System. Extension User Manual", European Computer-Industry Research Center, 1994.
(Este es el manual de usuario que explica los predicados predefinidos de la biblioteca de dominios finitos de ECLiPSe y ejemplos de su utilización).

[Escrig, 98] Escrig M.T., Toledo F. "Qualitative Spatial Reasoning: Theory and Practice. Application to robot navigation", in Frontiers in Artificial Intelligence and Applications Series, IOS Press, Vol. 47, ISBN 90 5199 412 5, 1998. (La exposición de este tema ha sido inspirada en el capítulo 3 de este libro).

[Fruehwirth, 90] Fruehwirth, T., "Constraint Logic Programming An Overview". CD-TR 90/8, Technische Universität Wien. Institut für Informationssysteme, 1990. (Este es un informe técnico que incluye un estudio sobre el estado del arte en programación lógica basada en restricciones).

[Fruehwirth, 93] Fruehwirth, T., Herold, A., Küchenhoff, V., Le Provost, T., Lim, P., Monfroy, E., Wallace, M., "Constraint Logic Programming —An Informal Introduction". Technical Report ECRC-93-5. European Computer-Industry Research Center, 1993. (Este es un informe técnico que incluye una introducción informal muy buena a la programación lógica basada en restricciones).

[Hentenryck, 89] Hentenryck, P. V., Constraint Satisfaction in Logic Programming. The MIT Press, 1989. (Este libro es de consulta obligatoria para profundizar en el estudio de los algoritmos de anticipación (look-ahead)).

[Hentenryck, 91a] Hentenryck, P. V., "Constraint Logic Programming". The knowledge Engineering Review, Vol. 6:3, pag. 151-194, 1991. (En este artículo se introduce la programación lógica basada en restricciones).

[Hentenryck, 91b] Hentenryck, P. and Deville, Y., "Operational Semantics of Constraint Logic Programming over Finite Domains", in Proceedings of PLILP'91, Passau, Germany, 1991. (En este artículo se introduce la programación lógica basada en restricciones sobre dominios finitos).

[Kumar, 92] Kumar V. “Algorithms for constraint satisfaction problems: A survey”, AI Magazine, 13(1), 1992, pag. 32-44. (En este artículo se incluye un estudio del estado del arte en programación lógica basada en restricciones que ha sido utilizado para la escritura de este tema).

[Meseguer, 89] Meseguer P. “Constraint Satisfaction Problems: An Overview”, AICOM, Volume 2(1), 1989, pp. 3-17. (En este artículo, igual que en el anterior, se incluye un estudio del estado del arte en programación lógica basada en restricciones que también ha sido utilizado para la escritura de este tema).

[Nilson 94] Nilson, U., Matuszynski, J., Logic, Programming and Prolog. Ed John Wiley & Sons, 1994. (Capítulo 14)

[Van Hentenryck 89] Van Hentenryck, P, Constraint Satisfaction in Logic Programming. The MIT Press, 1989.

9. Desarrollo de un PROYECTO.

DESARROLLO

Para el desarrollo de esta parte de las prácticas se hará una presentación de algunos de los proyectos que se han propuesto a los alumnos. Para cada uno de ellos se presentarán los objetivos particulares perseguidos y la bibliografía específica recomendada (también se puede consultar la bibliografía específica de cada tema si el proyecto es una extensión de algún tema en concreto).

9.1. PROYECTO 1

Hacer una búsqueda bibliográfica (en varios libros) y plantear, resolver, y comentar ejercicios interesantes y originales de los temas de teoría que ayuden a comprender lo expuesto en las clases (indicando la referencia correspondiente). Indicar aquellos ejercicios cuya solución está planteada en el libro y los que no (hacer la mitad de cada).

9.1.1. Objetivos

Promover la búsqueda bibliográfica del alumno.
Ayudar a la comprensión de temas teóricos expuestos en clase

9.1.2. Bibliografía

Libre

9.2. PROYECTO 2

Hacer una búsqueda bibliográfica (en varios libros) y plantear, resolver, y comentar ejercicios interesantes y originales de cada uno de los temas tratados en las prácticas (indicando la referencia correspondiente). Indicar aquellos ejercicios cuya solución está planteada en el libro y los que no (hacer la mitad de cada).

Ejemplo 9.1:
Definición de append/4 (Solución planteada en [Nilson 94] (C10, pag. 186)

Solución 1: (utilizando append/3)

```
append(X,Y,Z,Rdo):- append(X,Y,Rdoint), append(Rdoint,Z,Rdo).
```

Solución 2: (sin utilizar append/3, más eficiente)

```
append([],[],Z,Z).
append([],[Cab|Y],Z,[Cab|Rdo]):- append([],Y,Z,Rdo).
```

```
append([Cab|X],Y,Z,[Cab|Rdo]):- append(X,Y,Z,Rdo).
```

9.2.1. Objetivos

Promover la búsqueda bibliográfica del alumno.
Ver soluciones a problemas en PROLOG.

9.2.2. Bibliografía

Libre

9.3. PROYECTO 3

Definir una base de datos que relaciones comidas con sus ingredientes. Además se definirá otra relación que describe los ingredientes que se disponen y otras dos relaciones:

`puedo_cocinar(Comida)` que se satisfará para una comida si se dispone de todos los ingredientes
`necesita_ingrediente(Comida,Ingrediente)` que se satisfará para una comida y un ingrediente si la comida contiene el ingrediente.

Añadir a la base de datos la cantidad disponible de cada ingrediente y también la cantidad de cada ingrediente necesaria para cada comida y modificar la relación `puedo_cocinar(Comida)` para que compruebe que se puede cocinar esa comida si se dispone de la cantidad necesaria de cada ingrediente.

9.3.1. Objetivos

Profundizar en el estudio de las bases de datos

9.3.2. Bibliografía

[Nilson 94] (Capítulo 6)

9.4. PROYECTO 4

Utilizar la base de datos de las familias del ejercicio 5.2, con la definición de todas las reglas que se piden en el ejercicio para generar una base de datos deductiva, utilizando los predicados predefinidos de la biblioteca “knowledge base” de ECLiPSe.
Realizar algunas consultas interesantes sobre esa base de datos que utilicen tanto información almacenada mediante hechos como información almacenada mediante reglas (incluso recursivas).

9.4.1. Objetivos

Introducirse en el estudio de las bases de datos deductivas

9.4.2. Bibliografía

(Una idea parecida, más simple está en [Nilson 94] (Capítulo 6))

[ECLiPSe 93] Manual de extensiones de ECLiPSe, capítulo sobre “knowledge base”.

9.5. PROYECTO 5

Escribir un programa PROLOG que traduzca frases simples del castellano al inglés y viceversa, utilizando un vocabulario reducido.

Pista: Escribir dos analizadores para cada idioma de manera que cada uno de ellos devuelva un árbolo de análisis. Escribir después un programa que haga las conversiones entre árboles de análisis.

9.5.1. Objetivos

Profundizar en la práctica de programación lógica y gramáticas.

9.5.2. Bibliografía

[Van Le 93]

9.5.3. Solución

```
%%%% Proyecto resuelto por el alumno Jordi Paraire (Febrero
2001) %%%%

oracion(S0,S,T0,T):- sintagma_nominal(NUM,_,PERS,S0,S1,T0,T1),
          sintagma_verbal(NUM,_,PERS,S1,S,T1,T).

oracion(S0,S,T0,T):- pronombre(NUM,GEN,PERS,S0,S1,T0,T1),
          sintagma_verbal(NUM,GEN,PERS,S1,S,T1,T).
%-------------------------------------------------------------

sintagma_nominal(NUM,GEN,PERS,S0,S,T0,T):-
determinante(NUM,GEN,PERS,S0,S1,T0,T1),
          nombre(NUM,GEN,S1,S,T1,T).

sintagma_nominal(NUM,GEN,PERS,S0,S,T0,T):-
determinante(NUM,GEN,PERS,S0,S1,T0,T1),
          nombre(NUM,GEN,S1,S2,T2,T),
             adjetivo(NUM,GEN,S2,S,T1,T2).

%-------------------------------------------------------------

sintagma_verbal(NUM,_,PERS,S0,S,T0,T):-
verbo(NUM,PERS,S0,S,T0,T).

sintagma_verbal(NUM,_,PERS,S0,S,T0,T):-
verbo(NUM,PERS,S0,S1,T0,T1),
          sintagma_nominal(_,_,_,S1,S,T1,T).
```

```
%-----------------------------------------------------------------

determinante(singular,masculino,tercera,[el|S],S,[the|T],T).

determinante(singular,femenino,tercera,[la|S],S,[the|T],T).

determinante(plural,masculino,tercera,[los|S],S,[the|T],T).

determinante(plural,femenino,tercera,[las|S],S,[the|T],T).

%-----------------------------------------------------------------

nombre(singular,masculino,[hombre|S],S,[man|T],T).

nombre(plural,masculino,[hombres|S],S,[men|T],T).

nombre(singular,femenino,[manzana|S],S,[apple|T],T).

nombre(plural,femenino,[manzanas|S],S,[apples|T],T).

%-----------------------------------------------------------------

verbo(singular,primera,[como|S],S,[eat|T],T).
verbo(plural,primera,[comemos|S],S,[eat|T],T).

verbo(singular,segunda,[comes|S],S,[eat|T],T).
verbo(plural,segunda,[comeis|S],S,[eat|T],T).

verbo(singular,tercera,[come|S],S,[eats|T],T).
verbo(plural,tercera,[comen|S],S,[eat|T],T).

%-----------------------------------------------------------------

adjetivo(singular,masculino,[pequenyo|S],S,[little|T],T).

adjetivo(singular,femenino,[pequenya|S],S,[little|T],T).

adjetivo(plural,masculino,[pequenyos|S],S,[little|T],T).

adjetivo(plural,femenino,[pequenyas|S],S,[little|T],T).

%-----------------------------------------------------------------

pronombre(singular,_,primera,[yo|S],S,[i|T],T).

pronombre(singular,_,segunda,[tu|S],S,[you|T],T).

pronombre(singular,masculino,tercera,[el|S],S,[he|T],T).

pronombre(singular,femenino,tercera,[ella|S],S,[she|T],T).
```

```
pronombre(plural,masculino,primera,[nosotros|S],S,[we|T],T).

pronombre(plural,masculino,segunda,[vosotros|S],S,[you|T],T).

pronombre(plural,masculino,tercera,[ellos|S],S,[they|T],T).

pronombre(plural,femenino,primera,[nosotras|S],S,[we|T],T).

pronombre(plural,femenino,segunda,[vosotras|S],S,[you|T],T).

pronombre(plural,femenino,tercera,[ellas|S],S,[they|T],T).
```

9.6. PROYECTO 6

Desarrollar un sistema experto que ayude en la decisión de tratamiento de enfermedades a un médico homeópata. La entrada al sistema serán los síntomas y las características de carácter del paciente. La salida será la medicación más adecuada al paciente.
Este proyecto puede ser tan complicado como el elumno desee. Si desea desarrollar solamente un proyecto para la asignatura, bastará con estudiar un libro sobre medicina homeopática e incluir un conjunto de reglas. Si se desea profundizar en el tema se visitará un médico homeópata para estudiar las necesidades reales en el desarrollo de este sistema experto. El resultado puede proponerse como proyecto final de carrera.

9.6.1. Objetivos

Profundizar en el estudio del diseño de sistemas expertos utilizando la programación lógica.

9.6.2. Bibliografía

[Van Le 93] (Capítulo 9)

9.7. PROYECTO 7

Realizar un estudio de la biblioteca paralela de ECLiPSe (peclise), comprendiendo el funcionamiento de todos sus predicados predefinidos.
Resolver el problema de multiplicar dos matrices de N N elementos enteros (para un N arbitrario) en PROLOG y su versión para PROLOG paralelo. Comparar la mejora en tiempo que se obtiene con la ejecución paralela.

9.7.1. Objetivos

Practicar con la programación paralela en PROLOG.

9.7.2. Bibliografía

[Nilson 94](Capítulo 12)

[ECLiPSe 95] Manual del usuario de ECLiPSe, Capítulo 4.

9.8. PROYECTO 8

Estudio de distintas técnicas de búsqueda, como un método de resolución de problemas: búsqueda primero en profundidad, en anchura, primero el mejor, el de mejor coste, la búsqueda con mejor camino, etc.
Resolución del siguiente problema utilizando la mejor de las anteriores técnicas de búsqueda: El tráfico en Castellón es bastante denso. Para ayudar a los turistas que llegan a la ciudad, la oficina de turismo te ha solicitado escribir un programa PROLOG para guiar a los conductores de turismos a encontrar el camino entre distintos lugares de la ciudad. El programa recibe puntos de llegada y partida y debe producir una lista de movimientos apropiados para los turismos, como indica el siguiente ejemplo:

Donde está usted ahora? Introduzca el cruce de la forma calle1 + calle2. >
colon + enmedio
Donde quiere ir? Introduzca el cruce de la forma calle1 + calle2.>
gobernador + mar
Desde el cruce colon + enmedio
ir por colon hasta el cruce colon + caballeros
ir por colon hasta colon + mayor
...
ir por cardona_vives hasta llegar a gobernador + mar.

9.8.1. Objetivos

Ampliar los conocimientos de cómo se pueden implementar en PROLOG técnicas de resolución de problemas.

9.8.2. Bibliografía

[Van Le 93] (Capítulo 7)

9.9. PROYECTO 9

Escribir un programa que simule un psiquiatra y que genere una respuesta basándose en palabras clave de la entrada (parecido al programa que cambiaba unas palabras de una frase por otras). Un diálogo típico podría ser el siguiente (las respuestas que da el programa están en cursiva):

¿Cuál es su problema?
Esto es demasiado trabajo.
¿Qué más considera como demasiado trabajo?
Escribir cartas.
Ya veo. Siga, por favor.
También lavar el coche de mi madre.
Cuénteme más cosas sobre su familia.
¿Por qué tendría que hacerlo?
¿Por qué tendría que qué?

Aquí las palabras clave adecuadas eran **esto es**, **madre** y **por que**. La falta de una palabra clave apropiada desencadena la respuesta ***Ya veo. Siga, por favor***.

9.9.1. Objetivos

Profundizar en la práctica de la programación lógica y gramáticas.

9.9.2. Bibliografía

[Van Lee 93]

Apendice A

Solución a una selección de problemas planteados

Este apéndice pretende que el estudiante pueda tener respuesta a muchos de los ejercicios planteados, buscando en algunos casos más de una posible solución. Las respuestas son mayoritariamente de los primeros capítulos (quizá donde el estudiante deba tener más referencias al comenzar a estudiar un nuevo lenguaje de programación), dejándole algunos ejercicios de los capítulos más avanzados sin resolver para poder autoevaluarse mejor.

CAPÍTULO 1

Ejercicio 1.1

a) La respuesta de PROLOG a la pregunta "los hijos que tiene Jaime" será `NO`, ya que esa información no está disponible en la base de datos.

b) La respuesta a la pregunta "¿quiénes son los padres de Jaime?" será `X=patricia`.

c) La respuesta a la pregunta "¿es Clara abuela de Patricia?" será `X=jose`.

d) La respuesta a la pregunta "¿quiénes son los nietos de Tomás?", será `X=jose`, `Y=patricia`, `Z=jaime`. Z es la variable que corresponde al nieto, pero PROLOG saca como respuesta todas las variables que se han instanciado en la comprobación

Ejercicio 1.2

a)
```
?-progenitor(X,patricia).
```

b)
```
?-progenitor(isabel,X).
```

c)
```
?-progenitor(X,isabel), progenitor(Y,X).
```

d)
```
?-progenitor(X,patricia), progenitor(Y,X), progenitor(Y,Z).
```

Ejercicio 1.3

a)
```
es_madre(X):- mujer(X), progenitor(X,Y).
```

b)
```
es_padre(X):- hombre(X), progenitor(X,Y).
```

c)
```
es_hijo(X):- hombre(X), progenitor(Y,X).
```

d)
```
hermana_de(X,Y):-  mujer(X),  progenitor(Z,X),  progenitor(Z,Y),
                   dif(X,Y).
```

e) `abuelo_de(X,Y):- hombre(X), progenitor(X,Y), progenitor(Y,Z).`
`abuela_de(X,Y):- mujer(X), progenitor(X,Y), progenitor(Y,Z).`

f) `hermanos(X,Y):- progenitor(Z,X), progenitor(Z,Y), dif(X,Y).`

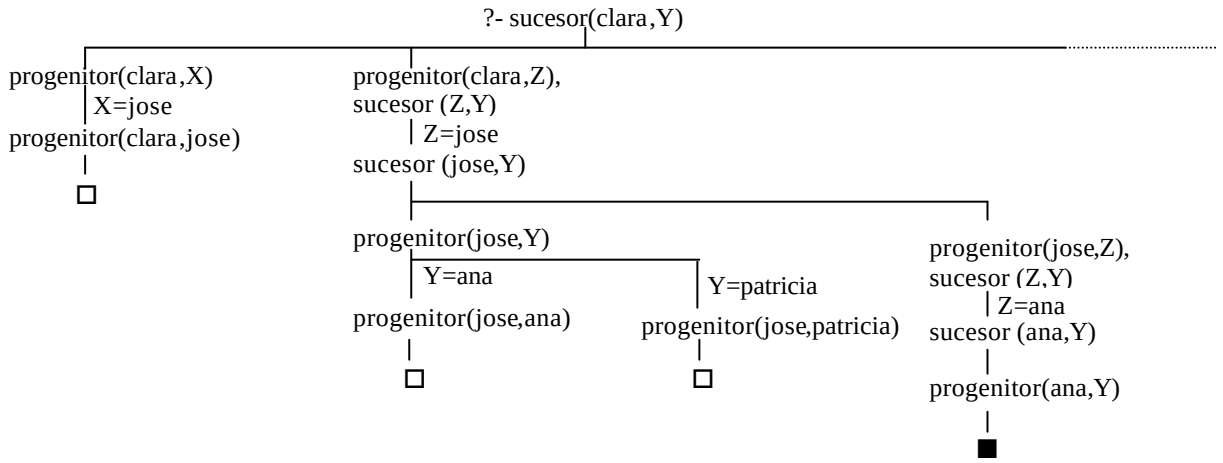
g) `tia(X,Y):- mujer(X), hermana(X,Z), progenitor(Z,Y).`

Ejercicio 1.4

a) `sucesor(X,Y):- predecesor(Y,X).`
Otra solución alternativa, que no utiliza la definición de predecesor podría ser la siguiente:

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

b) Si se desean obtener todos los sucesores de Clara, el árbol de derivación natural es el siguiente, que no está concluido por falta de espacio. La primera respuesta que PROLOG proporcionaría sería `X=jose`, la segunda `Z=jose`, `Y=ana`, la tercera `Z=jose`, `Y=patricia`. La instanciación de `Z=ana` no proporciona ninguna solución, porque Ana no es progenitor de nadie. Faltaría la instanciación de `Z=patricia`, para conocer los sucesores de Patricia.



c) Sí es una alternativa válida por la segunda regla.
`predecesor(X,Z):- progenitor(Y,Z), predecesor(X,Y).`
Pero en el predicado progenitor las variables están cambiadas

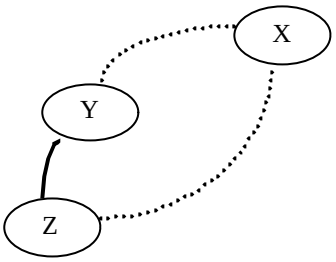


Diagrama para explicar la definición de:

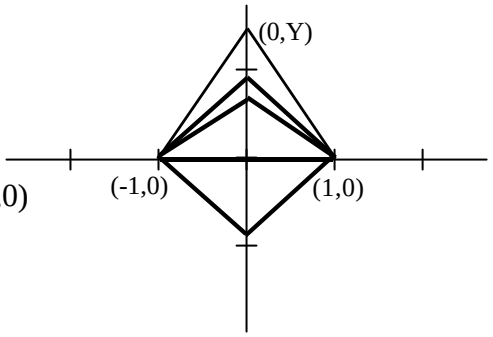
```
predecesor(X,Z):- progenitor(X,Z).
predecesor(X,Z):- progenitor(Y,Z), predecesor(X,Y).
```

Ejercicio 1.5

La operación tiene éxito y la instanciación es:

```
P1 = punto(-1,0)
P2 = punto(1,0)
P3 = punto(0,Y)
```

La familia de triángulos viene determinada por la variable Y.
Son los triángulos que tienen un lado que va del punto (-1,0) al (1,0) y el otro vértice en el eje de abcisas

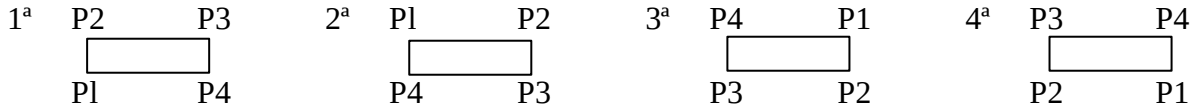


Ejercicio 1.6

```
segmento(punto(5,Y1),punto(5,Y2)).
```

Ejercicio 1.7

Las ordenaciones positivas de los vértices pueden ser:



Nota.- Las ordenaciones primera y tercera son equivalentes entre sí y lo mismo pasa con la segunda y la cuarta

Para las primera y tercera ordenaciones, el rectángulo será regular si coinciden las coordenadas X de los puntos Pl y P2, las coordenadas X de los puntos P3 y P4, las coordenadas Y de los puntos P1 y P4, y las coordenadas Y de los puntos P2 y P3.

```
regular(rectangulo(punto(Pxl,Pyl),punto(Pxl,Py2),punto(Px3,Py2),punto(Px3,Py1)).
```

Para las segunda y cuarta ordenaciones, el rectángulo será regular si coinciden las coordenadas X de los puntos Pl y P4, las coordenadas X de los puntos P2 y P3, las coordenadas Y de los puntos P1 y P2, y las coordenadas Y de los puntos P3 y P4.

```
regular(rectangulo(punto(Px1,Pyl),punto(Px2,Pyl),punto(Px2,Py3),punto(Px1,Py3)).
```

Ejercicio 1.8

a) `A=dos`

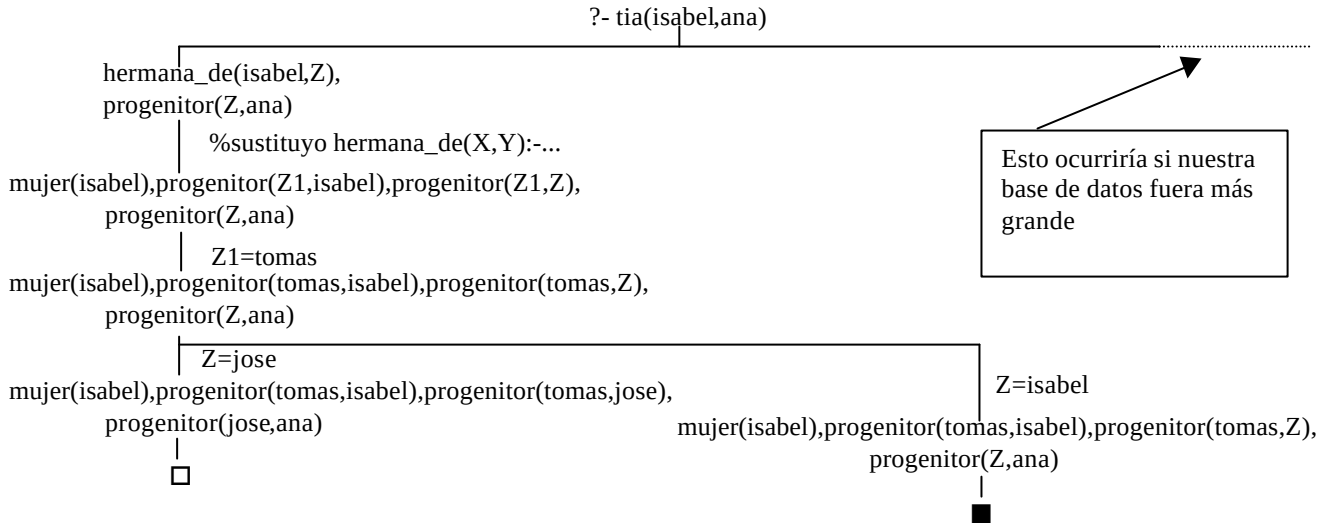
b) NO

c) C=uno

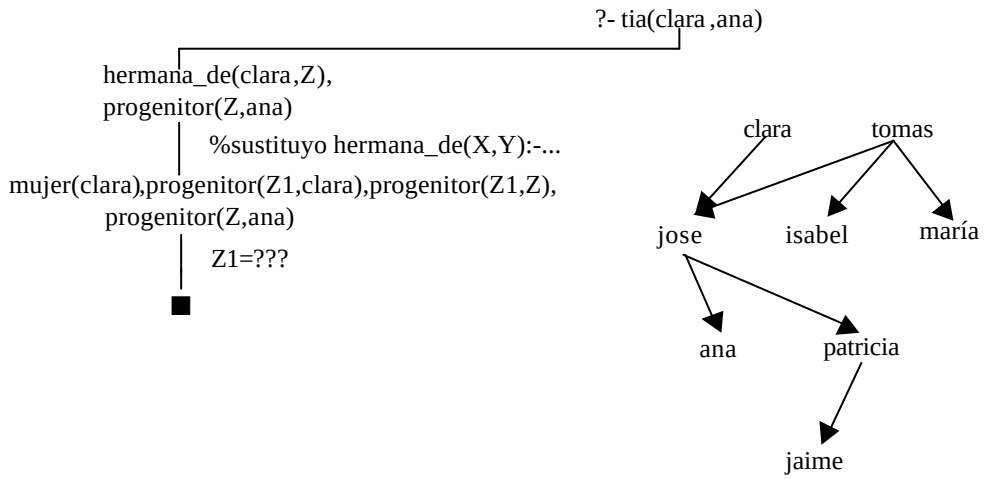
d)D=s(s(1)); D=s(s(s(s(s(l))))); D=s(s(s(s(s(s(s(l)))))));...

Ejercicio 1.9

a) Mostraremos el árbol de derivación natural



b) Mostraremos el árbol de derivación natural



c) No variarían.

CAPÍTULO 2

Ejercicio 2.1

a) Si nuestra lista es [a,b,c,d]

```
?-conc(L1,L2,[a,b,c,d]).
```

/* Obtendremos todas las descomposiciones de la lista [a,b,c,d] al teclear ";" (punto y coma) sucesivamente */

b) `?-L1=[a,b,c,d,z,z,e,z,z,z,f,g], conc(L2,[z,z,z|_],L1).`

c) `?-L1=[a,b,c,d,z,z,e,z,z,z,f,g], conc(L,[X,Y,Z],L1).`

d) `miembro(X,L):- conc(L1,[X|L2],L).`

e.l) `ultimo(Elem,Lista):- conc(_,[Elem],Lista).`

e.2)
```
ultimo(Elem,[Elem]).
   ultimo(Elem,[_|Lista]):- ultimo(Elem,Lista).
```

Ejercicio 2.2

```
borrar(X,[X|Resto],Resto).
borrar(X,[Y|Resto],[Y|Resto1]):- borrar(X,Resto,Resto1).
```

Ejercicio 2.3

Solución 1:
El hecho de escribir [X|L], ya añade el elemento X a la lista L.

Solución 2:
Si lo queremos hacer explícito: `add(X,L,[X|L]).`

Solución 3:
```
insertar(X,L,Lgrande):- borrar(X,Lgrande,L).
```

Ejercicio 2.4

```
consecutivos(X,Y,[X,Y|_]).
consecutivos(X,Y,[_|Resto]):- consecutivos(X,Y,Resto).
```

Ejercicio 2.5

% Ejemplo de palíndromo: [a,b,c,d,c,b,a]
```
palindromo(X):- inversa(X,X).
```

Ejercicio 2.6

a) Ejemplo l:
% El orden de las cláusulas da lo mismo en este caso
```
subconjunto([A|L1],L2):-
     miembro(A,L2),
     subconjunto(L1,L2).
subconjunto([],L).
```

Ejemplo 2:
% El orden de las cláusulas da lo mismo en este caso

```
subconjunto2(L,[A|L1]):-
     miembro(A,L),
     subconjunto2(L,Ll).
subconjunto2(L,[]).
```

b)
Solución l:

```
disjuntos(X,Y):-
     miembro(Z,X),
     not miembro(Z,Y).
```

Solución 2:

```
disjuntos([],_).
disjuntos([X|L1],L2):-
     disj(X,L2),
     disjuntos(L1,L2).

disj(_,[]).
disj (X, [_|L]):-disj (X,L).
```

Ejercicio 2.7

Solución l:

```
par([]).
par([X|Resto]):-
     impar(Resto).

impar([_]).
impar([X|Resto]):-
     par(Resto).
```

Solución 2:

```
par([]).
par([X,Y|Resto]):- par(Resto).

impar([X]).
impar([X,Y|Resto]):- impar(Resto).
```

Ejercicio 2.8

```
shift([Primero|Resto],L):- conc(Resto,[Primero],L).
```

Ejercicio 2.9

```
trasladar([],[]).
trasladar([X|Resto],[X1|Resto1]):-
     significa(X,X1),
     trasladar(Resto,Resto1).
```

Ejercicio 2.10

Solución l:

```
permutacion([],[]).
permutacion([X|Cola],L):-
     permutacion(Cola,L1),
     insertar(X,L1,L).
```

Solución 2:

```
permutacion2([],[]).
permutacion2(L,[X|P1):-
     borrar(X,L,L1),
     permutacion2(L1,P).
```

Solución 3:

```
permutacion3(Lista,[Cabeza|Cola]):-
     conc(V,[Cabeza|U],Lista),
     conc(V,U,W),
     permutacion3(W,Cola).
permutacion3([],[]).
```

Solución 4:

```
permutacion4([],[]).
permutacion4(Lista,[Cabeza|Cola]):-
     permutacion4(W,Cola),
     conc(V,U,W),
     conc(V,[Cabeza|U],Lista).
```

Solución 5:

```
permutacion5([],[]).
permutacion5([X|L1],[X|L2]):- permutacion5(L1,L2).
permutacion5([X|L1],[Y|L2]):-
     borrar(Y,L1,L3),
     permutacion5([X|L3],L2).
```

Ejercicio 2.11

a)

```
pos_y_neg([],0,0).
pos_y_neg([X|L],Npos,Nneg):-
     X>=0,
     pos_y_neg(L, N1pos,Nneg),
     Npos is N1pos + 1.
pos_y_neg([X|L],Npos,Nneg):-
     X<0,
     pos_y_neg(L, Npos,N1neg),
     Nneg is N1neg + 1.
```

```
par_impar([X],[X],[]).
par_impar([X,Y|L],[X|Li],[Y|Lp]):- par_impar(L,Li,Lp).
```

b)

```
pos_y_neg([],0,[],0,[]).
pos_y_neg([X|L],Npos,[X|Pos],Nneg,Neg):-
     X>=0,
     pos_y_neg(L,N1pos,Pos,Nneg,Neg),
     Npos is N1pos + 1.
pos_y_neg([X|L],Npos,Pos,Nneg,[X|Neg]):-
     X<0,
     pos_y_neg(L,Npos,Pos,N1neg,Neg),
     Nneg is N1neg + 1.
```

Ejercicio 2.12

```
par-impar([],[],[]).
par_impar([X],[X],[]).
par_impar([X,Y|L],[X|Li],[Y|Lp]):- par_impar(L,Li,Lp).
```

Ejercicio 2.13

```
min_max(L,Min,Max):- minimo(L,Min), maximo(L,Max).

minimo([X],X).
minimo([X|L],X):-
     minimo(L,Minl),
     X<Min1.
minimo([X|L],Min):-
     minimo(L,Min),
     X>=Min.

maximo([X],X).
maximo([X|L],X):-
     maximo(L,Max1),
     X>Max1.
maximo([X|L],Max):-
     maximo(L,Max),
     X=<Max.
```

Ejercicio 2.14

```
aparicion(_,[],0).
aparicion(X,[X|L],N):- aparicion(X,L,N1),
                       N is N1 + 1.
aparicion(X,[Y|L],N):- aparicion(X,L,N).
```

/* No hace falta poner X/=Y debido al orden de las reglas */

CAPÍTULO 3

Ejercicio 3.1

a)
Solución l:

```
interseccion([],L,[]).
interseccion([X|L1],L2,[X|L3]):-
     miembro(X,L2),
     !,
     interseccion(L1,L2,L3).
interseccion([_|L1],L2,L3):-
     interseccion(L1,L2,L3).
```

Solución 2:

```
interseccion([X|L1],L2,[X|L3]):-
     miembro(X,L2),
     !,
     interseccion(L1,L2,L3).
interseccion([_|L1],L2,L3):-
     !,
     interseccion(L1,L2,L3).
interseccion(_,_,[]). %Cláusula recogetodo
```

b)
Solución l:

```
union([],L,L).
union([X|L1],L2,[X|L3]):-
     not miembro(X,L2),
     !,
     union(L1,L2,L3).
union([X|L1],L2,L3):-
     union(L1,L2,L3).
```

Solución 2:

```
union([],L,L).
union([X|L1],L2,L3):-
     miembro(X,L2),
     !,
     union(L1,L2,L3).
union([X|L1],L2,[X|L3]):-
     union(L1,L2,L3).
```

c)

```
diferencia([],_,[]).
diferencia([A|L1],L2,L3):-
     miembro(A, L2), diferencia(L1,L2,L3).
diferencia(([A|L1],L2,[A|L3]):-
```

```
     not miembro(A,L2), diferencia(L1,L2,L3). % El corte está implícito en el not
```

Ejercicio 3.2

```
aplanar([Cabeza|Cola],Lplana):-
     aplanar(Cabeza,Cplana),
     aplanar(Cola,Coplana),
     conc(Cplana,Coplana,Lplana),
     !.
aplanar([],[]).
aplanar(X,[X]). % es para aplanar algo que no sea una lista
```

Ejercicio 3.3

```
cambiar_frase([],[]).
cambiar_frase([C|R],[C1|R1]):-
     cambiar(C,C1),
     !,
     cambiar_frase(R,Rl).
```

Ejercicio 3.4

```
miembro(X,[X|L]):-!.
miembro(X,[_|L]):- miembro(X,L).
```

Ejercicio 3.5

```
add(X,L,L):- miembro(X,L),!.
add(X,L,[X|L]).
```

Ejercicio 3.6

% Cada ";" (punto y coma) genera una nueva solución.

a) `X=l; X=2; NO.`

b) `X=1, Y=1; X=1, Y=2; X=2, Y=l; X=2, Y=2; NO.`

c) `X=1, Y=1; X=1, Y=2; NO.`

Ejercicio 3.7

```
clase(Numero,positivo):- Numero>0, !.
clase(0,cero):- !.
clase(Numero,negativo).
```

Ejercicio 3.8

```
divide([],[],[]).
```

```
divide([X|Numeros],[X|Positivos],Negativos):-
     X>=0, !, divide(Numeros,Positivos,Negativos).
divide([X|Numeros],Positivos,[X|Negativos]):-
     divide(Numeros,Positivos,Negativos).
```

Ejercicio 3.9

a)% tablas de la verdad

```
valor([not,0],1).
valor([not,1],0).

valor([0,and,0],0).
valor([0,and,1],0).
valor([1,and,0],0).
valor([1,and,1],1).
```

b)

```
valor([not|L],X):- valor(L|Y), cambiar(X,Y).

cambiar(0,1).
cambiar(1,0).
```

c)

```
valor([0,and,X|L],Y):- valor([0|L],Y).
valor([1,and,X|L],Y):- valor([X|L],Y).
```

Ejercicio 3.10

a)

```
desde(_,[],[]).
desde(X,[X|L],[X|L]):- !.
desde(X,[_|L],L1):- desde(X,L,L1).
```

b)

Solución 1

```
hasta(_,[],[]).
hasta(X,[X|L],[X]):- !.
hasta(X,[Y|L],[Y|L1]):- hasta(X,L,L1).
```

Solución 2

```
hasta(X,L,L1):- hasta(X,L,[],L1).

hasta(_,[],L,L).
hasta(X,[X|_],L1,Lr):-
     inv([X|L1],Lr),
     !. % Este corte es para que no saque más que una solución
hasta(X,[Y|L],L1,Lr):-
     hasta(X,L,[Y|L1],Lr).
```

```
inv(L,L1):- inv (L,[],L1).

inv([],L,L).
inv([X|L],L1,L2):- inv(L,[X|L1],L2).
```

c)

```
desde_hasta(X,Y,L,L1):-
     desde(X,L,L2),
     hasta(Y,L,L3),
     interseccion(L2,L3,L1).

interseccion([],L,[]).
interseccion([X|L1],L2,[X|L3]):-
     miembro(X,L2),
     !,
     interseccion(L1,L2,L3).
interseccion([_|L1],L2,L3):-
     interseccion(L1,L2,L3).

miembro(X,[X|_]).
miembro(X,[_|R]):-miembro(X,R).
```

Ejercicio 3.11

% Si hay elementos repetidos, sólo se escriben una vez

Solución 1:

```
union_ordenada([],L,L).
union_ordenada(L,[],L).
union_ordenada([X|L1],[X|L2],[X|L]):-
     !,
     union_ordenada(L1,L2,L).
union_ordenada([X|L1],[Y|L2],[X|L]):-
     X>Y,
     !,
     union_ordenada(([L1],[Y|L2],[L]).
union_ordenada([X|L1],[Y|L2],[X|L]):-
     X<Y,
     !,
     union_ordenada(([X|L1],[L2],[L]).
```

Solución 2:

```
union_ordenada(L1,L2,L3):- union_ordenada(L1,L2,[],L3).

union_ordenada([],L2,L3,L4):-
     inv(L3,L31),
     union(L31,L2,L4).
union_ordenada(L1,[],L3,L4):-
     inv(L3,L31),
```

```
     union(L31,L1,L4).
union_ordenada([X|L1],[X|L2],L3,L4):-
     union_ordenada(L1,L2,[X|L3],L4).
union_ordenada([X1|L1],[X2|L2],L3,L4):-
     X1<X2,
     union_ordenada(L1,[X2|L2],[X1|L3],L4).
union_ordenada([X1|L1],[X2|L2],L3,L4):-
     X1>X2,
     union_ordenada([X1|L1],L2,[X2|L3],L4).

union(L1,L2,L3):-
     inv(L1,L11),
     union1(L2,L11,L31),
     inv(L31,L3).

union1([],L,L).
union1([X|L1],L2,L3):-
     union1(L1,[X|L2],L3).
```

Solución 3:

```
union_ordenada(L1,L2,L3):- union(L1,L2,[],L3).

union([],L2,L3,L4):-
     conc(L3,L2,L4),
     !.
union(L1,[],L3,L4):-
     conc(L3,L1,L4),
     !.
union([X|L1],[X|L2],L3,L4):-
     conc(L3,X,L5),
     union(L1,L2,L5,L4),
     !.
union_ordenada([X|L1],[Y|L2],L3,L4):-
     X<Y,
     conc(L3,X,L5),
     union_ordenada(L1,[Y|L2],L5,L4),
     !.
union([X|L1],[Y|L2],L3,L4):-
     X>Y,
     conc(L3,Y,L5),
     union([X|L1],L2,L5,L4),
     !.
```

Ejercicio 3.12

```
lista_nombres(Q1,Q2,Lq,Lr):- desde_hasta(Q1,Q2,Lq,Lr).
```

CAPÍTULO 4

Ejercicio 4.1

```
irhacia([Xi,Yi],[Xmin,Ymin,Xmax,Ymax],Ruta):-
     ir([([Xi,Yi],[X,Y],Ruta),
     miembro_cuadrado([X,Y],[Xmin,Ymin,Xmax,Ymax]).

miembro_cuadrado([X,Y],[Xmin,Ymin,Xmax,Ymax]):-
     Xmin =< X,
     X =< Xmax,
     Ymin =< Y,
     Y =< Ymax.

ir([Salidax,Saliday],[Destinox,Destinoy],Ruta):-
     ir0([Salidax,Saliday],[Destinox,Destinoy],[],R),
     inv(R,Ruta).

ir0([Dx,Dy],[Dx,Dy],T,[[Dx,Dy]|T]).
ir0([Sx,Sy],[Dx,Dy],T,R):-
     nodolegal([Sx,Sy],T,[Sigx,Sigy]),
     ir0([Sigx,Sigy],[Dx,Dy],[[Sx,Sy]|T],R).

nodolegal([Sx,Sy],Camino,[Sigx,Sigy]):-
     (
     calle(Sx,Sy,Sigx,Sigy)
     ;
     calle(Sigx,Sigy,Sx,Sy)
     ),
     not miembro([Sigx,Sigy],Camino).
```

Ejercicio 4.2

Solución 1:
Los hechos los mostraremos a la vista del gráfico (pasar = p)

```
p(a,b)
p(b,c)
p(b,e)
p(c,d)
p(d,e)
p(e,f)
p(e,g)
```

para ir de una habitación a otra

```
ir(X,X,T).
ir(X,Y,T):-
          (
```

```
          p(X,Z),
          ;
          p(Z,X)
          ),
          not miembro(Z,T),
          ir(Z,Y,[Z|T]).
```

La pregunta para ir a la habitación que tiene teléfono, sería
`?- ir(a,X,[]), tienetelefono(X).`

Donde el predicado `tienetelefono(X)` indica que la habitación `X` tiene teléfono.

Solución 2:
Creamos una base de datos referida a los sitios por donde podemos ir, que a la vista del dibujo (suponiendo que los huecos son puertas por dande podemos pasar), sería:

```
recorrer(a,b)
recorrer(b,c)
recorrer(b,e)
recorrer(c,d)
recorrer(d,e)
recorrer(e,f)
recorrer(e,g)
```

Además para localizar si una habitación tiene telefono o no podríamos tener unos hechos de la forma

```
telefono(Habitacion,si).
telefono(Habitacion,no).
```

Que nos dirían si hay telefono o no en esa habitación. (Realmente sólo sería necesario el primero dado nuestro programa).

```
recorrido(Salida,Destino,Ruta):-
     telefono(Destino,si),
     recorre(Salida,Destino,Ruta).

recorre(X,X,[X]).
recorre(S1,S2,[S1|T]):-
     nodolegal(S1,S3,T),
     recorre(S3,S2,T).

nodolegal(A,B,Camino):-
     (
     recorrer(A,B)
     ;
     recorrer(B,a)
     ),
     not miembro(A,Camino).
```

```
/* La pregunta sería */
?- recorrido (a,Habitacion,Ruta).
```

a) Para que no busque en ciertas habitaciones, podemos hacer dos cosas, o eliminarlas de la base de conocimiento, o introducirlas en la lista por donde debe busca (recordamos que no entra dos veces en la misma habitación para evitar los bucles infinitos).

b) Para ampliar el programa (Por ejemplo en la solución 2) modificaríamos las reglas

```
recorre(X,X,[X]).
recorre(S1,S2,[S1|T]):-
     nodolegal(S1,S3,T),
     recorre(S3,S2,T).
```

Por esta otra:

```
recorre(X,X,[X]):-entrando(X).
recorre(S1,S2,[S1|T]):-
     nodolegal(S1,S3,T),
     entrando(S1),
     recorre(S3,S2,T).

entrando(X):- write('Entrando en la habitación '),
              write(X).
```

Y para encontrar el teléfono modificaríamos:

```
telefono(Habitacion,si).
```

Por esta otra

```
telefono(Habitacion,si):-
           write ('He encontrado el teléfono en la habitación '),
           write(Habitacion).
```

c)Si que saldrían caminos alternativos, para evitarlo hay que "cortar" cuando se acabe de encontrar la solución, es decir, modificar:

```
recorre(X,X,[X]).
```

Por esta otra:

```
recorre(X,X,[X]):-!.
```

d) Por último el orden de la búsqueda en las habitaciones lo determina el orden que éstas tengan en la base de datos.

Ejercicio 4.3

```
/* Crear un nuevo átomo que empiece con la raíz dada y acabe con
un número único */

gensim(Raiz,Atomo):-
```

```
                get_num(Raiz,Num),
                name(Raiz,Nombre1),
                nombre_entero(Num,Nombre2),
                append(Nombre1,Nombre2,Nombre),
                name(Atomo,Nombre).

get_num(Raiz,Num):-
                retract(num_actual(Raiz,Num)),
                !,
                num is Num1 + 1,
                asserta(num_actual(Raiz,Num)).
get_num(Raiz,Num):-
                asserta(num_actual(Raiz,1)).

/* Transformar un entero en una lista de caracteres */

nombre_entero(Ent,Lista):-
                nombre_entero(Ent,[],Lista),.
nombre_entero(E,Hastaaqui,[C|Hastaaqui]):-
                I < 10,
                !,
                C is I + 48.  % 48 es el código ASCII que le corresponde al cero
nombre_entero(E,Hastaaqui,Lista):-
Mitadsuperior is E/10,
Mitadinferior is E mod 10,
C is Mitadinferior + 48,
nombre_entero(Mitadsuperior,[C|Hastaaqui],Lista).
```

CAPÍTULO 5

Ejercicio 5.1

```
%  Suministradores que suministran al menos una parte roja:

ejercicio51a(Snombre):-
     sumin(Scodigo,Snombre,_,_),
     partes(Pcodigo,_,roja,_,_),
     s_p(Scodigo,Pcodigo,_).

%  Suministradores que NO suministran la parte p2:

ejercicio51b(Snombre):-
     sumin(Scodigo,Snombre,_,_),
     not s_p(Scodigo,p2,_).

%  Suministradores que suministran al menos todas las partes que suministra s2 (pedro):

ejercicio51c(Snombre):-
```

```
     findall(Pcodigo,s_p(s2,Pcodigo,_),Lista_s2),
     sumin(Scodigo,Snombre,_,_),
     findall(Pcodigo,s_p(Scodigo,Pcodigo,_),Lista_otro),
     subcto_partes(Lista_s2,Lista_otro),
     Scodigo \= s2.
```

% Suministradores que sólo suministran piezas de Castellón:

```
ejercicio51d(Snombre):-
     findall(Pcodigo,partes(Pcodigo,_,_,_,castellon),Piezas_cs),
     sumin(Scodigo,Snombre,_,_),
     findall(Pcodigo,s_p(Scodigo,Pcodigo,_),Piezas_sum),
     subcto_partes(Piezas_sum,Piezas_cs),
     Piezas_sum \= [].   % Si no luis también nos lo devuelve.
```

% Nombre de las partes que han sido suministradas por todos los suministradores.

```
ejercicio51e(Pnombre):-
     partes(Pcodigo,Pnombre,_,_,_),
     findall(Scodigo,sumin(Scodigo,_,_,_),Lista_sum),
     findall(Scodigo1,s_p(Scodigo1,Pcodigo,_),Lista_sumpieza),
     subcto_partes(Lista_sum,Lista_sumpieza).
```

% Suministradores "al por mayor" (aquellos que suministran siempre >=400 piezas de cada parte).

```
ejercicio51f(Snombre):-
     sumin(Scodigo,Snombre,_,_),
     findall(Scodigo,menores_400(Scodigo),Lista_menor400),
     not miembro(Scodigo,Lista_menor400).

menores_400(Scodigo):-
     s_p(Scodigo,_,Cantidad),
     Cantidad<400.
```

% Nombre de los suministradores y cantidad total de partes que han suministrado.

```
ejercicio51g(Snombre,SumaPartes):-
     sumin(Scodigo,Snombre,_,_),
     findall(Cantidad,s_p(Scodigo,_,Cantidad),Lista),
     suma(Lista,SumaPartes).

suma([],SumaPartes):-
     SumaPartes is 0.
suma([A|Resto],SumaPartes):-
     suma(Resto,SumaResto),
     SumaPartes is SumaResto + A.
```

% Media de las cantidades totales que han suministrado todos suministradores.

```
ejercicio51h(Snombre,Media):-
     sumin(Scodigo,Snombre,_,_),
     findall(Cantidad,s_p(Scodigo,_,Cantidad),Lista),
     media(Lista,Media).

media(Lista,Media):-media1(Lista,Media,0,0).

media1([],0,0,0).
media1([],Media,Elems,Total):-
     Media is Total / Elems.
media1([A|Resto],Media,Elems,Total):-
     Elems1 is Elems+1,
     Total1 is Total+A,
     media1(Resto,Media1,Elems1,Total1),
     Media is Media1.
```

Ejercicio 5.3

```
crear_bd:-createdb('p5_db').

crear_relaciones:-
     sumin <=> [atom(scodigo,2,+),
     atom(snombre,12,-),
     integer(estatus,4,-,
     atom(ciudad,12,-)],
     partes <=> [atom(pcodigo,2,+),
     atom(pnombre,12,-),
     atom(color,12,-),
     integer(peso,4,-),
     atom(ciudad,12,-)],
     s_p <=> [atom(scodigo,2,+),
     atom(pcodigo,2,+),
     integer(cantidad,4,-)].

insertar_datos:-
     sumin <++ [
          [s1,juan,20,madrid],
          [s2,pedro,10,castellon],
          [s3,raquel,30,alicante],
          [s4,maria,20,valencia],
          [s5,luis,30,castellon] ],
     partes <++ [
          [p1,mesa,verde,20,castellon],
          [p2,silla,verde,6,castellon],
          [p3,armario,azul,60,alicante],
          [p4,sofa,amarillo,55,valencia],
          [p5,cama,marron,20,madrid],
          [p6,libreria,roja,70,castellon] ],
```

```
     s_p <++ [
          [s1,p1,300],  [s1,p2,200],  [s1,p3,400],  [s1,p4,300],
          [s1,p5,700], [s1,p6,100],
          [s2,p1,300], [s2,p2,400],
          [s3,p2,200], [s4,p2,200], [s4,p4,300], [s4,p5,400] ].
```

% Suministradores que suministran al menos una parte roja:

```
ejercicio53a:-
     result <=> [],!,
     result isr [scodigo] :^:
          (s_p :*: partes where partes^pcodigo==s_p^pcodigo and
          partes^color == roja),
     printrel(result).
```

% Suministradores que no suministran la parte 2

```
ejercicio53b:-
     sumis <=> [],sumis_p2 <=> [],result <=> [],!,
     sumis_p2 isr [scodigo] :^: s_p where pcodigo == p2,
     sumis isr [scodigo] :^: sumin,
     result isr sumis :-: sumis_p2,
     printrel(result).
```

% Suministradores que solo suministran partes de Castellon (s2,s3):

```
ejercicio53d:-
     sumis <=> [],result <=> [],sumis_nocs <=> [],!,
     sumis_nocs isr [scodigo] :^: ( partes :*: s_p
          where s_p^pcodigo == partes^pcodigo
          and partes^ciudad \== castellon ),
     sumis isr [scodigo] :^: s_p,
     result isr sumis :-: sumis_nocs,
     printrel(result).
```

% Suministradores "al por mayor" (ninguno exceptuando a s5) (que nunca suministran piezas en cantidades menores de 400 unidades).

```
ejercicio53f:-
     sumis <=> [],sumis_menos <=> [],result <=> [],!,
     sumis_menos isr [scodigo] :^: s_p where s_p^cantidad < 400,
     sumis isr [scodigo] :^: s_p,
     result isr sumis :-: sumis_menos,
     printrel(result).
```

% Suministradores y cantidad total de partes que han suministrado (suma):

```
consulta_1g:-
```

```
     cantidades <=> [ atom(scodigo,2,+),
          count( integer(cantidad,4,-) ) ],
     cantidades <++ [scodigo,cantidad] :^: s_p,
     printrel(cantidades).
```

CAPÍTULO 6

Ejercicio 6.1

```
% Ejemplo 1
oracion(Z):-
     conc(X,Y,Z),
     sintagma_nominal(X),
     sintagma_verbal(Y).

sintagma_nominal(Z):- conc(X,Y,Z), determinante(X),nombre(Y).

sintagma_verbal(Z):- conc(X,Y,Z), verbo(X), sintagma_nominal(Y).
sintagma_verbal(X):- verbo(X).
determinante([el]).
determinante([la]).
nombre([hombre]).
nombre([manzana]).
verbo([come]).
```

```
% Ejemplo 2

oracion(S0,S):- sintagma_nominal(S0,S1), sintagma_verbal(S1,S).
sintagma_nominal(S0,S):- determinante(S0,S1),nombre(S1,S).
sintagma_verbal(S0,S):- verbo(S0,S).
sintagma_verbal(S0,S):- verbo(S0,S1),sintagma_nominal(S1,S).
determinante([el|S],S).
determinante([la|S],S).
nombre([hombre|S],S).
nombre([manzana|S],S).
verbo([come|S],S).
```

Ejercicio 6.2

```
oracion1(S0,S):-
     sintagma_nominal(Num,_,S0,S1),
     sintagma_verbal(Num,S1,S).

sintagma_nominal(Num,Genero,S0,S):-
     determinante(Num,Genero,S0,S1),
     nombre(Num,Genero,S1,S).

sintagma_verbal(Num,S0,S):-
     verbo(Num,S0,S).
```

```
sintagma_verbal(Num,S0,S):-
     verbo(Num,S0,S1),
     sintagma_nominal(_,_,S1,S).


nombre(singular,masc,[hombre|S],S).
nombre(singular,fem,[manzana|S],S).
nombre(plural,masc,[hombres|S],S).
nombre(plural,fem,[manzanas|S],S).
nombre(singular,masc,[juan|S],S).
nombre(singular,masc,[hombre|S],S).
nombre(singular,fem,[maria|S],S).
nombre(singular,fem,[mujer|S],S).
verbo(singular,[ama|S],S).
verbo(plural,[aman|S],S).
verbo(singular,[come|S],S).
verbo(plural,[comen|S],S).
verbo(singular,[viene|S],S).
verbo(singular,[vive|S],S).
determinante(singular,masc,[el|S],S).
determinante(plural,masc,[los|S],S).
determinante(singular,fem,[la|S],S).
determinante(plural,fem,[las|S],S).
```

Ejercicio 6.4

```
% Se debe ampliar el diccionario léxico

determinante(singular,masc,[un|S],S).
determinante(singular,fem,[una|S],S).
determinante(plural,masc,[unos|S],S).
determinante(plural,fem,[unas|S],S).
determinante(singular,masc,[todo|S],S).
determinante(singular,fem,[toda|S],S).
preposicion([a|S],S).

frase(S):- oracion2(_,_,S,[]).

oracion2(Num,Genero,S0,S):-
     sintagma_nominal2(Num,Genero,S0,S1),
     sintagma_verbal2(Num,S1,S).

sintagma_nominal2(Num,Genero,S0,S):-
     determinante(Num,Genero,S0,S1),
     sujeto(Num,Genero,S1,S),!.
sintagma_nominal2(Num,Genero,S0,S):-
     sujeto(Num,Genero,S0,S),!.
```

```
sujeto(Num,Genero,S0,S):-
     nombre(Num,Genero,S0,S),!.
sujeto(Num,Genero,S0,S):-
     nombre(Num,Genero,S0,S1),
     que(Num,S1,S),!.

que(Num,[que|S0],S):-
     sintagma_verbal2(Num,S0,S).

sintagma_verbal2(Num,S0,S):-
     verbo(Num,S0,S1),
     compl_dir2(S1,S),!.
sintagma_verbal2(Num,S0,S):-
     compl_dir2(S0,S1),
     verbo(Num,S1,S),!.
sintagma_verbal2(Num,S0,S):-
     verbo(Num,S0,S),!.

compl_dir2(S0,S):-
     preposicion(S0,S1),
          sintagma_nominal2(_,_,S1,S),!.
compl_dir2(S0,S):-
     sintagma_nominal2(_,_,S0,S),!.
```

CAPÍTULO 7

Ejercicio 7.4

```
:-op(800,fx,si).
:-op(700,xfx,entonces).
:-op(300,xfy,o).
:-op(200,xfy,y).

:-op(100,xfx,:).

minimo(C1,C2,C3):-
      (
      C1>C2,
      C3 is C2
      ;
      C1=<C2,
      C3 is C1
      ).

maximo(C1,C2,C3):-
      (
      C1>C2,
      C3 is C1
      ;
      C1=<C2,
      C3 is C2
      ).
```

Ejercicio 7.5

```
:-op(800,fx,si).
:-op(700,xfx,entonces).
:-op(300,xfy,o).
:-op(200,xfy,y).

si pelo y da_leche
entonces
   mamifero.

si plumas y vuela y pone_huevos
entonces
   ave.

si mamifero y come_carne
entonces
   carnivoro.

si     mamifero     y     dientes_agudos     y     garras     y
ojos_que_miran_hacia_delante
entonces
   carnivoro.

si mamifero y pezunyas
entonces
   ungulado.

si carnivoro y color_leonado y manchas_oscuras
entonces
   onza.

si carnivoro y color_leonado y franjas_negras
entonces
   tigre.

si ungulado y patas_largas y cuello_largo y color_leonado
   y manchas_oscuras
entonces
   jirafa.

si ave y no_vuela y patas_largas y cuello_largo y blanca_y_negra
entonces
   avestruz.

si ave y no_vuela y nada y blanca_y_negra
entonces
```

```
   pinguino.

si ave y nada_bien
entonces
   albatros.
```

% Mecanismo de razonamiento encadenado hacia delante

```
hacia_delante:-
     findall(P,hecho(P),L),    % recoge todas las evidencias en una lista L
     hacia_delante(L).

hacia_delante(L):-
     (
     nuevo_hecho_derivado(L,P),
     !,
     write('Derivado: '), write(P), nl,
     append([P],L,L1),    %append/3 es el predicado predefinido
     hacia_delante(L1)    %correspondiente a concatenar
     ;
     write('No hay mas hechos')
     ).

nuevo_hecho_derivado(L,Conclusion):-
     si Condicion entonces Conclusion,
     not miembro(Conclusion, L),
     hecho_compuesto(L,Condicion).

hecho_compuesto(L,Condicion):-
     miembro(Condicion,L).    % Es un hecho simple, está incluido en la lista
hecho_compuesto(L,Cond1 y Cond2):-
     hecho_compuesto(L,Cond1),
     hecho_compuesto(L,Cond2).
hecho_compuesto(L,Cond1 o Cond2):-
     (
     hecho_compuesto(L,Cond1)
     ;
     hecho_compuesto(L,Cond2)
     ).
```

% Mecanismo de razonamiento encadenado hacia detrás.

```
     es_verdad(P):- hecho(P).
     es_verdad(P):-
          si Condicion entonces P,
          es_verdad(Condicion).
     es_verdad(P1 y P2):-
          es_verdad(P1),
          es_verdad(P2).
     es_verdad(P1 o P2):-
```

```
          (
          es_verdad(P1)
          ;
          es_verdad(P2)
     ).
```

a) Para saber qué animal es Estirada, necesitamos utilizar el mecanismo de razonamiento encadenado hacia delante con los siguientes hechos.

```
hecho(pelo).
hecho(rumia).
hecho(patas_largas).
hecho(cuello_largo).
hecho(color_leonado).
hecho(manchas_oscuras).
```

b) Para comprobar si Ligera es una onza, necesitamos utilizar el mecanismo de razonamiento encadenado hacia detrás con los siguientes hechos.

```
hecho(pelo).
hecho(dientes_agudos).
hecho(garras).
hecho(ojos_que_miran_hacia_delante).
hecho(color_leonado).
```

CAPÍTULO 8

Ejercicio 8.3

```
?-use_module(library(fd)).

distrmuebles(Lista):-
     Lista = [V,P,C,MN,A,M,CU],
     Lista :: 1..4,
     VP is V - P,abs(VP,Result1),Result1 = 2,
     VC is V - C,abs(VC,Result2),(Result2 = 1;Result2 = 3),
     C #= MN,
     A ## C,A ## V,A ## P,
     M ## A,M ## C,M ## P,
     CU ## A,
     labeling(Lista).

labeling([]).
labeling([X|Y]):-
     indomain(X),
     labeling(Y).
```

Ejercicio 8.4

```
?-use_module(library(fd)).
```

```
asigregalos(Lista):-
     Lista = [Pepe,Juan,Rafael,Ana,Concha,Eva],
     Lista :: [coche,casa,viaje,barco],
     Pepe ## Juan, Pepe ## Rafael, Pepe ## Ana, Pepe ## Concha, Pepe ## Eva,
     Juan ## Rafael, Juan ## Ana, Juan ## Concha,
     Rafael ## Concha, Rafael ## Eva,
     Ana ## Concha,
     Concha ## Eva,
     labeling(Lista).

labeling([]).
labeling([X|Y]):-
     indomain(X),
     labeling(Y).
```

Apendice B

Direcciones de interés en Internet

Este apéndice pretende que el estudiante pueda tener acceso a ciertas direcciones de Internet que le permitan ampliar conocimientos y contactar con direcciones interesantes.

DIRECCIONES DE INTERÉS EN INTERNET

FAQ "PROLOG Resource Guide":
http://www.cs.cmu.edu/Web/Groups/AI/html/faqs/lang/prolog/prg/top.html

CMU PROLOG Repository, con archivos de código PROLOG, technical reports, etc.:
http://www.cs.cmu.edu:8001/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/0.html

ISO standard for PROLOG ISO/IEC 13211-1 es accesible via ftp anónimo en:
ai.uga.edu:/pub/prolog.standard/

Richard O'Keefe's 1984 PROLOG standard draft es accesible via ftp anónimo en:
ftp.ecrc.de:/pub/eclipse/std/plstd.doc

The PROLOG 1000 Database mantenida por la *Association for Logic Programming* y el *PROLOG Vendors Group*, contiene información de más de 500 aplicaciones prácticas de PROLOG con más de 2 millones de líneas de código y es accesible via ftp anónimo en:
src.doc.ic.ac.uk:/packages/prolog-progs-db/prolog1000.v1.gz

Implementaciones PROLOG de algoritmos de satisfacción de restricciones pueden obtenerse en:
http://web.cs.city.ac.uk/archive/constraints/constraints.html
ftp.cs.ualberta.ca:/pub/ai/csp/csplib.tar.Z

Implementaciones PROLOG de algoritmos de aprendizaje pueden obtenerse por ftp anónimo en:
ftp.gmd.de:/gmd/mlt/ML-Program-Library/

Un *WAM tracer* es accesible por ftp anónimo en:
ftp.csd.uu.se:/pub/WAM-tracer/luther.tar.Z

Un emulador de la WAM es accesible por ftp anónimo en:
ftp.csd.uu.se:/pub/WAM-emulator/

Bibliografía de Programación con restricciones puede obtenerse via ftp anónimo en el servidor de la Universidad de Aarhus:
ftp.daimi.aau.dk:/pub/CLP/
june.cs.washington.edu:/pub/constraints/papers/

Bibliografía en BiBTeX de Conferencias de programación lógica es accesible via ftp anónimo en:
duck.dfki.uni-sb.de

Bibliografía en formato PostScript y BibTeX sobre integración de programación lógica y orientación a objetos, es accesible via ftp anónimo en:

menaik.cs.ualberta.ca:/pub/oolog/

ALGUNAS PÁGINAS WEB DE INTERÉS

NALP: North American Logic Programming.	http://www.als.com/nalp.html
Oxford Comlab	http://www.comlab.ox.ac.uk/archive/logic-prog.html
Jonathan Bowen	http://www.comlab.ox.ac.uk/archive/logic-prog.html
Peter Reintjes	http://www.watson.ibm.com/watson/logicpgm/
Peter Van Roy	http://ps-www.dfki.uni-sb.de/~vanroy/impltalk.html
Michael Ley	http://www.informatik.uni-trier.de/~ley/db/index.html (índices de Proceedings y Revistas de bases de datos y programación lógica)
Ken Bowen	http://www.als.com/nalp.html
M.M. Corsini	http://dept-info.labri.u-bordeaux.fr/~corsini/Public/Reports/abint-biblio.ps
M. Hermenegildo	http://www.clip.dia.fi.upm.es/~herme (M. Hermenegildo pertenece a la red europea de lógica COMPULOG, y desde esta página se tienen enlaces con el resto de nodos de la red).

NEWSGROUPS

comp.lang.prolog
comp.object.logic
sci.logic
comp.constraints, que mantiene un servidor ftp y una página Web accesibles en:
ftp.cs.city.ac.uk:/pub/constraints
http://web.cs.city.ac.uk/archive/constraints/constraints.html

LISTAS DE CORREO ELECTRÓNICO (MAILING LISTS)

- Sobre PROLOG y programación Lógica:
  prolog@sushi.stanford.edu
  prolog-hackers@sushi.stanford.edu

- Sobre Lambda PROLOG:
  prolog@cis.upenn.edu

- Sobre Mercury:
  mercury-announce@cs.mu.oz.au
  mercury-users@cs.mu.oz.au

- Del Electronic Journal of Functional and Logic Programming (EJFLP):
  subscriptions@ls5.informatik.uni-dortmund.de

- Sobre PDC-L:
  PDC-L@nic.surfnet.nl

- Sobre demostradores de teoremas:
  theorem-provers@ai.mit.edu

- Sobre Logica:
logic@cs.cornell.edu

- De la ALP (French Chapter):
prog-logique@irisa.fr

Bibliografía

[Bratko 94] Bratko, I., PROLOG. Programming for Artificial Intelligence. 2ª edición. Ed. Addison-Wesley, 1994.

[Bratko, 90] I. Bratko. PROLOG. Programming for Artificial Intelligence. Second Edition. Addison- Wesley. 1990.

[Brisset 94] P. Brisset et al., ECLiPSe 3.4 Extensions User Manual, European Computer-Industry Research Center, Munich, Germany, 1994.

[Clocksin 93] Clocksin, W.F., Mellish, C.S., Programación en PROLOG. Colección Ciencia Informática, 2ª edición. Editorial Gustavo Gili, S.A., 1993.

[Clocksin 97] Clocksin, W.F., Clause and effect: PROLOG programming for the working programmer. Ed. Springer, 1997.

[Covington, 94] M.A.Covington, Natural Language processing for PROLOG programmers. 1st edition. Prentice Hall Inc. 1994.

[Date, 93] C.J.Date, Introducción a los Sistemas de bases de datos. Volumen I. Quinta edición. Ed. Addison-Wesley Iberoamericana, 1993.

[Durkin 96] J. Durkin, Expert Systems. Design and development. Ed. Prentice-Hall, 1996.

[ECLiPSe, 94] ECLIPSE 3.4 (ECRC Common Logic Programming System). User Manual and ECLIPSE DataBase and Knowledge Base. July 1994.

[Escrig, 98] Escrig M.T., Toledo F. “Qualitative Spatial Reasoning: Theory and Practice. Application to robot navigation”, in Frontiers in Artificial Intelligence and Applications Series, IOS Press, Vol. 47, ISBN 90 5199 412 5, 1998. (La exposición de este tema ha sido inspirada en el capítulo 3 de este libro).

[Fruehwirth, 90] Fruehwirth, T., “Constraint Logic Programming An Overview”. CD-TR 90/8, Technische Universität Wien. Institut für Informationssysteme, 1990. (Este es un informe técnico que incluye un estudio sobre el estado del arte en programación lógica basada en restricciones).

[Fruehwirth, 93] Fruehwirth, T., Herold, A., Küchenhoff, V., Le Provost, T., Lim, P., Monfroy, E., Wallace, M., "Constraint Logic Programming —An Informal Introduction". Technical Report ECRC-93-5. European Computer-Industry Research Center, 1993. (Este es un informe técnico

que incluye una introducción informal muy buena a la programación lógica basada en restricciones).

[Giarratano & Riley 98] J.Giarratano, G.Riley, Expert Systems. Principles and programming. 3rd edition. Ed. PWS Publishing Company, 1998.

[Gray et al., 88] P.M.D.Gray, R.J.Lucas, editors. PROLOG and databases. Implementations and new directions. Ellis Horwood Series in AI. 1988.

[Hentenryck, 89] Hentenryck, P. V., Constraint Satisfaction in Logic Programming. The MIT Press, 1989. (Este libro es de consulta obligatoria para profundizar en el estudio de los algoritmos de anticipación (look-ahead)).

[Hentenryck, 91a] Hentenryck, P. V., "Constraint Logic Programming". The knowledge Engineering Review, Vol. 6:3, pag. 151-194, 1991. (En este artículo se introduce la programación lógica basada en restricciones).

[Hentenryck, 91b] Hentenryck, P. and Deville, Y., "Operational Semantics of Constraint Logic Programming over Finite Domains", in Proceedings of PLILP'91, Passau, Germany, 1991. (En este artículo se introduce la programación lógica basada en restricciones sobre dominios finitos).

[Jackson 90] P. Jackson, Introduction to expert systems. 2nd edition. Ed. Addison-Wesley, 1990.

[Konigsberger & Bruyn 90] Konigsberger, H., Bruyn, F., PROLOG from the beginning. McGraw-Hill book Company, 1990.

[Kumar, 92] Kumar V. “Algorithms for constraint satisfaction problems: A survey”, AI Magazine, 13(1), 1992, pag. 32-44. (En este artículo se incluye un estudio del estado del arte en programación lógica basada en restricciones que ha sido utilizado para la escritura de este tema).

[Kumer-Das, 92] S. Kumer-Das, Deductive databases and logic programming. Addison-Wesley publishing company, 1992.

[Lazarev,89] Lazarev. Why PROLOG? Justifying logic programming for practical applications. Prentice Hall, 1989.

[Lucas, 88] R. Lucas, Database Applications using PROLOG. Ellis Horwood books in Computer Science. Halsted Press: a division of Jonh Wiley & Sons, 1988.

[Matthews, 98] C. Matthews, An Introduction to Natural Language Processing through PROLOG. 1st edition. Computer Science, linguistics. Series learning about language. 1998.

[Meseguer, 89] Meseguer P. “Constraint Satisfaction Problems: An Overview”, AICOM, Volume 2(1), 1989, pp. 3-17. (En este artículo, igual que en el anterior,

se incluye un estudio del estado del arte en programación lógica basada en restricciones que también ha sido utilizado para la escritura de este tema).

[Mizoguchi, 91] F. Mizoguchi, editor. PROLOG and its applications. A Japanese perspective. Chapman & Hall Computing, 1991.

[Nilson, 94] Nilson, U., Matuszynski, J., Logic, Programming and Prolog. Ed John Wiley & Sons, 1994.

[Nussbaum, 92] M.Nussbaum, Building a deductive database. Ablex Publishing Corporation Norwood, New Jersey, 1992.

[O'Keefe 90] O'Keefe, R.A., The craft of PROLOG. The MIT Press, 1990.

[Pereira, 94] F.Pereira, S.M.Shieber, PROLOG and Natural-language Analysis. CSLI Publications, 1987.

[Silberschatz et al. 98] A. Silberschatz, H.F.Korth, S. Sudarshan, Fundamentos de bases de datos, Tercera edición. Ed. McGraw Hill, 1998.

[Starling & Shapiro 94] Starling, L., Shapiro, E., The art of PROLOG. Second edition. The MIT Press, 1994.

[Starling 90] Starling, L. editor, The practice of PROLOG. The MIT Press, 1990.

[Stobo 89] Stobo, J., Problem solving with PROLOG. Pitman publishing, 1989.

[Van Hentenryck, 89] Van Hentenryck, P, Constraint Satisfaction in Logic Programming. The MIT Press, 1989.

[Van Le, 93] Van Le, T., Techniques of PROLOG programming with implementation of logical negation and quantified goals. Ed. John Wiley & Sons, Inc., 1993.

[Walker 90] Walker, A., et. al., Knowledge Systems and Prolog: Developing Expert, Database, and Natural Languaje Systems. 2ª edición. Addison-Wesley, 1990.

[Walker 90] Walker, A., et. al., Knowledge Systems and Prolog: Developing Expert, Database, and Natural Languaje Systems. 2ª edición. Addison-Wesley, 1990.