Linguistics, Biology & Computation

TRIANGLE

LLENGUATGE, LITERATURA, COMPUTACIÓ
LANGUAGE, LITERATURE, COMPUTATION

2

Programación lógica

Veronica Dahl
Alejandro Javier García

[publicacions] urv

TRIANGLE 2

December 2010

Programación lógica

Veronica Dahl
Alejandro Javier García

LLENGUATGE, LITERATURA, COMPUTACIÓ
LANGUAGE, LITERATURE, COMPUTATION

[publicacions] urv
Tarragona, 2010

Edita: Publicacions URV

ISSN: 2013-939X
ISBN: 978-84-693-7385-9
DL: T-1492-2010

Per a més informació de la revista consulteu la pàgina
http://revista-triangle.blogspot.com/

Publicacions de la Universitat Rovira i Virgili:
Av. Catalunya, 35 - 43002 Tarragona
Tel. 977 558 474 - Fax: 977 558 393
www.urv.cat/publicacions
publicacions@urv.cat

Arola Editors: Polígon Francolí, parcel·la 3, nau 5 - 43006 Tarragona
Tel. 977 553 707 - Fax 977 542 721
arola@arolaeditors.com

Cossetània Edicions: C. de la Violeta, 6 - 43800 Valls
Tel. 977 602 591 - Fax 977 614 357
cossetania@cossetania.com

Programación lógica

Verónica Dahl

Chair of Excellence from the European Commision
GRLMC-Research Group Mathematical Linguistics
Rovira i Virgili University
43005 Tarragona, SPAIN

Professor, Department of Computer Science
Director, Logic and Functional Programming Group
Simon Fraser University
Burnaby, B.C., CANADA

email: veronica@cs.sfu.ca

Alejandro Javier García

Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

Laboratorio de Investigación y Desarrollo en Inteligencia Artificial (LIDIA)
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, ARGENTINA

email: ajg@cs.uns.edu.ar

Índice general

Parte I

Introducción a la programación lógica

1

Introducción

La programación lógica ofrece herramientas de programación que, en algunos casos, son únicas de este paradigma, y que permiten desarrollar aplicaciones de inteligencia artificial con suma facilidad y elegancia. La manera de pensar en los lenguajes de programación lógica es bastante diferente a la de los lenguajes de programación tradicionales. Debido a que la programación lógica nace de la lógica, que desde tiempo inmemorial intenta capturar el pensamiento humano, por un lado constituye una manera más natural de organizar nuestro pensamiento que la de los lenguajes tradicionales y, por el otro, requiere una mayor capacidad de abstracción. En lugar de pensar en términos de bajo nivel, como bits, bytes, asignaciones y repeticiones; la programación lógica nos permite pensar en términos de premisas y conclusiones, y representar un dominio de problemas mediante una serie de axiomas lógicos que serán usados por un demostrador automático para deducir la solución a un problema particular planteado frente a esa serie de axiomas. La mayoría de las aplicaciones de inteligencia artificial se desarrollan en este lenguaje por ese motivo.

Como se verá a continuación, en programación lógica es muy fácil representar y recuperar información en forma de conocimiento; además, se proveen herramientas para programar distintos mecanismos de razonamiento sobre ese conocimiento. En los últimos años, por otra parte, los intérpretes y compiladores de Prolog (el lenguaje más difundido de programación lógica) han desarrollado herramientas para combinar de una manera muy natural programas en Prolog con código en otros lenguajes de programación. Esta integración permite desarrollar sistemas complejos con herramientas adecuadas para cada parte del sistema.

Este texto pretende introducir los conceptos básicos e imprescindibles para poder programar con este paradigma. Dado que la exposición detallada de todas las herramientas que proporciona la Programación en Lógica requeriría de cientos de páginas, algunos temas sólo se mencionarán y luego se remitirá a la literatura para su tratamiento en profundidad. Afortunadamente existen en la literatura excelentes libros sobre programación lógica, como [1, 2, 4, 6, 9], por mencionar algunos. En el apéndice B se indicarán algunos libros de referencia para que el lector inquieto y curioso de profundizar en los temas introducidos en este texto pueda seguir descubriendo las posibilidades que brinda esta manera de programar.

2

Construcciones básicas: datos, reglas y preguntas

La sintaxis de los lenguajes de programación lógica es bastante minimalista. Solamente hay tres construcciones posibles: **datos**, **reglas** y **preguntas**. En este capítulo, se mostrará en detalle cada una de estas construcciones a través de ejemplos. Para ello se utilizará la notación estándar de Prolog, que es el más utilizado de los lenguajes de programación lógica.

A modo de introducción, la Figura 1 muestra un ejemplo clásico, donde hay tres datos, tres reglas y tres preguntas. Los datos `padre(juancito,pedro)`, `padre(pedro,luis)` y `padre(luis,alberto)` se utilizan para representar (respectivamente) la información: "*el padre de Juancito es Pedro*", "*el padre de Pedro es Luis*" y "*el padre de Luis es Alberto*".

```
padre(juancito,pedro).
padre(pedro,luis).
padre(luis,alberto).

abuelo(X,Y):- padre(X,Z), padre(Z,X).
hijo(UnPadre,UnHijo):- padre(UnHijo,UnPadre).
tiene_hijos(X):- hijo(X,_).

?-padre(juancito,pedro).
?-abuelo(juancito,X).
?-tiene_hijos(juancito).
```

Figura 1. Un ejemplo clásico con datos, reglas y preguntas en Prolog.

Las reglas que muestra la Figura 1 permiten definir (a partir de datos u otra regla) los conceptos de "*abuelo*", "*hijo*" y "*tiene hijos*". En particular, estas reglas tienen variables y, como se explicará más adelante, las variables se diferencian de otros elementos del lenguaje porque tienen una letra mayúscula inicial o el símbolo especial "`_`" (por ejemplo, `X`, `Y`, `Z`, `UnPadre`, `UnHijo` y `_` son variables). Como se mostrará en detalle en el capítulo 3, las reglas usan el símbolo "`:-`", que puede interpretarse como "si", y la coma ",", que denota una conjunción (y). La primera regla, por ejemplo, define el concepto de `abuelo`, usando la definición de `padre`, y expresa que "*el abuelo de un individuo `X` es un individuo `Y`, **si** es cierto que el padre de `X` es el individuo `Z`, **y** además el padre de `Z` es `Y`*". En este ejemplo, usando esta regla se podrá inferir que Luis es el abuelo de Juancito, usando los datos `padre(juancito,pedro)` y `padre(pedro,luis)`. La forma como se computan estas inferencias se explicará en detalle en el capítulo 3.

La segunda regla de la Figura 1 expresa que "*el hijo de un individuo `UnPadre` es `UnHijo`, si es el caso que el padre de `UnHijo` es `UnPadre`*". Luego, con esta regla se podrá inferir que "*el hijo de Alberto es Luis*", a partir del dato `padre(luis,alberto)`. Finalmente, la regla `tiene_hijos(X):-hijo(X,_)` expresa que "*un individuo `X` tiene hijos, si el hijo de `X` existe*". En este ejemplo, podrá inferirse que "*Pedro tiene hijos*", mientras que "*Juancito tiene hijos*" no puede deducirse de los datos, ya que no hay ningún individuo que tenga como padre a Juancito.

De esta forma, los datos nos permitirán representar información que consideramos cierta de manera incondicional, mientras que las reglas se utilizarán para inferir o deducir información a partir de los datos (o de otras reglas). Las preguntas, por su parte, permiten interrogar o consultar al programa para conocer si cierta información es cierta o no (*i.e.*, si puede inferirse a partir de los datos y las reglas del programa). Por ejemplo, la pregunta `?-padre(juancito,pedro)` (que representa "*¿Es el padre de Juancito, Pedro?*") tendrá una respuesta afirmativa, ya que hay un dato que lo "prueba", mientras que la pregunta `?-tiene_hijos(juancito)` tendrá una respuesta negativa. En el capítulo 2 se mostrará en detalle cómo se puede interrogar a un programa y qué ocurre cuando las preguntas tienen variables, como por ejemplo en `?-abuelo(juancito,X)`.

Además de los datos y las reglas que introduce un programador, en Prolog hay construcciones reservadas, que permitirán por ejemplo dar declaraciones importantes para el compilador o intérprete (estas se denominan "comandos"), o realizar operaciones auxiliares como leer e imprimir en

pantalla, podar el espacio de búsqueda, etc. A continuación se explicarán en detalle los datos, preguntas y reglas.

1. Datos

Los datos, también llamados **hechos o axiomas** (en inglés *facts*), son aserciones incondicionales. La Figura 2 muestra un programa Prolog completo y correcto, listo para ser *consultado*, cuyos axiomas (en forma de "datos") describen el dominio de problemas de una base de datos cosmológica. Aun sin conocimiento previo del lenguaje, es fácil incluso en una primera lectura interpretar la mayor parte de dichos axiomas: "Selene es la luna, Helios es el sol, Gaia es la tierra, es decir el mundo; Helios brilla, Selene también, Selene refleja a Helios, y Selene refleja a Helios sobre Gaia".

```
luna(selene).
sol(helios).
tierra(gaia).
mundo(gaia).

brilla(helios).
brilla(selene).
refleja(selene,helios).
refleja_sobre(selene,helios,gaia).
ilumina(helios,gaia).

ama(dios,X).
ama(madre(X),X).
```

Figura 2. Programa en Prolog que representa una base de datos cosmológica.

Las *constantes* (como por ejemplo "`helios`" y "`gaia`") nombran a individuos en el universo y comienzan con minúscula (en oposición a la lengua natural, en que los nombres propios se escriben con mayúscula). Las variables, como `X` en nuestro ejemplo, comienzan con mayúscula y en los datos se leen como universalmente cuantificadas. Así, el penúltimo axioma de la Figura 2 representa la información de que Dios ama a todo individuo `X`, sin excepción. Nótese que en los datos puede haber también argumentos que no son constantes, como `madre(X)` en el último axioma, el cual representa la

información de que todo individuo es amado por su madre. A continuación introduciremos cierta terminología de programación lógica y Prolog que se usará a lo largo del texto. Las definiciones formales de estos conceptos se pueden encontrar en [1], [6] o [9].

En programación lógica hay una única estructura de datos: los términos. Un **término** puede ser una *variable*, un *átomo*, un *número* o un *término compuesto*. Las **variables** se denotan con una secuencia de letras, dígitos o el símbolo "_" (en inglés *underscore*), pero deben comenzar obligatoriamente con una letra mayúscula o con el caracter "_". Por ejemplo, `X`, `R2d2`, `_`, `Numero`, `_numero` y `Un_numero_par` son variables, mientras que `x`, `un_par` y `2` no lo son. La variable "_" se denomina generalmente *variable anónima* y, como se verá más adelante, en Prolog tiene un tratamiento especial. Los **átomos** se denotan con una secuencia de símbolos entre comillas simples, o con una secuencia de letras, dígitos o el símbolo "_", pero deben comenzar obligatoriamente con una letra minúscula. Por ejemplo, `x`, `r2d2`, `numero`, `un_numero_par`, `'un numero par'`, `'c:/prolog/mi prop.pl'` y `' '` son átomos, mientras que `3d`, `_par`, y `c:/` no lo son. Los **números** incluyen los enteros y los reales con punto flotante, por ejemplo: `5`, `-5`, `-5.0` y `0.55`. Una **constante** es un átomo o un número. Un **término compuesto** es una expresión que tiene un nombre (llamado usualmente *functor*), que debe ser un átomo, seguido de un paréntesis "(", una secuencia de *términos* (llamados argumentos) separados por comas y un paréntesis que cierra, ")". Es importante destacar aquí la definición recursiva de término, ya que un término compuesto puede incluir un término compuesto como argumento. Por ejemplo, `'madre'(X)`, `r(5,x,_var)`, `p(a,p(a,X),p(p(p)))`, y `t(t(t(t(X))))` son términos compuestos válidos, mientras que `X(a)` y `5(a)` no lo son.

Un término compuesto se dice básico, o totalmente instanciado (en inglés *ground*), si no posee ninguna variable. Por ejemplo, `madre(ana,laura)` y `r(5,f(x),var)`, son términos básicos, mientras que `'madre'(X)` y `r(_,a)` no lo son. La **aridad** de un término (en inglés *arity*) es la cantidad de argumentos que tiene el término. Esto es, si un término cuyo functor es `t` tiene *n* argumentos entonces se dice que `t` tiene aridad *n* y usualmente se denota "`t/n`".

Un **dato** es un axioma constituido por un término y finalizado por un punto, tal como finalizan las oraciones en lengua natural. Un dato representa una relación (predicado) entre sus argumentos. Por ejemplo, `padre(abel,leo)` es un hecho que establece que Leo es el padre de Abel

o, en otras palabras, que existe la relación "*padre*" entre los individuos Abel y Leo. Si un predicado tiene como nombre (functor) al átomo `p` y es de aridad n, entonces también se denotará `p/n`. Es importante destacar que dos predicados con el mismo functor pero diferente aridad, representan diferentes predicados, como por ejemplo `a(1)` y `a(1,1)`.

El orden de los argumentos en un término o un predicado es significativo. Por ejemplo, en la Figura 1 el dato `padre/2` representa la relación "*padre*" y el primer argumento es el hijo y el segundo argumento es el padre. Esta convención es arbitrariamente elegida por el programador. Lo importante es ser consistentes: una vez decidido cual será el orden, debe respetarse el mismo orden en todo el programa y en todas las consultas a ese programa. Por ejemplo: la consulta o pregunta `?-padre(luis,juancito)` tendrá una respuesta negativa ya que, de acuerdo a la definición de la relación "*padre*" de la Figura 1, esta pregunta representa "*¿Es Juancito el padre de Luis?*".

Los datos en Prolog permiten representar estructuras de datos arbitrariamente complejas y en forma explícita. Esto permitirá, como se verá más adelante, mucha flexibilidad para almacenar y recuperar información. En la Figura 3 se muestran tres ejemplos. En el primero, el predicado `infoVuelo/5` tiene 5 argumentos: el número de vuelo, la ciudad de origen, la de destino, los datos de su salida y los de su llegada. Obsérvese que `salida/2` es un término compuesto que representa la fecha y hora de salida del vuelo, y `fecha/3` es, a su vez, un término compuesto. Es evidente que esta no es la única forma en la que esta información puede representarse. Por ejemplo, la ciudad de origen podría incluir aeropuerto, ciudad, país, etc. El tercer ejemplo es un predicado `mensaje/4` que representa un mensaje que el agente `ag1` reenvía al agente `ag2`, cuyo contenido es a su vez un mensaje que el agente `ag3` envió a `ag1`. El primer argumento de `mensaje/4` es el predicado `ac/1`, que representa el "acto comunicativo" e indica la intención que tiene el agente emisor al enviar el mensaje.

Como puede verse en los ejemplos mostrados hasta el momento, los datos en Prolog permiten representar información de manera muy flexible. Esto es sumamente atractivo en aplicaciones de inteligencia artificial, ya que provee una forma simple, flexible y poderosa de representar conocimiento por parte de un agente de software. Como se verá más adelante, este conocimiento podrá ser consultado; además, se podrán representar reglas de inferencias que transforman un programa Prolog en una base de datos deductiva.

```
infoVuelo(numero(123),origen('Buenos Aires'),destino('Vancouver'),
          salida(fecha(d(29),m(2),a(2000)),hora(h(20),m(12))),
          arribo(fecha(d(1),m(3),a(2000)),hora(h(12),m(34)))).

registro_empleado(nombre('Diego'),fecha_nac(a(1960),m(10),d(30)),
                  domicilio(calle(monumental),num(10)),).

mensaje(ac(reenviar),emisor(ag1),receptor(ag2),id(m123),
        contenido(mensaje(ac(consultar),emisor(ag3),receptor(ag1),
                  contenido('busco a ag2')) )).
```

Figura 3. Datos en Prolog.

Convención 1: No admitiremos sinónimos. Por ejemplo, si usamos el término unario "`madre(abel)`" para denominar a la madre de Abel, en el mismo programa no tendremos derecho a denominarla también `eva`. Esta restricción se debe a que los términos en programación lógica no se evalúan, sino que se unifican. El término `madre(abel)` no puede evaluarse, por ejemplo, a "`eva`", sino (como se verá más adelante) solamente *unificarse* con otro término, por ejemplo, `madre(X)` (mediante la asignación del valor "`abel`" a la variable `X`). La única manera de admitir el sinónimo "`eva`" para el individuo que hemos denominado "`madre(abel)`" sería mediante la incorporación de axiomas de reflexividad, simetría y transitividad, lo cual supone un enorme precio en términos de tiempo de ejecución.

Ejercicio 1 Decir si las siguientes expresiones son una constante, una variable, un término compuesto, o ninguna de estas cosas: `_` , `x`, `X`, `xVARIABLE`, 2, `_dos`, `dos_` , `'t(X)'`, `t(X)`, `es par`, `{a}`.

Ejercicio 2 Escribir predicados para almacenar la información de correos electrónicos que recibe una persona y que poseen la siguiente información: asunto, de, para, fecha, hora, servidor de correo, contenido.

2. Preguntas

Una pregunta, también llamada **consulta** (en inglés *query*), es una secuencia de predicados separados por comas, utilizada para interrogar o consultar si cierta información puede inferirse de un programa. Por ejemplo, "`luna(helios)`" o "`luna(selene)`" o "`mundo(gaia),sol(X),ilumina(X,ga-`

`ia)`" son tres consultas válidas, que representan, respectivamente, las siguientes preguntas: "*¿es Helios la luna?*", "*¿es Selene la luna?*" y "*¿es Gaia el mundo y existe un X que es el sol y X ilumina a Gaia?*". Si la pregunta `luna(selene)` se realizara al programa de la Figura 2, entonces esta pregunta tendría una respuesta afirmativa. La pregunta `luna(helios)`, en cambio, tendría una respuesta negativa.

Como ya se explicó antes, en Prolog la coma (",") representa una conjunción. Por lo tanto, si en una consulta la secuencia de predicados tiene más de un elemento, entonces será afirmativa cuando todos sus elementos individualmente dan una respuesta afirmativa; y negativa si al menos uno da negativa. Siguiendo con el ejemplo, si se consulta al programa de la Figura 2, la pregunta "`luna(helios),luna(selene)`" (*i.e.*, "*¿es Helios la luna y Selene la luna?*") dará una respuesta negativa, y "`sol(helios),luna(selene)`", afirmativa.

Una pregunta puede tener variables, como por ejemplo `luna(X)`, o estar totalmente instanciada, como `luna(selene)`. A continuación se mostrará que esto permite dos modos distintos de consultar un programa: en *modo verificación* (sin usar variables) o en *modo averiguación* (usando variables).

En general, los intérpretes de Prolog disponen de una "consola" (también llamada "*top level*") desde donde se puede cargar el código de un programa y luego realizar preguntas en forma *interactiva* (en el apéndice C se indican algunos intérpretes de Prolog de distribución gratuita). La Figura 4 muestra el inicio de un *top level* (genérico) en el cual, usando el comando `consult/1`, se ha "cargado" el archivo "cosmos.pl", que tiene el código del programa de la Figura 2. Una vez iniciado el intérprete, el *prompt* "`?-`" indica que el *top level* está a la espera de una consulta para el programa cargado. Obsérvese que se han realizado las consultas explicadas en los párrafos anteriores. Para salir de un intérprete de Prolog es habitual utilizar el comando `halt.` A continuación se explicará en detalle los dos modos de consultar a un programa.

2.1. Preguntas en modo verificación

Podemos usar preguntas cuyos argumentos son todos conocidos (es decir, no contienen ninguna variable) para verificar si una información concreta es cierta (es decir, puede inferirse a partir del programa) o no. Por lo tanto, en modo verificación, una **respuesta** a una pregunta será "`yes`", si puede inferirse a partir del programa, o "`no`", en caso contrario.

```
Welcome to this Prolog Interpreter
For help, use ?- help(Topic). or ?- apropos(Word).

?- consult('cosmos.pl').
% cosmos.pl compiled 0.00 sec, 80 bytes
Yes

?- luna(selene).
Yes

?- luna(helios).
No

?- luna(X).
X = selene
Yes

?- luna(X),sol(Y).
X = selene,
Y = helios
Yes

?- halt.
```

Figura 4. *Top level* de un intérprete Prolog donde se ha cargado "cosmos.pl".

Así, por ejemplo, podemos preguntar si a partir del programa de la Figura 2 puede afirmarse que "Helios brilla" escribiendo:

```
?- brilla(helios).
```

o consultar si es cierto (puede inferirse a partir del programa) que "Selene refleja a Helios sobre Gaia", escribiendo:

```
?- refleja_sobre(selene,helios,gaia).
```

Prolog responderá afirmativa o negativamente a cada pregunta, según pueda deducir o no lo preguntado a partir de los axiomas en el programa consultado. Por ejemplo,

```
?- brilla(helios).
yes
?- refleja_sobre(selene,helios,gaia).
```

```
yes
?- tierra(helios).
no
?- planeta(gaia).
no
?- ilumina(helios).
no
?- ilumina(helios,luna).
no
```

En el caso de las cuatro últimas consultas, la respuesta es "no", ya que estos predicados no pueden inferirse del programa.

También es posible consultar por varios predicados separándolos por comas, lo cual supone una conjunción de consultas individuales. Esto es, la respuesta de Prolog será "yes" si todas las respuestas individuales son "yes", o "no", en caso contrario. Por ejemplo:

```
?- brilla(helios), sol(helios).
yes
?- brilla(helios), tierra(helios).
no
?- planeta(gaia), ilumina(helios).
no
```

2.2. Preguntas en modo averiguación

Para averiguar el valor de algún parámetro dentro de una pregunta, simplemente notamos ese argumento desconocido con una variable cuyo nombre elegimos arbitrariamente o "mnemónicamente". Por ejemplo, a continuación mostramos una secuencia de preguntas con variables y sus correspondientes respuestas. Estas preguntas fueron formuladas a un intérprete como el mostrado en la Figura 4, en el cual se cargó previamente el programa mostrado en la Figura 2. Estas preguntas tratan de averiguar, respectivamente, "*quién refleja a Helios?*", "*qué astro refleja a helios sobre algún otro?*", "*quién refleja que sobre quién?*" y "*quién es un satélite?*".

```
?- refleja(X,helios).
X=selene
yes
```

```
?- refleja(Astro, helios).
Astro=selene
yes

?- refleja_sobre(Astro,helios,Otro).
Astro=selene
Otro=gaia
yes

?- refleja_sobre(X,Y,Z).
X=selene
Y=helios
Z=gaia
yes

?- satelite(X).
no
```

Obsérvese que las dos primeras preguntas de la secuencia anterior son en realidad la misma, en las que, simplemente, se utilizó un nombre diferente para la variable. Como puede verse, en el caso de las preguntas en modo averiguación, cuando la respuesta de Prolog es "`yes`", se muestra además qué individuo fue *asociado* a cada variable para probar que la respuesta es afirmativa.

Las variables en una consulta se asumen *existencialmente cuantificadas*, esto es, una consulta como "`q(X)`" debe leese: "¿existe algún individuo `X` tal que sea cierto `q(X)`?". De esta manera, así como una consulta sin variables, como `refleja(selene,helios)`, representa la pregunta "*¿refleja Selene a Helios?*", una consulta que tiene una variable, como por ejemplo `refleja(X,helios)`, representa la pregunta "*¿existe algún individuo X que refleja a Helios?*", y la consulta `refleja(X,Y)` representa la pregunta "*¿existe algún X que refleja a algún individuo Y?*".

En una consulta, toda aparición de una misma variable representará al mismo individuo; así, `refleja(X,X)` representa "*¿existe algún individuo X que refleja a X (esto es, a sí mismo)*". Observe que si se realiza la consulta `refleja(X,X)` al programa de la Figura 2, la respuesta será negativa, ya que no hay ningún individuo que se refleje a sí mismo. Sin embargo, la consulta `refleja(X,Y)` tendrá una respuesta afirmativa, ya que, en el programa de la Figura 2, hay un hecho `refleja(selene,helios)` que muestra que existe

un individuo `X` (`selene`) que refleja a `Y` (`helios`). A continuación se muestra la interacción con el intérprete de Prolog.

```
?-refleja(X,X).
no

?-refleja(X,Y).
X=selene
Y=helios
yes
```

Esto es, para que la consulta `refleja(X,X)` tenga una respuesta positiva, es obligatorio que el mismo individuo esté en los dos argumentos. Sin embargo, en `refleja(X,Y)`, como hay dos variables con distinto nombre, pueden ser tanto dos individuos diferentes como el mismo. Por ejemplo, considere que se ha cargado en un intérprete un programa que representa arcos de un grafo dirigido con los siguientes datos:

```
arco(n1,n1).
arco(n1,n2).
arco(n2,n3).
```

A continuación se muestra una secuencia de preguntas y sus correspondientes respuestas. Obsérvese que la respuestas a las dos primeras preguntas son ambas afirmativas y se resuelven con el mismo dato: `arco(n1,n1)`.

```
?-arco(X,X).
X=n1
yes

?-arco(X,Y).
X=n1
Y=n1
yes

?- arco(X,Otro),arco(Otro,n2).
X=n1
Otro=n1
yes

?- arco(X,Otro),arco(Otro,n3).
```

```
X=n1
Otro=n2
yes
```

En la consulta "`arco(X,Otro),arco(Otro,n2)`" se pregunta si "*existe un nodo `X`, que esté conectado al nodo `n2`, a través de `Otro`*". La respuesta a esta consulta se obtiene con los dos primeros datos del programa. En la última consulta se pregunta si "*existe un nodo `X`, que esté conectado al nodo `n3`, a través de `Otro`*", y en este caso, el nodo `Otro` es `n2`.

En particular, en los ejemplos anteriores, las respuestas afirmativas (`yes`) incluían una constante vinculada con cada variable de la consulta. Como se muestra en el siguiente ejemplo, es posible que un término mucho más complejo sea retornado como "instancia" de una variable. Considere un programa con los siguientes datos

```
infoVuelo(numero(123),origen('Buenos Aires'),
          destino('Vancouver'),
          salida(fecha(d(29),m(2),a(2000)),hora(h(20),m(12))),
          arribo(fecha(d(1),m(3),a(2000)),hora(h(12),m(34)))).

infoVuelo(numero(505),origen('Buenos Aires'),destino('Roma'),
          salida(fecha(d(29),m(2),a(2000)),hora(h(19),m(23))),
          arribo(fecha(d(1),m(3),a(2000)),hora(h(8),m(15)))).
```

A continuación se muestran las respuestas a diferentes consultas realizadas a un programa con los dos datos anteriores. La primera consulta pregunta si existe algún vuelo de algún origen a algún destino con alguna información de salida y arribo. La respuesta es afirmativa y cada una de las cinco variables de esta consulta es "instanciada" con un término compuesto (ver Definición 3 a continuación).

```
?- infoVuelo(Num,Origen,Dest,Salida,Arribo).
Num = numero(123)
Origen = origen('Buenos Aires')
Dest = destino('Vancouver')
Salida = salida(fecha(d(29), m(2), a(2000)), hora(h(20),
         m(12)))
Arribo = arribo(fecha(d(1), m(3), a(2000)), hora(h(12),
         m(34)))
yes
```

```
?- infoVuelo(Num,Origen,destino('Madrid'),Salida,Arribo).
no

?- infoVuelo(N,O,D,salida(fecha(d(Dia),m(2),a(2000)),Hora),
   Arribo).
N = numero(123),
O = origen('Buenos Aires'),
D = destino('Vancouver'),
Dia = 29,
Hora = hora(h(20), m(12)),
Arribo = arribo(fecha(d(1), m(3), a(2000)), hora(h(12),
         m(34))).
```

La segunda consulta pregunta si existe algún vuelo con destino a Madrid y la respuesta es negativa. Para ello, en el tercer argumento, en lugar de usar una variable se usa el término `destino('Madrid')`. La tercera consulta pregunta si existe algún vuelo cuya fecha de salida sea en febrero de 2000. Obsérvese que en el cuarto argumento no se utiliza una variable, sino que se utiliza el término `salida(fecha(d(Día),m(2),a(2000)),Hora)`, donde parte está instanciado y parte es indicado con variables. Como la respuesta es afirmativa, también incluye los valores para todas las variables utilizadas en la consulta, sin importar el nivel de anidamiento en la estructura.

Para resolver una consulta, Prolog debe vincular cada variable con algún término. En programación lógica, este concepto se denomina **unificación** y se realiza buscando una **sustitución** para cada una de las variables. Ambos conceptos son definidos a continuación.

Definición 1 (sustitución de variables) *Una sustitución de variables es un conjunto finito (posiblemente vacío) de pares de la forma $X_i = t_i$, donde X_i es una variable y t_i es un término; además $X_i \neq X_j$ para todo $i \neq j$, y X_i no aparece en ningún t_k, para todo i y k.*

Esto es, una sustitución de variables por valores es la asignación, a cada una de las variables, de un término (es decir, una constante, una variable o un término compuesto). Por ejemplo, podemos indicar la sustitución que asigna a X el valor f(a,b) con la notación {X=f(a,b)}; y la sustitución que asigna a X el valor f(a,Y) y además a Y el valor 1 con la notación {X=f(a,Y), Y=1}. El resultado de aplicar una sustitución a un término es otro término, en el cual cada aparición de cada variable que interviene en la sustitución

ha sido sustituida por el valor correspondiente. Por ejemplo, la aplicación de {X=f(a,Y), Y=1} sobre el término p(X,Y) es el término p(f(a,1),1).

Definición 2 (términos unificables) *Dos términos son unificables si existe una sustitución que, aplicada a cada uno de ellos, resulta en un único término, que llamamos término resolvente.*

Por ejemplo, la aplicación de {X=f(a,Y), Y=1} sobre los términos p(X,1) y p(f(a,1),Y) resulta en un único término p(f(a,1),1), llamado resolvente de p(X,1) y p(f(a,1),Y). Si dos términos son unificables a través de la sustitución θ, entonces θ es llamada **unificador** de T_1 y T_2.

Definición 3 (instancia) *El término T_1 es una instancia de otro término T si T_1 se obtiene de aplicar una sustitución de variables θ sobre el término T (usualmente denotado $T\theta$).*

Podemos ahora definir más formalmente cómo se obtiene una respuesta a una pregunta que se realiza sobre un programa que solo tiene datos. Una consulta C tiene una respuesta afirmativa si existe un hecho H en el programa y una sustitución de variables θ, tal que H y C son términos unificables usando θ. Cuando una consulta con variables C se realiza sobre un programa formado solamente por datos totalmente instanciados, la respuesta es afirmativa (*i.e.*, `yes`) si existe una sustitución de variables θ que muestra que en el programa existe un hecho H que es una instancia de la consulta C (*i.e.*, $H = C\theta$). Si no existe tal instancia, entonces la respuesta es `no`. Como se vio en los ejemplos anteriores, cuando la respuesta es `yes`, el *top level* del intérprete de Prolog también muestra en pantalla la sustitución de variables que, aplicada a la consulta y a un hecho del programa, hace que estos sean unificables. Más adelante, se explicará qué ocurre cuando más de un hecho del programa es unificable con la consulta realizada, y cómo se obtiene una respuesta si el programa, además de datos, tiene reglas.

Ejercicio 3 Aplicar las siguientes subtituciones a los siguientes términos. La sustitución `{X=a,Y=f(a)}` a `p(f(X),Y)` y `{X=que,Y='hola'}` a `sms(Y,X,tal)`.

Ejercicio 4 Si es posible unificar los siguientes términos, mostrar la sustitución que los unifica. Si no es posible, decir por qué. (a) `p(X,f(Y),h(Y,X))` y `p(Z,Q,h(a,b))`; (b) `p(a,b)` y `p(X,Y,Z)`; (c) `p(a,b)` y `t(X,Y)`; y (d) `p(a,b)` y `p(b,X)`.

Es importante destacar que los datos de un programa también pueden tener variables. En el programa de la Figura 2 pueden verse dos hechos con variables:

```
ama(dios,X).
ama(madre(X),X).
```

A diferencia de las consultas, donde las variables se asumen *existencialmente cuantificadas*, en los datos, las variables se asumen *universalmente cuantificadas*. Esto es, un dato como `ama(dios,X)` representa que "*para todo individuo X, Dios ama a X*", y el dato `ama(madre(X),X)` tiene el significado "*cualquier individuo X es amado por madre(X)*".

Como se explicó antes, una consulta C tiene una respuesta afirmativa si existe un hecho H en el programa y sustitución de variables (unificador) θ, tal que H y C son términos unificables usando θ. Por ejemplo, la consulta `ama(A,padre(tío(pepe)))` tiene una respuesta afirmativa y el unificador es {`A=dios, X=padre(tío(pepe))`}. El lector comprenderá que cualquier consulta de la forma `ama(dios,`T_i`)`, donde T_i es un término válido, dará una respuesta afirmativa. Por ejemplo, `ama(dios,ana)`, `ama(dios,Y)`, `ama(dios,raíz(2))` y `ama(dios,dios))`, todas dan una respuesta afirmativa, gracias a las sustituciones de variables {`X=ana`}, {`X=Y`}, {`X=raíz(2)`} y {`X=dios`}, respectivamente.

A continuación se muestra una secuencia de preguntas que son realizadas a un intérprete de Prolog, el cual tiene cargado el programa de la Figura 2. Obsérvese que si la consulta es totalmente instanciada (*i.e.*, sin variables), entonces el *top level* de Prolog no muestra ninguna sustitución de variables. Esto es, la sustitución de variables que muestra un *top level* sólo contiene aquellas variables que están presentes en la consulta (y no las que puedan estar en los datos).

```
?- ama(dios,ana).
Yes

?- ama(dios,madre(ana)).
Yes

?- ama(X,pez).
X = dios
Yes
```

```
?- ama(madre(ana),ana).
Yes

?- ama(madre(X),ana).
X = ana
Yes

?- ama(madre(X),padre(tío(ana))).
X = padre(tío(ana))
Yes

?- ama(madre(X),madre(Z)).
X = madre(Z)
Yes

?- ama(madre(Q),madre(Z)),  Z = pedro.
Q = madre(pedro),
Z = pedro
Yes
```

En la última consulta del ejemplo anterior, se utilizó el símbolo "=". Este símbolo tiene un significado propio en Prolog y debe leerse como: "*unifica con*". Esto es, `Z=pedro` significa "la variable `Z` debe unificarse con el término `pedro`". A continuación mostramos una secuencia de consultas realizadas a un intérprete Prolog donde se utiliza este operador. En las tres primeras consultas puede verse que "=" no tiene el significado usual de las matemáticas, ni tampoco es un operador de asignación, como en otros lenguajes de programación. Obsérvese que en la primera consulta, la variable `X` es ligada a toda la expresión (término) `2+3`. En la segunda puede verse, además, la asociatividad del operador "+".

```
?- X = 2 + 3.
X = 2+3
Yes

?- X + Y = 2 + 3 + 4 + 5.
X = 2+3+4,
Y = 5
Yes
```

```
?- X+Y =  2*3 + 4*5.
X = 2*3,
Y = 4*5
Yes

?- p(X,Y) = p(f(a),t(t(U))).
X = f(a),
Y = t(t(U))
Yes

?- X=Y, Y=Z, X=a.
X = a,
Y = a,
Z = a
Yes

?- X=Y, Y=b.
X = b,
Y = b,
Yes
```

Las últimas dos consultas de la secuencia anterior muestran algo muy importante de Prolog: al ejecutar la consulta `X=Y`, la variable `X` queda ligada a `Y` y viceversa. Esto es, si luego un término `t` es ligado a cualquiera de estas dos variables, la otra también quedará ligada a `t`.

2.3. Obteniendo todas las respuestas de una consulta: backtracking

En los ejemplos anteriores, el lector puede haber notado que, cuando se realiza una consulta a un programa, puede ocurrir que más de un dato en el programa sea unificable con dicha consulta. Esto es, pueden existir varios datos que sean instancias de la consulta realizada. Cuando esto ocurre, Prolog retornará la primera instancia encontrada. Para ello utiliza el orden (de arriba abajo) en que los hechos se escribieron en el programa. Como se mostrará en el ejemplo siguiente, utilizando el símbolo "`;`" el *top level* de un intérprete de Prolog permite recuperar todas las instancias que unifican con una consulta, y muestra en pantalla todos los unificadores (sustituciones) correspondientes.

Considere un programa Prolog con los datos que se muestran en la Figura 5, en el cual se representa la información de todos los cursos dictados en el aula 38. Como indica el comentario de la primera línea[1], cada hecho (dato) indica el código del curso, el profesor asignado, el aula en que se dicta, el día y el rango horario. Por ejemplo, el curso `cs001` es dictado por Alan Turing, los lunes y los jueves, de 8 a 10 horas, en el aula 38.

```
% curso( código, profesor, aula, día, horario)
curso(cs001, 'Alan Turing', 38, lunes, h(8,10)).
curso(cs001, 'Alan Turing', 38, jueves, h(8,10)).
curso(cs002, 'John von Neumann', 38, lunes, h(14,16)).
curso(cs002, 'John von Neumann', 38, viernes, h(14,16)).
curso(cs003, 'George Boole', 38, martes, h(14,16)).
curso(cs003, 'George Boole', 38, jueves, h(14,16)).
curso(cs004, 'Alan Turing', 38, martes, h(8,10)).
curso(cs004, 'Alan Turing', 38, viernes, h(8,10)).
```

Figura 5. Datos con la información de cursos, profesores y horarios.

Es claro que la consulta `curso(cs0001,Prof,Aula,Día,Hora)` tendrá una respuesta positiva a partir del programa de la Figura 5, ya que unifica con el primer hecho de este programa. Sin embargo, no es el único hecho que unifica. Lo mismo ocurre con la consulta `curso(X,P,A,D,H)`, que unifica con todos los hechos del programa de la Figura 5. Como se indicó antes, el símbolo "`;`" puede utilizarse en el *top level* para ir obteniendo otras respuestas. Por ejemplo, a continuación se muestra una secuencia de consultas realizadas a un intérprete que tiene cargado el programa de la Figura 5. La primera consulta pregunta: "*¿qué cursos hay los días jueves?*". Esta consulta da una respuesta positiva y la sustitución de variables se muestra en pantalla. Luego, el símbolo "`;`" es usado, y se obtiene una segunda sustitución que también responde a esta consulta. Obsérvese que el símbolo "`;`" se usa para interrogar si hay una tercera instancia, pero la respuesta es `no`.

```
?- curso(X,P,A,jueves,H).

X = cs001,
```

[1] En Prolog, el símbolo "`%`" se utiliza para denotar un comentario en el código del programa e indica que el resto de la línea debe ser ignorada por el intérprete.

```
P = 'Alan Turing',
A = 38,
H = h(8, 10) ;

X = cs003,
P = 'George Boole',
A = 38,
H = h(14, 16) ;

No
```

Como se muestra a continuación, la consulta `curso(_,_,38,D,H)` (“*¿en qué horario está ocupada el aula 38?*”) unifica con todos los datos del programa de la Figura 5. Como se apuntó antes, la variable anónima “`_`” tiene un tratamiento especial en Prolog: unifica con cualquier término, pero no produce una ligadura que pueda utilizarse luego en otra parte de la consulta o cláusula. Por lo tanto, puede utilizarse cuando no se quieren mostrar algunos de los valores de una respuesta.

```
?- curso(_,_,38,D,H).

D = lunes,
H = h(8, 10) ;

D = jueves,
H = h(8, 10) ;

D = lunes,
H = h(14, 16) ;

D = viernes,
H = h(14, 16) ;

D = martes,
H = h(14, 16) ;

D = jueves,
H = h(14, 16) ;
```

```
D = martes,
H = h(8, 10) ;

D = viernes,
H = h(8, 10) ;

No
```

El método por el cual Prolog busca todas las respuestas posibles se llama *backtracking. A continuación mostraremos en detalle el tercer elemento de un programa: las reglas.*

3. Reglas

Los datos son aserciones incondicionales. Las reglas, en cambio, permiten establecer aserciones condicionales. Por ejemplo, para expresar que "si un mundo está contaminado entonces sufre", se puede usar la regla:

```
sufre(X):- mundo(X), contaminado(X).
```

De esta manera, para expresar las condiciones bajo las cuales un predicado es cierto, escribimos ese predicado seguido de ":-", y a continuación la lista de aquellos predicados que se necesitaría satisfacer para que fuera cierto. Eso constituye una regla. El predicado que se encuentra a la izquierda del símbolo ":-" se llama ***conclusión*** *de la regla, y los que se encuentran a la derecha,* ***premisas****. En el ejemplo anterior, si hay algún individuo* `t` *para el cual se cumplen las premisas* `mundo(t)` `y contaminado(t)`*, esta regla permite concluir que* `sufre(t)`*.*

Las reglas son sentencias de la forma `H :-` $B_1, B_2, \ldots B_n$ *(con* $n \geq 0$*), donde la conclusión* `H` *también es llamada cabeza de la regla y las premisas* $B_1, B_2, \ldots B_n$*, cuerpo de la regla. En el caso particular de que no haya premisas (*$n = 0$*), entonces tenemos un dato, o aserción incondicional. Es decir, los datos pueden pensarse como un caso especial de las reglas que no tienen premisas, y en los cuales se puede prescindir del símbolo ":-". Por otro lado, las consultas pueden tomarse como un caso particular de regla sin cabeza. Las reglas, los datos y las consultas son también llamados* ***cláusulas de Horn*** *(en inglés Horn clauses) o, simplemente, cláusulas.*

Por un lado, las reglas pueden pensarse como una forma de definir nuevas relaciones (predicados) usando otras relaciones más simples. Por ejemplo, en la regla `abuelo(X,Y):-padre(X,Z),padre(Z,Y)`*, el predicado* `abuelo` *está definido*

usando el predicado `padre`*. Al igual que en las consultas conjuntivas, si una variable con nombre* `X` *aparece más de una vez en una regla, entonces se refiere siempre al mismo individuo. Por lo tanto, la lectura declarativa de la regla anterior es: "Para todo X,Y, y Z, Y es el abuelo de X, si Z es el padre de X, e Y es el padre de Z".*

Por otro lado, las reglas también se pueden pensar como una forma de establecer consultas más complejas en términos de consultas más simples. Por ejemplo, dada la regla `sufre(X):- mundo(X), contaminado(X)`*, la pregunta* `sufre(gaia)` *se traducirá en poder responder las preguntas* `mundo(gaia)` *y* `contaminado(gaia)`*. Por supuesto, cada vez que usamos un predicado como condición de otro, debemos incluir en nuestro programa definiciones que permitan comprobar o desacreditar esa condición.*

Por ejemplo, si agregamos el dato "`contaminado(gaia)`*" y la regla anterior al programa de la Figura 2, podemos preguntar ahora "¿qué mundo sufre?" de la siguiente manera.*

```
?- mundo(X), sufre(X).
```

a lo cual Prolog responderá:

```
X=gaia
```

El proceso por el cual se responde a una pregunta Prolog está automatizado y consiste en:

1. *Encontrar el primer dato o la primera regla cuya conclusión sea unificable con el primer predicado en la pregunta.*
2. *Encontrar la sustitución de variables θ menos comprometida que logra tal unificación (por "menos comprometida" se entiende la que unifique las dos expresiones con el menor número posible de asignaciones).*
3. *Reemplazar el primer predicado de la pregunta por las premisas de la regla usada, a las cuales se aplicó la sustitución θ.*
4. *Aplicar la sustitución θ al resto de la pregunta que no fue aún considerado.*

Así, por ejemplo, ante la pregunta "¿Qué mundo sufre?", esto es:

```
?- mundo(X), sufre(X).
```

Prolog encuentra la "regla" incondicional (o dato) `mundo(gaia)`*, que se unifica con el primer predicado de la pregunta mediante la sustitución* `{X=gaia}`*, y reemplaza ese primer predicado de la pregunta por la lista de premisas de la regla usada (que en este caso es vacía, ya que se usó un dato), al tiempo que aplica la sustitución usada sobre el resto de la pregunta. Obtenemos de todo ello la nueva pregunta:*

```
?- sufre(gaia).
```

Aplicando nuevamente los pasos (1) a (4), reemplazamos "`sufre(gaia)`*" por la lista de predicados que corresponde a las premisas de la regla: "*`mundo(gaia), contaminado(gaia)`*, obteniendo así la nueva pregunta:*

```
?- mundo(gaia), contaminado(gaia).
```

Como Prolog no re-usa resultados intermedios, demostramos "de nuevo" el primer predicado, usando los pasos (1) a (4) con el dato `mundo(gaia)` *y obtenemos así la nueva pregunta*

```
?- contaminado(gaia).
```

que desaparece aplicando los pasos (1) a (4) usando el dato `contaminado (gaia)` *que figura en nuestro programa. El proceso termina cuando se llega a la consulta vacía:*

```
?-
```

Prolog, entonces, usa las sustituciones que se relacionan con variables en la pregunta, componiendo sustituciones si hiciera falta, para imprimir la respuesta en pantalla, en este caso:

```
X=gaia.
```

Para que la conclusión de una regla sea cierta, todas las premisas deben ser ciertas. Esto es, si la consulta `H` *se intenta resolver con la regla* `H :-` $B_1, B_2, \ldots B_n$, *entonces ahora hay que resolver la consulta* $B_1, B_2, \ldots B_n$ *que representa una conjunción de consultas.*

Dada una regla `H :-` $B_1, B_2, \ldots B_n$, *su cuerpo,* $B_1, B_2, \ldots B_n$, *representa una conjunción de consultas. Esto es, para que la conclusión de una regla sea cierta, todas las premisas deben ser ciertas. No obstante, puede ocurrir que, para un predicado* `H`, *existan varias reglas con cabeza* `H`. *Esto representa diferentes opciones para probar* `H`, *es decir es una disyunción de reglas. Por lo tanto, se podrá concluir* `H`, *si al menos para una de esas reglas sus premisas son ciertas. Por ejemplo, a continuación se define el predicado* `nieto(A,N)` *con cuatro reglas.*

```
nieto(A,N):- madre(N,M),madre(M,A).
nieto(A,N):- madre(N,M),padre(M,A).
nieto(A,N):- padre(N,P),madre(P,A).
nieto(A,N):- padre(N,P),padre(P,A).
```

Esto es, "el nieto de A es N ***si*** *M es la madre de N* ***y*** *A es la madre de M"* ***o*** *"M es la madre de N y A es el padre de M"* ***o*** *"P es el padre de N y A es la madre de M"* ***o*** *"P es el padre de N y A es el padre de M".*

Considere que se tienen los datos `padre(leo,luis)` *y* `madre(luis,ana)`*, entonces para responder la pregunta* `?-nieto(ana,leo)` *se utilizará la tercera regla.*

Como se verá más adelante en los ejemplos, durante la búsqueda de una respuesta, si hay más de una regla cuya cabeza unifica con una consulta, entonces, a pesar de que cualquiera de las reglas podría usarse para resolver la consulta, Prolog intentará primero con la que está más arriba en el código del programa. Las otras opciones se convierten en puntos a los cuales se puede retornar en busca de otra opción (en inglés choice points). Si en cualquier momento se produce un fracaso antes de haber podido responder a una consulta dada, el espacio de búsqueda se recorre ordenadamente, deshaciendo sustituciones previas hasta encontrar una alternativa que permita dar una respuesta positiva, si es que existe. Dicho proceso se denomina backtrack (volver sobre su propio camino), y es el mismo mecanismo que cómo Prolog encuentra todas las respuestas posibles a una consulta.

Por ejemplo, considere el programa de la Figura 6, donde la primera regla establece que "*T es el tío de S, si T es el hermano de P, y P el progenitor de S*". La segunda y la tercera regla definen al predicado progenitor. Esto es, definen dos opciones para probar que P es el progenitor de Hijo: "*P es el progenitor de Hijo, si P es la madre de Hijo*" o como segunda opción "*P es el progenitor de Hijo, si P es el padre de Hijo*". Cualquiera de las dos opciones es válida, pero, como fue dicho antes, Prolog probará en el orden establecido.

```
tío(S,T):- hermano(T,P),progenitor(S,P).
progenitor(Hijo,P):- madre(Hijo,P).
progenitor(Hijo,P):- padre(Hijo,P).
hermano(c,a).
hermano(c,d).
hermano(c,e).
madre(j,a).
madre(h,a).
padre(i,e).
```

Figura 6. Programa que define el predicado `tío/2`.

El lector puede comprobar fácilmente que la consulta `tío(i,c)` tiene una respuesta positiva, ya que `c` es el hermano de `e` y `e` es el progenitor (padre) de `i`. Sin embargo, Prolog, siguiendo el algoritmo anterior y de acuerdo al orden de las reglas en el programa, tendrá que ver primero otras opciones. Inicialmente, usando la primera regla, la consulta `tío(i,c)` se transforma

en `hermano(c,P),progenitor(i,P)` (*i.e.*, buscar un hermano de `c` que sea progenitor de `i`). Obsérvese que hay tres opciones para buscar un hermano de `c`.

El primero que aparece es `hermano(c,a)`, con lo cual ocurren dos cosas: (1) se deja marcado que hay otras opciones para seguir buscando la respuesta, y (2) la consulta `hermano(c,P)` se contesta provisoriamente con `hermano(c,a)` y la sustitución de variables `{P=a}` se aplica al resto de la consulta pendiente. Con esto, `progenitor(i,P)` se convierte en `progenitor(i,a)`. Aunque hay dos formas para buscar un progenitor (padre o madre), ningún caso tiene éxito, ya que `a` no es el padre, ni la madre de `i`. Por lo tanto, la consulta `progenitor(i,a)` **falla**.

Esto no quiere decir que la consulta original tenga una respuesta negativa, porque, como ya se explicó antes, aún quedan opciones por explorar: hay otras dos opciones más de hermanos de `c`. En este caso, Prolog regresa atrás (*backtrack) y deshace las substituciones de variables correspondientes (la explicación de cómo se realiza este mecanismo es sencilla, pero está fuera del alcance de este texto). Con la segunda opción (`hermano(c,d)`), ocurre lo mismo que antes, ya que `d` no es progenitor de `i`. Prolog vuelve otra vez hacia atrás (backtrack) a otro punto de elección y ahora, con la tercera opción (`hermano(c,e)`), se puede probar que `progenitor(i,e)` es cierto, ya que `padre(i,e)` es un dato del programa. De esta manera, la respuesta a `tío(i,c)` es afirmativa.*

La Figura 7 muestra la traza completa que se obtiene de un intérprete Prolog usando la opción `trace`, al ejecutar la consulta `tío(i,c)` al programa anterior. El comando predefinido `trace` hace que el top level muestre en pantalla toda consulta que es ejecutada (`Call`), cuando regresa con éxito de una consulta (`Exit`), cuando la consulta falla (`Fail`) o cuando vuelve a intentar con otra opción (`Redo:`) mediante el backtracking. Nótese que (`Redo:`) es indicado en aquellos predicados que tienen más de una opción en el programa, esto es: `progenitor/2` y `hermano/2`. Obsérvese, además, que el identificador "`_L170`" corresponde a una variable, y que esta variable es instanciada con cada uno de los hermanos de `c`, cada vez que la consulta retorna con éxito de un `Call` o `Redo`.

Este mismo mecanismo de *backtracking permite obtener todas las respuestas posibles para una consulta usando el símbolo ";". Por ejemplo, a continuación se muestra que la consulta `tío(Sobrino,Tío)` tiene tres respuestas posibles.*

```
?- tío(Sobrino,Tío).
Sobrino = j,
Tío = c ;
```

```
4 ?- trace.
Yes

[trace] 4 ?- tío(i,c).
   Call: (7) tío(i, c) ?
   Call: (8) hermano(c, _L170) ?
   Exit: (8) hermano(c, a) ?
   Call: (8) progenitor(i, a) ?
   Call: (9) madre(i, a) ?
   Fail: (9) madre(i, a) ?
   Redo: (8) progenitor(i, a) ?
   Call: (9) padre(i, a) ?
   Fail: (9) padre(i, a) ?
   Redo: (8) hermano(c, _L170) ?
   Exit: (8) hermano(c, d) ?
   Call: (8) progenitor(i, d) ?
   Call: (9) madre(i, d) ?
   Fail: (9) madre(i, d) ?
   Redo: (8) progenitor(i, d) ?
   Call: (9) padre(i, d) ?
   Fail: (9) padre(i, d) ?
   Redo: (8) hermano(c, _L170) ?
   Exit: (8) hermano(c, e) ?
   Call: (8) progenitor(i, e) ?
   Call: (9) madre(i, e) ?
   Fail: (9) madre(i, e) ?
   Redo: (8) progenitor(i, e) ?
   Call: (9) padre(i, e) ?
   Exit: (9) padre(i, e) ?
   Exit: (8) progenitor(i, e) ?
   Exit: (7) tío(i, c) ?
Yes
```

Figura 7. Traza que muestra el *backtraking realizado en la consulta* `tío(i,c)`

```
Sobrino = h,
Tío = c ;

Sobrino = i,
Tío = c ;
```

```
No
```

Ejercicio 5 Escribir reglas para definir los predicados `primo/2` y `prima/2`.

4. Negación por falla

Prolog posee un operador para proveer una forma de negación, llamada ***negación por falla*** *(en inglés negation as failure). Este operador se denota con el símbolo* `\+` *y se antepone a una consulta (aunque en algunos intérpretes también se suele denotar con* `not`*). Por ejemplo,* `\+ tío(i,c)` *es una consulta que utiliza la negación por falla. Su nombre proviene del comportamiento que tiene este operador. Esto es, si una consulta* `Q` *tiene éxito, entonces* `\+ Q` *falla, y si* `Q` *falla, entonces* `\+ Q` *tiene éxito. Por ejemplo:*

```
?- tío(i,c).
Yes

?- \+ tío(i,c).
No

?- tío(leo,marcos).
No

?- \+ tío(leo,marcos).
Yes
```

El operador `\+` *puede usarse también en el cuerpo de las reglas de un programa. Por ejemplo, "un profesor puede evaluar a un alumno si no es su padre, ni su madre" puede escribirse con la siguiente regla:*

```
puede_evaluar(Profesor,Alumno):- \+ padre(Alumno,Profesor),
                                 \+ madre(Alumno, Profesor).
```

Más detalles sobre la negación en Prolog pueden encontrarse en [2, 4, 6, 9].

5. Reglas recursivas

Considere el siguiente programa en Prolog, que define de manera recursiva los números naturales. La primera regla (en realidad un hecho) establece que "el `0` *es*

un número natural", y la segunda regla (que se define en términos de sí misma) establece que "si X es un número natural, entonces, el sucesor de X, que denotamos por simplicidad s(X), también es un número natural".

```
natural(0).
natural(s(X)):-natural(X).
```

De esta manera, los números naturales serán `0`, `s(0)`, `s(s(0))`, `s(s(s(0)))`, *y así siguiendo; con la convención de que* $s^n(0)$ *denota el número natural n. Si realizamos consultas totalmente instanciadas (i.e., preguntas en modo verificación), podremos determinar si un elemento es o no un número natural de acuerdo a la notación anterior. Por ejemplo:*

```
?- natural(0).
Yes

?- natural(s(s(0))).
Yes

?- natural(1).
No
```

Sin embargo, también es posible realizar una pregunta en modo averiguación, como, por ejemplo, `natural(N)`*, la cual retornará el primer natural. Luego, usando el símbolo ";" se podrán obtener uno a uno, "todos" los números naturales: siempre que pongamos ";" Prolog devolverá una nueva respuesta que es el sucesor de la anterior.*

```
?- natural(N).
N = 0 ;

N = s(0) ;

N = s(s(0)) ;

N = s(s(s(0))) ;

N = s(s(s(s(0)))) ;

N = s(s(s(s(s(0))))) ;
```

```
N = s(s(s(s(s(s(0))))))  ;

N = s(s(s(s(s(s(s(0)))))))  ;

N = s(s(s(s(s(s(s(s(0))))))))  ;

N = s(s(s(s(s(s(s(s(s(0)))))))))  ;

Yes
```

Ejercicio *6* Con la misma notación para los números naturales establecida antes, donde $\texttt{s}^n\texttt{(0)}$ denota el número natural n, definir un predicado `par/1` que determine si un número es par o no. Por ejemplo, las preguntas `par(0)` y `par(s(s(0)))` deben responder `yes` y la pregunta `par(s(0))` debe responder `no`. ¿Qué ocurre con la pregunta `par(X)`? ¿Pueden definirse los impares usando los pares?

Considere el programa de la Figura 8. El predicado `arco/2` representa los arcos de un grafo dirigido, y las reglas con cabeza `nodo/1` permiten determinar si un nodo pertenece al grafo. En el caso de que se quieran tener nodos aislados (i.e., que no pertenezcan a ningún arco), entonces pueden agregarse al programa datos de la forma `nodo(n)`. Las dos reglas con cabeza `conectado/2` definen cuándo un nodo `N1` está conectado a través de algún camino con otro nodo `N2`. Obsérvese que la segunda de estas dos reglas es recursiva. Finalmente, la última regla define una relación 0-aria que devuelve una respuesta positiva si hay un ciclo en el grafo. Esto es, si existe un nodo `X` tal que `X` esté conectado con él mismo. Como `hay_ciclo/0` no tiene variables se usa el predicado predefinido `write/1` para mostrar en pantalla cual es el nodo en el que se encontró un ciclo.

Ejercicio *7* Usando reglas recursivas, definir un predicado `mostar_camino/2`, que reciba dos nodos n_1 y n_2 de un grafo, y muestre en pantalla la sucesión de nodos que hay que recorrer para llegar de n_1 a n_2.

6. Listas

Las listas son una estructura de datos muy importante en Prolog, tanto que existe una notación especial y operaciones predefinidas para manejo de listas. Referimos al lector a [9], donde se introducen diferentes notaciones para una lista. En este texto utilizaremos únicamente la forma más usual de escribir una lista en Prolog:

```
arco(n1,n6).
arco(n1,n7).
arco(n1,n2).
arco(n2,n3).
arco(n3,n4).
arco(n4,n5).
arco(n3,n1).
arco(n5,n4).
nodo(X):- arco(X,_).
nodo(X):- arco(_,X).
conectado(N1,N2):-arco(N1,N2).
conectado(N1,N2):-arco(N1,X),conectado(X,N2).
hay_ciclo:-nodo(X),conectado(X,X),write(esta_conectado(X,X)).
```

Figura 8. Grafos en Prolog

como una sucesión (posiblemente vacía) de términos separados por una coma, que comienza con el símbolo "[" y termina con "]". De esta manera, `[]` es la lista vacía (tiene cero elementos), `[hola]` una lista de un elemento y `[a, f(t), [1,2], X, f(X,X)]` una lista de cinco elementos, donde el tercero es, a su vez, una lista de dos elementos. Obsérvese que `[[]]` es una lista de un elemento que tiene como elemento la lista vacía. Como es de esperar, una lista no es un conjunto, y el orden de los elementos importa.

*El operador "|" puede usarse para "separar" en dos partes una lista. En una lista `[H|T]`, la primera parte `H` unifica con el primer elemento y se llama cabeza de la lista (*head); y la segunda `T`, llamada cola (*tail), unifica con el resto de la lista. A continuación se muestran, a modo de ejemplo, algunas consultas que utilizan el operador `=`, a fin de mostrar cómo se unifican los elementos de una lista en la presencia del operador "|". Las dos primeras consultas muestran claramente que no es lo mismo `[X,Y]` que `[X|Y]`. La lista `[X,Y]` tiene exactamente dos elementos, mientras que `[X|Y]` representa una lista de uno o más elementos.*

```
?- [X|Y] = [1,2,3,4].
X = 1,
Y = [2, 3, 4]
Yes

?- [X,Y] = [1,2,3,4].
No
```

```
?- [X|Y] = [1].
X = 1,
Y = []
Yes

?- [Pri,Seg,Ter|Resto] = [a, b, c, d, e, f].
Pri = a,
Seg = b,
Ter = c,
Resto = [d, e, f]
Yes

?- [X|Y] = [ [1 ,2, 3] , a , [ ] , ultimo].
X = [1, 2, 3],
Y = [a, [], ultimo]
Yes

?- [X|Y] = [[]].
X = [],
Y = []
Yes

?- [X|Y] = [].
No
```

Las listas en Prolog son una estructura de datos muy flexible y poderosa para representar, recuperar y procesar información. Está claro que las listas son una estructura que se puede definir recursivamente: la lista vacía es una lista, y una secuencia con un elemento y seguida de una lista es una lista. En Prolog:

```
lista([]).
lista([X|Xs]):-lista(Xs).
```

Dos de las operaciones más elementales sobre listas son determinar si un elemento pertenece a una lista y concatenar dos listas. Los siguientes predicados definen la relación `member/2` *para determinar si un elemento pertenece a una lista; y* `append/3`*, que concatena dos listas en una tercera. A continuación incluimos el código en Prolog de dichas operaciones. Estas operaciones son tan importantes que, habitualmente, ya están predefinidas en los intérpretes de Prolog.*

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

append([],L,L).
append([X|Xs],L,[X|Zs]):-append(Xs,L,Zs).
```

Obsérvese la definición recursiva de `member/2`: un elemento pertenece a una lista si es su primer elemento o pertenece al resto de la lista. La definición de `append/3` también es recursiva: concatenar la lista vacía con una lista cualquiera `L` da como resultado la lista `L`; ahora, concatenar una lista `[X|Xs]` que no es vacía con `L` es concatenar `Xs` con `L` y luego agregar `X` al principio.

A continuación se muestran unas consultas a los predicados `member/2` y `append/3`. Obsérvese que, al realizar la consulta `member(X,[1,2,3,4])` ("¿Existe algún elemento de la lista [1,2,3,4]?"), es posible ir obteniendo uno por uno todos los elementos de la lista. Al realizar la pregunta `append(X,Y,[1,2,3,4])` ("¿qué listas al concatenarse forman [1,2,3,4]?"), se pueden obtener todas las formas posibles en las cuales dos listas concatenadas dan como resultado la lista [1,2,3,4].

```
?- member(3,[1,2,3,4]).
Yes

?- append([1,2],[a,b],L).
L = [1, 2, a, b]
Yes

?- member(X,[1,2,3,4]).
X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
No

?- append(X,Y,[1,2,3,4]).
X = [],
Y = [1, 2, 3, 4] ;

X = [1],
Y = [2, 3, 4] ;
```

```
X = [1, 2],
Y = [3, 4] ;

X = [1, 2, 3],
Y = [4] ;

X = [1, 2, 3, 4],
Y = [] ;
No
```

Ejercicio *8* Definir un predicado para mostrar en pantalla el contenido de una lista en orden inverso.

Hasta aquí hemos dado una introducción a los principales conceptos de programación lógica. El lector interesado en profundizar en ellos puede consultar el apéndice B, donde se incluye una importante bibliografía. No obstante, en el capítulo siguiente mostraremos algunas aplicaciones de Prolog en inteligencia artificial, las cuales muestran claramente el potencial que tiene este lenguaje de programación. A estas alturas el lector comprenderá que la recursión es una estructura de control fundamental para programar en Prolog y que las listas proveen una estructura de datos muy útil y flexible. Ambas herramientas se utilizarán en las aplicaciones que introducimos en el texto siguiente.

3

Variantes interesantes de la programación lógica

A continuación ofrecemos una introducción a algunas aplicaciones de Prolog muy interesantes. Sugerimos al lector probar los códigos de los ejemplos para experimentar con la aplicación y leer la bibliografía referida para profundizar sobre estos temas.

1. Presunciones (*Assumptions)*

Del trabajo sobre lógica lineal de Girard [7] surgieron propuestas dentro de la programación lógica para incorporar información dinámica, es decir, información que se agrega en medio de la ejecución. En este capítulo introduciremos una de dichas propuestas, la presunción lineal, que se establece mediante el símbolo reservado "+" y se consulta mediante el símbolo reservado "-". Una vez hecha una presunción lineal, queda disponible durante el resto de la presente computación, hasta que sea consultada. Como efecto lateral de la primer consulta, la presunción desaparece y no se puede consultar de nuevo en lo sucesivo. También desaparece en el caso de backtrack.

Por ejemplo, podemos representar el grafo de la Figura 9 mediante presunciones lineales, como en el ejemplo siguiente:

```
grafo:- +c(1,[2,3]), +c(2,[1,4]), +c(3,[1,5]), +c(4,[1,5])
```

Y a continuación definir un programa tal que la consulta `?-buscar _camino(Xs)` nos dé los resultados esperados, es decir:

```
Xs=[1,2,4,5];
Xs=[1,3,5]
```

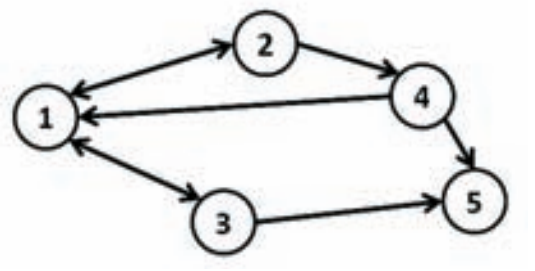

Figura 9. Grafo con ciclos descripto por el predicado `c/2`

al tiempo que evita resultados que contengan ciclos. El programa que buscamos es:

```
camino(X,X,[X]).
camino(X,Z,[X|Xs]):- conexion(X,Y),camino(Y,Z,Xs).

conexion(X,Y):- -c(X,Ys),member(Y,Ys).

buscar_camino(Xs):-
  grafo,
  camino(1,5,Xs).
```

Existen otras variantes de las presunciones: la presunción intuicionista, que se puede consultar un número indefinido de veces, y la atemporal, que puede consultarse indistintamente antes o después de haber sido establecida. Para más detalles ver [3, 5].

2. Gramáticas lógicas

2.1. Introducción

Las gramáticas lógicas son parte de la programación lógica y permiten describir un lenguaje que puede, a continuación, ser consultado para determinar si una oración pertenece o no al lenguaje descrito. También, en el caso de que la gramática en cuestión incluya una manera de construir alguna representación de la oración (ya sea sintáctica, semántica o pragmática), permiten determinar cuál es la representación que corresponde a una

oración determinada. La notación de las reglas gramaticales es parecida a la de las reglas de Prolog propiamente dichas, excepto que el símbolo ":-" se reemplaza por el símbolo de reescritura "->"; además, distinguimos dos tipos distintos de "predicados": los símbolos no terminales, es decir, los que nunca aparecen dentro de la oración a analizar; y los símbolos terminales o palabras.

Por ejemplo, el balido de una oveja es un lenguaje que se puede definir con las siguientes tres reglas de reescritura:

```
balido --> [b], es.

es --> [e].
es --> [e], es.
```

cuyos símbolos no terminales son "balido" y "es", y cuyos símbolos terminales son "b" y "e". Dicha gramática genera oraciones tales como

```
be
bee
beee
beeee
```

Para consultar una gramática necesitamos incluir un parámetro que contendrá la oración que se pretende analizar, en forma de lista de palabras (ya sea explícitamente, como por ejemplo [b,e], o implícitamente, con una variable cualquiera). Debemos, además, agregar un parámetro con la lista vacía `[]` (que por razones de orden práctico no explicaremos por qué). Por ejemplo:

```
?- balido([b,e],[]).
yes

?- balido([b,e,e,e,e],[]).
yes

?- balido([b,e,a,e,e,e],[]).
no
```

2.2. Predicados utilitarios

Mientras estamos poniendo a punto un programa, conviene elegir tests convenientes y tipearlos de una vez en nuestro programa, como se ejem-

plifica al pie. También mostramos abajo cóm hacer que Prolog testee una serie de oraciones con una sola pregunta `?- vamos`. El predicado `write/1` es reservado y simplemente tiene como resultado imprimir el valor de su argumento; `nl` (new line) es también un predicado reservado, cuyo efecto es ir a una nueva línea. El predicado `fail` es reservado y produce una "falla" adrede en la consulta para forzar el backtracking y explorar todas las posibilidades.

```
% TESTS

i([b,e,e,e]).
i([b,e]).
i([b]).
i([b,e,e,e,e]).
i(_).

vamos:- i(Input), nl, write(Input), nl, s(Input,[]), fail.
vamos.
```

2.3. Análisis versus generación

Algunas gramáticas lógicas pueden consultarse tanto en modo análisis (como hemos hecho hasta ahora) como en modo generación. Por ejemplo, podemos generar balidos:

```
?- balido(X,[]).
X=[b,e];
X=[b,e,e]);
X=[b,e,e,e])
yes
```

2.4. Gramáticas para interfaces de lengua natural

Como se verá a continuación, es posible generar aplicaciones más interesantes que el lenguaje de las ovejas. Por ejemplo, una gramática que nos permitiera consultar en lenguaje natural (castellano) nuestra base de conocimiento cosmológica de la Figura 2. A continuación mostramos una gramática simple que nos permite hacer preguntas de tipo verificación solamente.

```
% Sintaxis

 s --> np, vp.

 np --> d, n.
 np --> nom.

% Tres tipos de verbos: intransitivos, transitivos,
  bitransitivos

 vp  --> iv.
 vp  --> tv, np.
 vp  --> bv, np, pp.

 pp  --> p, np.

% Lexico

d --> [el].
d --> [la].

p --> [sobre].

n --> [sol].
n --> [luna].
n --> [mundo].

nom --> [gaia].
nom --> [helios].

iv --> [brilla].

tv --> [refleja].
tv --> [illumina].

bv --> [refleja].

% TESTS
```

```
i([la,luna,brilla]).
i([la,luna,refleja,el,sol]).
i([la,luna,refleja,el,sol,sobre,el,mundo]).
i([helios,illumina,a,gaia]).

vamos:- i(Input), nl, write(Input), nl, s(Input,[]), fail.
vamos.
```

Hay que notar que esta gramática está pensada sólo para usarse en modo análisis; en modo generación "sobregenera", es decir, genera oraciones que no pertenecen a nuestro lenguaje deseado, como por ejemplo: "la sol brilla sobre el luna". Para evitar desacuerdos gramaticales como este, podríamos modificar toda la gramática agregando en lugares relevantes información de género, por ejemplo:

```
d(masculino) --> [el].
d(femenino)--> [la].

p --> [sobre].

n(masculino) --> [sol].
n(femenino) --> [luna].
n(masculino) --> [mundo].

np(Genero) --> d(Genero), n(Genero).
```

Esto obligaría a un determinante y un nombre en la misma frase nominal a tener el mismo género.

2.5. Obtención de representaciones semánticas como efecto lateral

Podemos también construir información semántica con el uso de argumentos suplementarios. Por ejemplo, la siguiente gramática obtiene, como efecto lateral de aceptar una oración en el lenguaje que describe, su representación semántica:

```
% Reglas sintactico-semanticas
```

```
  s(Significado) --> np(X), vp(X,Significado).

  np(X) --> nom(X).

% Three types of verbs: intransitive, transitive,
  bitransitive

  vp(X,M)  --> iv(X,M).
  vp(X,M)  --> tv(X,Y,M), np(Y).
  vp(X,M)  --> tv(X,Y,M), pp(Y).
  vp(X,M)  --> bv(X,Y,Z,M), np(Y), pp(Z).

  pp(X)  --> prep, np(X).

% Lexicon

  nom(gaia) --> [gaia].
  nom(helios) --> [helios].
  nom(selene) --> [selene].

  iv(X,brilla(X)) --> [brilla].

  tv(X,Y,refleja(X,Y)) --> [refleja].
  tv(X,Y,ilumina(X,Y)) --> [ilumina].

  bv(X,Y,Z,refleja_sobre(X,Y,Z)) --> [refleja].

  prep --> [sobre].


% TESTS

  i([helios,ilumina,a,gaia]).
  i([selene,refleja,a,helios,sobre,gaia]).
  i([selene,refleja,a,helios]).

  vamos:- i(Input), nl, write(Input), nl, s(M,Input,[]),
```

```
        write(M), nl, fail.
  vamos.
```

Ejercicio 9 Testear la gramática dada para ver si obtiene resultados tales como:

```
M=ilumina(helios,gaia)

M=refleja_sobre(selene,helios,gaia)

M=refleja(selene,helios)
```

2.6. Consulta de bases de conocimiento en lengua natural

Hay que recalcar que, con esto, estamos a un paso de consultar bases de conocimiento como la cosmológica que hemos dado, a partir del castellano. Simplemente cargamos tanto la base de datos como la gramática en la memoria de Prolog y pedimos el significado de una oración que contiene una variable para luego dar ese significado a ser evaluado por la base de datos. Así, para preguntar a quién ilumina helios, escribimos:

```
?- s(Semantica,[helios,ilumina,a,X],[]), call(Semantica),
   write(X).
```

El primer predicado obtiene el valor `ilumina(helios,X)` para la variable `Semantica`, y el segundo, que usa el predicado reservado "`call/1`", evalúa dicha expresión, con lo cual obtenemos el resultado: `X=gaia.`

A continuación incluimos varios apéndices: el primero, con las respuestas a algunos de los ejercicios propuestos; el segundo, con algunos libros sugeridos para profundizar sobre los temas presentados en este texto, y el tercero, con enlaces de interés.

4

Referencias bibliográficas

1. K. Apt. *From Logic Programming to Prolog*. Prentice-Hall, 1997.
2. Ivan Bratko. *PROLOG Programming for Artificial Intelligence, Second Edition*. Addison-Wesley, 1990.
3. H. Christiansen and V. Dahl. Assumptions and abduction in prolog. In *Proc. of MULTICPL'04. Third International Workshop on Multiparadigm Constraint Programming Language*, 2004.
4. M.S. Dawe C.M. Dawe. *Prolog for Computer Science.* Springer-Verlag, 1994.
5. V. Dahl and P. Tarau. Assumptive logic programming. In *Proc. of ASAI'04, Cordoba, Argentina*, 2004.
6. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.
7. J.-Y. Girard. Linear logic. In *Theoretical Computer Science*, volume 50, pages 1–102, 1987.
8. Quintus Prolog Home Page. http://www.sics.se/isl/quintuswww/site/biblio.html.
9. Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed*. MIT Press, 1994.

Parte II

Apéndices

A

Apéndice I: Respuestas a algunos de los ejercicios

Ejercicio 1: `_` es una variable (anónima); `x` es una constante (átomo); `X` es una variable; `xVARIABLE` es una constante (átomo); `2` es una constante (número); `_dos` es una variable; `dos_` es una constante (átomo); `'t(X)'` es una constante (átomo entre comillas); `t(X)` es un término compuesto; `es par` no es válido; `{a}` no es válido.

Ejercicio 2:

```
email(asunto(prolog),de(ajg), para(agarcia),
        fecha((d(28),m(8),a(2008)),hora(h(9),m(28))),
        servidor(gmail),
        contenido('Hay un nuevo intérprete de Prolog')
     ).
```

Ejercicio 3: Al aplicar `{X=a,Y=f(a)}` a `p(f(X),Y)` resulta `p(f(a),f(a))`. Al aplicar `{X=que,Y='hola'}` a `sms(Y,X,tal)` resulta `sms('hola',que,tal)`.

Ejercicio 4: (a) `p(X,f(Y),h(Y,X))` y `p(Z,Q,h(a,b)` unifican con `{X = b, Y = a, Z = b, Q = f(a) }`. (b) `p(a,b)` y `p(X,Y,Z)` no unifican porque los términos son de distinta aridad. (c) `p(a,b)` y `t(X,Y)` no unifican porque tienen diferente functor. (d) `p(a,b)` y `(b,X)` no unifican porque el primer argumento tiene dos constantes distintas.

Ejercicio 5:

```
% X es primo de Y, si X es hombre y sus padres son hermanos
```

```
primo(X,Y):- hombre(X), progenitor(X,Px),
             progenitor(Y,Py), hermano(Px,Py).

% X es prima de Y, si X es mujer y sus padres son hermanos
prima(X,Y):- mujer(X), progenitor(X,Px),
             progenitor(Y,Py), hermano(Px,Py).

%dos individuos son hermanos si comparten un progenitor.
hermano(X,Y):- progenitor(X,P),progenitor(Y,P).
```

Ejercicio 6:

```
% Una opción
par(0).
par(s(s(X))):-par(X).
% Otra opción:
espar(0).
impar(s(X)):-espar(X).
espar(s(X)):-impar(X).
```

Ejercicio 7: A continuación se muestran tres opciones. En la primera, el predicado `mostrar_camino/2` mostrará únicamente los nodos del camino, pero en orden inverso al recorrido. En la segunda, el predicado `mostrar/2` mostrará en el orden de la búsqueda, pero todos los nodos visitados, esto es, aquellos nodos que no son parte del camino, pero fueron visitados, también son mostrados. Finalmente, el predicado `traza_busqueda/2` mostrará primero el recorrido de búsqueda, usando el símbolo "?"; y finalmente el camino hallado (en orden inverso).

```
mostrar_camino(N1,N2):-arco(N1,N2),write(-(N2)).
mostrar_camino(N1,N2):-arco(N1,X),mostrar_camino(X,N2),
                       write(-(X)).

mostrar(N1,N2):-write(+(N1)),arco(N1,N2).
mostrar(N1,N2):-arco(N1,X),write(+(X)),mostrar(X,N2).

traza_busqueda(N1,N2):-write('?'(N1)),arco(N1,N2),
                       write(+(N2)).
```

```
traza_busqueda(N1,N2):-arco(N1,X),write('?'(X)),
                       traza_busqueda(X,N2),write(-(X)).
```

Ejercicio 8:

```
mostrar_inv([]).
mostrar_inv([X|Xs]):-mostrar_inv(Xs),write(X),write(',').
```

Ejercicio 9: Use algún intérprete de Prolog, cargue el programa anterior al ejemplo y ejecute la consulta `vamos/0`. En la pantalla deberá ver las tres instancias mostradas en el enunciado. Puede cambiar el predicado `i/1` para experimentar con otros tests.

B

Apéndice II: Algunos libros de Prolog y programación lógica

A continuación incluimos una lista de algunos libros de texto sobre Prolog, programación lógica, Prolog e inteligencia artificial, y otras aplicaciones y ramificaciones de programación lógica. Esta lista de libros fue obtenida de [8], donde puede consultarse pora conseguir más material.

B.1. Algunos libros de texto sobre Prolog

- Prolog Programming for Students. D. Caellar, Continuum, 2001. ISBN: 0-826-45496-8
- Clause and Effect : Prolog Programming for the Working Programmer. W.F. Clocksin, Springer-Verlag, 1997. ISBN: 3-540-62971-8
- Prolog: The Standard. P. Deransart et al, Springer-Verlag, 1996. ISBN: 3-540-59304-7
- Mastering Prolog. R.J. Lucas, UCL Press, 1996. ISBN: 1-857-28400-3
- Prolog Programming in Depth. (2nd edition). M.A. Covington et al. Prentice-Hall, 1996. ISBN: 0-131-38645-X
- Programming in Prolog. (4th edition). Clocksin, W.F. and C.S. Mellish. Springer-Verlag, 1994. ISBN: 3-540-58350-5
- The Art of Prolog. (2nd edition). Sterling, L. and E. Shapiro. The MIT Press, 1994. ISBN: 0-262-19338-8
- Prolog for Computer Science. C.M. Dawe, M.S. Dawe. Springer-Verlag, 1994. ISBN: 0-387-19811-3
- The Logic Programming Tutor. J. Paine, Kluwer, 1992. ISBN: 1-871-51609-9
- An Introduction to Programming in Prolog. P. Saint-Dizier, Springer-Verlag, 1990. ISBN: 0-387-97144-0

- Prolog: A Logical Approach. Dodd, T. Oxford University Press, 1990. ISBN: 0-198-53821-9
- The Craft of Prolog. O'Keefe, R.A. The MIT Press, 1990. ISBN: 0-262-15039-5
- Advanced Prolog. Ross, P. Addison-Wesley, 1989. ISBN: 0-201-17527-4
- Prolog by Example. H. Coelho, J.C. Cotta, Springer-Verlag, 1988. ISBN: 0-387-18313-2
- Prolog for Programmers. Kluzniak, F. and Szpakowicz, S. Academic Press, 1985. Logic Programming

B.2. Algunos libros sobre programación lógica

- From Logic Programming to Prolog. K. Apt. Prentice-Hall, 1997. ISBN: 0-132-30368-X
- An Introduction to Logic Programming Through Prolog. J.M. Spivey, M. Spivey, Prentice-Hall, 1996. ISBN: 0-135-36047-1
- Logic: The Basis for Understanding Prolog. D.E. Cooke. Ablex, Norwood NJ, 1994. ISBN: 1-567-50028-5
- Foundations of Logic Programming (2nd edition). J.W. Lloyd, Springer-Verlag, 1993. ISBN: 3-540-18199-7
- Essentials of Logic Programming. C. Hogger, Clarendon Press, Oxford, 1990. ISBN: 0-198-53832-4
- Logic, Programming and Prolog. U. Nilsson, J. Maluszynski, Wiley, 1990. ISBN: 0-471-92625-6
- Prolog Versus You. A.-L. Johansson, Chartwell-Bratt, 1989.
- Computing With Logic : Logic Programming With Prolog. D. Maier, D.S. Warren, Benjamin Cummings, 1988.
- Introduction to Logic Programming. C. Hogger, Academic Press, 1984. ISBN: 0-123-52092-4
- Logic for Problem Solving. Kowalski, R. Artificial Intelligence Series, North Holland, 1979. General Applications

B.3. Prolog e inteligencia artificial

- Artificial Intelligence Techniques in Prolog. Shoham, Y. Morgan Kaufmann, 1993. ISBN: 1-558-60167-8
- Cognitive Science Projects in Prolog. P. Scott, R. Nicolson, Lawrence Erlbaum Assoc, 1991 ISBN: 0-863-77181-5
- Knowledge Systems and Prolog. Walker, A., M. McCord, J. Sowa, and W. Wilson. Addison-Wesley, 1990. ISBN: 0-201-52424-4

- Prolog Programming for Artificial Intelligence. (2nd edition). Bratko, CITE. Addison-Wesley, 1990. ISBN: 0-201-41606-9
- Artificial Intelligence through Prolog. Rowe, N.C. Prentice-Hall, 1988. ISBN: 0-201-41606-9
- Artificial Intelligence : Structures and Strategies for Complex Problem Solving. (2nd edition). G.F. Luger, W.A. Stubblefield, Addison-Wesley, 1997. ISBN: 0-805-31196-3
- Simply Logical: Intelligent Reasoning by Example. P. Flach, John Wiley & Sons, 1994. ISBN: 0-471-94152-2
- Knowledge Systems Through Prolog : An Introduction . S.H. Kim, Oxford University Press, 1991. ISBN: 0-195-07241-3
- Prolog and Expert Systems. Bowen, K.A. McGraw-Hill, 1991. ISBN: 0-070-06731-7
- Building Expert Systems in Prolog. Merritt, D. Springer-Verlag, 1989. ISBN: 0-387-97016-9
- Expert Systems Lab Course. Schnupp, P., C.T. Nguyen Huu and L.W. Bernhard. Springer-Verlag, 1989. ISBN: 0-387-50570-9
- An Introduction to Natural Language Processing Through Prolog. C. Matthews, Pearson, 1998. ISBN: 0-582-06622-0
- Natural Language Processing for Prolog Programmers. Covington, M.A. Prentice Hall, 1994. ISBN: 0-136-29213-5
- Natural Language Computing - An English Generative Grammar in Prolog. R.C. Dougherty, Lawrence Erlbaum Co., 1994 ISBN: 0-805-81526-0
- Functional Grammar in Prolog : An Integrated Implementation for English, French, and Dutch. S.C. Dik, Lawrence Erlbaum Assoc, 1991. ISBN: 0-863-77181-5
- Prolog for Natural Language Processing. Gal, A., G. Lapalme, P. Saint-Dizier and H. Somers. Wiley, 1991. ISBN: 0-471-93012-1
- Natural Language Processing in Prolog. Gazdar, G., and C.S. Mellish. Addison-Wesley, 1989. ISBN: 0-201-18053-7
- Logic Grammars. H. Abramson and V. Dahl, Springer-Verlag, 1989. ISBN: 0-387-96961-6
- Prolog and Natural-Language Analysis. F.C.N. Pereira and S.M. Shieber, CSLI Publications, 1987. ISBN: 0-937-07318-0

B.4. Otras aplicaciones de Prolog

- Intelligent Image Processing in Prolog. B.G. Batchelor, Springer-Verlag, 1991. ISBN: 3-540-19647-1
- Eco-Logic: Logic-Based Approaches to Ecological Modelling. Robertson, D., A. Bundy, R. Muetzelfeldt, M. Haggith, and M. Uschold. The MIT Press, 1991. ISBN: 0-262-18143-6

- Logic Programming, Systematic Program Development. Y. Deville, Addison-Wesley, 1990. ISBN: 0-201-17576-2
- Logic Programming and Databases. Ceri, S., G. Gottlob and L. Tanca. Springer-Verlag, 1990. ISBN: 0-387-51728-6
- The Practice of Prolog. Sterling, L. The MIT Press, 1990. ISBN: 0-262-19301-9
- Why Prolog? Justifying Logic Programming for Practical Applications. G.L. Lazarev, Prentice Hall, 1989. Structured Systems Analysis Through Prolog. Goble, T. Prentice Hall, 1989. ISBN: 0-138-53581-7
- Prolog Programming and Applications. Burnham, W.D. and Hall, A.R. Macmillan, 1985.
- Start Problem Solving with Prolog. T. Conlon, Addison-Wesley, 1985.

C

Apéndice III: Intérpretes y otros sitios de interés

En este apéndice se incluyen algunos sitios donde poder consultar sobre intérpretes de Prolog de distribución gratuita o sitios de información sobre Prolog y programación lógica.

- SWI-Prolog: `http://www.swi-prolog.org`
- Bin Prolog: `http://www.binnetcorp.com/BinProlog/`
- Ciao Prolog: `http://www.clip.dia.fi.upm.es/Software/Ciao/`
- Una comparación sobre intérpretes de Prolog: `http://www.fraber.de/university/prolog/comparison.html`
- Otros links de interés: `http://www.sics.se/isl/sicstuswww/site/links.html`

TRIANGLE is a periodical published by the Department of Romance Studies. It aims to present the results of interdisciplinary research which adopts new approaches to understanding language sciences.

La Programación Lógica es una disciplina que conecta el lenguaje natural, la lógica y los lenguajes de programación. Esta característica la hace muy adecuada para desarrollar aplicaciones de Inteligencia Artificial con suma facilidad y elegancia. Este volumen es una introducción básica a la LP que intenta combinar el rigor con la claridad, ofreciendo vías para la profundización en este paradigma.

UA
UNIVERSIDAD AUTONOMA
DE MADRID

Universitat Rovira i Virgili
Departament de Filologies Romàniques