Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2019-20

Parte III Programación Lógica

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Práctica 7: Introducción a Prolog

Índice

1. Objetivos de la Práctica

Los objetivos de la práctica son dos:

1. Introducir el uso de un lenguaje lógico, concretamente Prolog, para representar una base de conocimiento y realizar consultas sobre ella.

2. Profundizar en dos conceptos básicos del paradigma lógico que permiten resolver las consultas: los conceptos de *unificación* y *búsqueda con retroceso (o vuelta atrás)*, e introducir un mecanismo para realizar cálculos aritméticos simples.

2. SWI-Prolog

SWI-Prolog es una implementación en código abierto del lenguaje de programación Prolog. El nombre SWI deriva de Sociaal-Wetenschappelijke Informatica, el antiguo nombre de un grupo de investigación en la Universidad de Amsterdam donde se inició su desarrollo.

2.1. Carga y ejecución de programas

SWI-Prolog está instalado en los laboratorios de prácticas y es accesible desde cualquier directorio escribiendo el comando `swipl` en la terminal Linux.

Abre un terminal y crea un directorio de trabajo donde descargarás y editarás los programas (p. ej., `DiscoW/LTP/pr7`). Desde Linux, accede a dicho directorio e invoca a SWI-Prolog mediante el comando `swipl`:

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.6.1)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

?-
```

El sistema ejecuta un intérprete que queda a la espera de que introduzcamos llamadas de ejecución al programa Prolog cargado en memoria (en la terminología Prolog, consultas a la base de conocimiento). En SWI-Prolog estas consultas pueden realizarse haciendo uso de un shell interactivo cuyo prompt es `?-`. La carga de programas guardados en un fichero puede hacerse desde esta misma línea de entrada de consultas usando cualquiera de las órdenes `consult(`*nombrefichero*`)`, `compile(`*nombrefichero*`)` o `[`*nombrefichero*`]` seguidas de un punto, donde *nombrefichero* es el nombre del fichero **sin incluir la extensión**. Podemos salir del intérprete mediante el comando `halt.` o bien pulsando CTRL+D.

Ejercicio 1 *Baja el fichero* `cars.pl` *que está en* PoliformaT *y cárgalo en Prolog haciendo uso de alguna de las instrucciones de carga anteriores. ¡No olvides que todas las consultas finalizan con un punto!*

El resultado debe parecerse al siguiente:

```
consult(cars).
% cars compiled 0.00 sec, 49 clauses
true.
```

Podemos comprobar el código que hemos cargado mediante el comando `listing`. También podemos utilizarlo para conocer la definición de algún predicado en particular, p. ej: `listing(model)`.

Ejercicio 2 *Ejecuta el predicado* `listing` *desde el intérprete* ***`swipl`*** *y lista la base de conocimiento para varios de los predicados cargados en memoria.*

Deberías haber obtenido algo parecido a esto:

```
?- listing.

:- thread_local thread_message_hook/3.
:- dynamic thread_message_hook/3.
:- volatile thread_message_hook/3.

brand(B, A) :-
        model(A, B).

isModelFrom(A, C) :-
        model(A, B),
        country(B, C).

since(ibiza, 1984).
since(cordoba, 1993).
since(altea, 2004).
since(golf, 1974).
since(touran, 2003).
since(clio, 1990).
since(twingo, 1993).
since(mégane, 1995).
since(scénic, 1995).
since('2008', 2013).
since('3008', 2008).
since(corsa, 1982).

segment(ibiza, b).
segment(cordoba, b).
segment(altea, c).
segment(golf, c).
segment(touran, c).
segment(clio, b).
segment(twingo, a).
segment(mégane, c).
segment(scénic, c).
segment('2008', b).
segment('3008', c).
```

```
segment(corsa, b).

isRelated(A, B) :-
        isSameBrand(A, B).
isRelated(A, B) :-
        isSameYear(A, B).
isRelated(A, B) :-
        segment(A, C),
        segment(B, C),
        A\==B.

model(ibiza, seat).
model(cordoba, seat).
model(altea, seat).
model(golf, volkswagen).
model(touran, volkswagen).
model(clio, renault).
model(twingo, renault).
model(mégane, renault).
model(scénic, renault).
model('2008', peugeot).
model('3008', peugeot).
model(corsa, opel).

country(seat, españa).
country(renault, francia).
country(peugeot, francia).
country(volkswagen, alemania).
country(opel, alemania).

isSameBrand(A, B) :-
        model(A, C),
        model(B, C),
        A\==B.

isSameYear(A, B) :-
        since(A, C),
        since(B, C),
        A\==B.
true.
```

En primer lugar, es importante destacar que el uso de mayúsculas o minúsculas es relevante para el compilador. Las variables empiezan por mayúsculas o por un símbolo de subrayado. Por tanto, `A` es una variable mientras que `ibiza` no lo es. Además, podemos ver que un programa está formado por un conjunto de:

- hechos, por ejemplo `model(ibiza,seat).`
- reglas, por ejemplo `brand(B, A) :- model(A, B).`

El predicado `model(ibiza,seat)` expresa el hecho de que `ibiza` es *un modelo de* `seat`. La definición de este predicado es *extensional*, es decir, en el programa aparecen explícitamente los casos que definen la relación.

El predicado `brand(B,A)` se define, de forma *intensional*, mediante la regla:

```
brand(B,A) :- model(A,B).
```

que expresa que `B` *es marca* de `A` *si* `A` *es modelo de* `B`. Es decir, podemos resolver el problema de saber cuál es la marca de un modelo consultando la relación `model`. Fíjate que el *si* condicional de la frase anterior se expresa en el programa con el par de símbolos `:-`.

El predicado `isRelated(A, B)` se define intensionalmente mediante tres cláusulas de tipo regla que expresan que `A` está relacionado con `B` *si* `A` y `B` son de la misma marca, o del mismo año de lanzamiento, o del mismo segmento (tipo de coche). Nota que el uso de varias reglas con la misma cabeza (en este ejemplo, `isRelated`) es una forma simple de expresar la disyunción en Prolog.

Finalmente, las definiciones de los predicados `isSameBrand` e `isSameYear` contienen tres predicados separados por comas en el cuerpo de la regla (detrás de `:-`). Las comas representan una conjunción de predicados, lo que implica que deben ejecutarse todos con éxito para que la conjunción sea cierta.

2.2. Consultas a los hechos

A continuación vamos a realizar varias consultas usando el intérprete sobre la base de conocimiento que ya tenemos cargada en memoria. Empezamos realizando consultas sencillas sobre los hechos del programa. Por ejemplo, preguntar si, de acuerdo con la información que consta en memoria, `clio` es un modelo de `renault`, se escribiría así:

```
?- model(clio,renault).
true.
```

Ejercicio 3 *¿Qué ocurre cuando preguntas si* `renault` *es un modelo de* `clio`*? Escribe la consulta y explica la respuesta que devuelve el intérprete.*

Las dos consultas anteriores tenían como respuesta una afirmación (`true`) o una negación (`false`), pero usando *variables lógicas* podemos realizar consultas que extraigan información más jugosa de la base de conocimiento. Por ejemplo, imagina que queremos saber si `clio` es un coche de alguna marca, y en tal caso, saber cuál es. Para ello podemos lanzar al intérprete la misma consulta que antes, pero usando una simple variable `X` como segundo argumento. Como respuesta se obtiene el valor de la variable que hace cierta la relación.

```
?- model(clio,X).
X = renault.
```

Es decir, a partir de los hechos del programa, el intérprete puede demostrar que `model(clio,X)` es cierto si `X = renault`.

En el caso de que exista más de una respuesta a la consulta planteada, el sistema retorna la primera que computa. Si deseamos recibir más respuestas debemos pulsar *r* (*redo*) para conocer otras respuestas. Compruébalo realizando el siguiente ejercicio.

Ejercicio 4 *Pregunta cuáles modelos son de* `renault` *sustituyendo, en la consulta del ejemplo anterior, la variable* `X` *por la constante* `renault` *y la constante* `clio` *por una variable (por ejemplo* `X`*). ¿Qué ocurre cuando pulsas la tecla* `r` *después de la primera respuesta?*

La siguiente ejecución ilustra el uso de dos variables para extraer de la base de conocimiento todos los pares `(X,Y)` donde `X` *es una marca del país* `Y` (los puntos y comas aparecen en el terminal tras pulsar la tecla `;` o la tecla `r`):

```
?- country(X,Y).
X = seat,
Y = españa ;
X = renault,
Y = francia ;
X = peugeot,
Y = francia ;
X = volkswagen,
Y = alemania ;
X = opel,
Y = alemania.
```

Ejercicio 5 *Realiza la consulta* `country(X,X).` *¿Qué contesta el intérprete? Explica esta respuesta.*

En esta sección hemos realizado consultas que solo requerían determinar si la relación entre un par de términos (`renault`, `clio`; `opel`, `alemania`, ...) está dada por un hecho (`model`; `country`). En el caso de usar variables, éstas se instanciaban con los términos que hacían el hecho cierto.

2.3. Consultas a las reglas

Ahora vamos a realizar consultas que requieren deducir información que no está expresada con hechos en el programa pero que puede obtenerse a partir de los hechos y las reglas. Recordemos que las reglas permiten expresar que "Si es verdad el antecedente (cuerpo) entonces es verdad el consecuente (cabeza)". Por ejemplo, haciendo uso de la regla `brand(A,B) :- model(B,A).` que expresa que si `B` es modelo de `A` entonces `A` es marca de `B`. Sabiendo que `ibiza` es un modelo de `seat`, el intérprete puede llegar a la conclusión de que `seat` es la marca de `ibiza` tras la siguiente consulta:

```
?- brand(seat,ibiza).
true.
```

También se puede preguntar cuáles son modelos de `seat`:

```
?- brand(seat,X).
X = ibiza ;
X = cordoba ;
X = altea.
```

o los de `opel`:

```
?- brand(opel,X).
X = corsa.
```

Ninguno de estos hechos `brand(seat,ibiza)`, `brand(seat,cordoba)`, `brand(seat,altea)`, `brand(opel,corsa)` está explícitamente en el programa. Sin embargo, el intérprete es capaz de llegar a la conclusión de que son deducibles de los hechos y reglas del programa.

Ejercicio 6 *Haciendo uso de* `cars.pl`*:*

- *Consultar cuáles modelos son de* `alemania`*.*
- *Consultar todos los modelos relacionados con el modelo* `cordoba`*.*

Ejercicio 7 *Usando un editor de texto, añadir al fichero* `cars.pl` *las siguientes reglas:*

- *Una regla que defina la relación* `isCountryOf` *de forma similar a* `brand` *pero usando el predicado* `isModelFrom` *en lugar de* `model`*. Guarda los cambios y recarga el fichero con el predicado* `reconsult(cars)`*. Averigua de cuál país es el modelo* `mégane` *usando el predicado* `isCountryOf`*.*
- *Una regla que defina la relación* `isClassic` *que indique si un modelo fue comercializado antes del año 1995. Averigua cuáles modelos son clásicos usando el predicado* `isClassic`*.*

Ejercicio 8 *Prueba y modifica la regla* `isRelated(A,B)`*, que indica si dos modelos están relacionados porque sean de la misma marca, del mismo segmento, o del mismo año de lanzamiento:*

- *Lanza la consulta* `isRelated(golf,X)`*,*
- *Amplía la relación* `isRelated(A,B)`*, añadiendo al programa una regla que indique que dos modelos están relacionados sin ambos son clásicos (usando* `isClassic`*).*
- *Repite la consulta* `isRelated(golf,X)`*. En qué difieren las dos respuestas computadas?*

3. Dos conceptos básicos en Prolog

Cada paradigma de programación basa su semántica operacional en distintos conceptos, como por ejemplo la noción de cambio de estado de los lenguajes imperativos o la reducción de términos usando ajuste de patrones de los lenguajes funcionales.

En esta sección vamos a practicar dos de los conceptos fundamentales de los lenguajes lógicos: la unificación y la búsqueda con retroceso o vuelta atrás[1].

3.1. Unificación de términos

A diferencia de los programas funcionales, en los programas lógicos no existe un conjunto de ecuaciones que se pueda usar para reescribir términos. Sin embargo, la ejecución de un programa lógico sí que manipula términos[2] que, además, pueden contener variables. Esto ofrece la posibilidad de extraer información a través de las variables que pueden aparecer dentro de los términos en los predicados consultados. El mecanismo bidireccional que hace posible que los parámetros del predicado puedan ser de entrada o de salida se llama unificación de términos.

En clases de teoría se estudia un algoritmo que determina si dos términos con variables unifican y, en el caso de hacerlo, devuelve un unificador (el más general). Este algoritmo puede verse como un mecanismo de paso de parámetros bidireccional que generaliza el ajuste de patrones de los lenguajes funcionales permitiendo dar valores no solo a las variables del programa sino también a las de los términos de la consulta.

Informalmente, dos términos con variables unifican si su estructura es compatible y hay un valor de sus variables que los hace idénticos. Es decir, tras dar valor a las variables, los términos resultantes contienen exactamente los mismos símbolos, o simplemente se diferencian por un renombramiento de las variables. Veamos algunos ejemplos:

[1] También conocido como *backtracking*

[2] En el apéndice 4 puedes encontrar una clasificación de dichos términos

- Los términos `date(10,nov,2030)` y `date(10,nov,2030)` unifican ya que son idénticos.
- `date(10,nov,2030)` y `date(X,nov,2030)` unifican si `X` = 10.
- `date(10,nov,2030)` y `date(X,nov,X)` no unifican porque `X` no puede valer 10 y 2030 a la vez.
- `date(10,nov,2030)` y `time(13,05)` no unifican porque `date` y `time` son dos functores[3] distintos.
- `X` y `time(13,05)` unifican si `X` = `time(13,05)`.
- `X` y `date(10,nov,Y)` unifican si `X` = `date(10,nov,Y)`.
- `date(10,oct,2030)` y `date(X,nov,2030)` no unifican porque los meses son distintos.
- los términos `moment(date(10,nov,2030),Y)` y `moment(X,time(13,05))` unifican si `X` = `date(10,nov,2030)` e `Y` = `time(13,05)`.
- los términos `moment(time(Time,Minutes))` y `moment(time(13,05))` unifican si `Time` = 13 y `Minutes` = 05.

Para que dos predicados unifiquen han de tener el mismo nombre, la misma aridad y deben unificar cada uno de los términos contenidos como argumentos. Por ejemplo:

- los hechos `model(X,seat)` y `model(altea,seat)` unifican si `X` = `altea`.
- los hechos `model(clio,X)` y `model(clio,renault)` unifican si `X` = `renault`.
- los hechos `model(X,seat)` e `brand(X,seat)` no unifican porque los nombres de los predicados son distintos.
- los hechos `model(X,seat)` y `model(ibiza,Y)` unifican si `X` = `ibiza` e `Y` = `seat`.

Ejercicio 9 *Comprueba alguna de las afirmaciones que se hacen en los ejemplos anteriores ejecutando el algoritmo de unificación usado por el intérprete SWI-Prolog. Para ello elige pares de términos y colócalos a la izquierda y derecha del símbolo igual a modo de consulta. Por ejemplo:*

```
date(10,nov,2030) = date(X,nov,2030).
model(time(Time,moment)) = moment(time(13,05)).
```

Ejercicio 10 *Realiza lo mismo que en el ejercicio anterior pero usando ahora predicados. Por ejemplo:* `model(X,volkswagen) = model(golf,Y).`

Unificación con términos compuestos

A continuación vamos a practicar la unificación con predicados que contienen términos compuestos en sus argumentos. Para ello, usaremos el fichero `flights.pl` que está en *PoliformaT* y que contiene información sobre vuelos. Para cada vuelo directo existe un hecho representado por el predicado `flight` con ocho parámetros: origen, destino, fecha de salida, hora de salida, fecha de llegada, hora de llegada, duración y precio. Todas las fechas se representan con el functor `date` que agrupa tres términos como argumentos: el día, mes y año (por ejemplo:

[3]Se llama functor al nombre de un predicado o de una función.

`date(10,nov,2030)`). Todos los parámetros que representan una hora se escriben con el functor `time` y dos argumentos (hora y minutos). Un ejemplo de un hecho completo representando un vuelo es el siguiente:

```
flight(barcelona,madrid,
       date(10,nov,2030),time(13,05),
       date(10,nov,2030),time(15,05),
       120,80).
```

en el que se indica que existe un vuelo directo desde Barcelona a Madrid con salida el 10 de noviembre de 2030 a las 13:05 y llegada el mismo día a las 15:05 con una duración de 120 minutos a un precio de 80 euros. El programa contiene los cinco siguientes hechos:

```
flight(barcelona,madrid,
       date(10,nov,2030),time(13,05),
       date(10,nov,2030),time(15,05),
       120,80).
flight(barcelona,valencia,
       date(10,nov,2030),time(13,05),
       date(10,nov,2030),time(15,05),
       120,20).
flight(madrid,london,
       date(10,nov,2030),time(16,05),
       date(10,nov,2030),time(17,35),
       90,140).
flight(valencia,london,
       date(10,nov,2030),time(16,05),
       date(10,nov,2030),time(17,35),
       90,50).
flight(madrid,london,
       date(10,nov,2030),time(23,05),
       date(11,nov,2030),time(00,25),
       80,50).
```

Ejercicio 11 *Carga el fichero* `flights.pl` *y busca todos los vuelos desde Valencia a Londres. Usa los nombres de variables adecuados para que la salida sea:*

```
DepartureDay = ArrivalDay, ArrivalDay = date(10, nov, 2030),
DepartureTime = time(16, 5),
ArrivalTime = time(17, 35),
Duration = 90,
Price = 50.
```

Ejercicio 12 *Realiza las siguientes consultas.*

- *Consulta todos los vuelos que salen desde Madrid el día 10 de noviembre de 2030. Para ello debes realizar una consulta en la que el tercer parámetro del predicado* `flight` *use el functor* `date` *con los parámetros de la fecha solicitada.*
- *Consulta los vuelos cuya hora de salida sea 13:05. Has de realizarlo de la misma forma que el ejercicio anterior pero con el functor* `time`*.*

- *Consulta los vuelos que salen a partir de las 16:00. Para ello debes realizar una consulta en la que el término del cuarto parámetro del predicado* `flight` *conste del functor* `time` *con dos variables (*`H` *y* `M`*) como parámetros. Después y separado por una coma, has de exigir que la hora sea mayor o igual que 16* `(H >= 16)`*. Recuerda que la coma entre predicados representa la conjunción lógica.*

Observa que el programa también contiene una regla que define el predicado de aridad 3 `connection_same_day` para representar vuelos con una única escala en el mismo día. Este predicado tiene tres parámetros para el origen, destino y la fecha. Fíjate como en el cuerpo de la cláusula aparece una nueva variable `Connection` que representa la ciudad del enlace aéreo. Los parámetros con símbolo subrayado (llamado variable anónima) no son relevantes para la resolución del problema. El cuerpo de este predicado usa el predicado `is` para evaluar expresiones aritméticas asignando a la variable de la izquierda el resultado de evaluar la expresión de la derecha. En esta expresión se suman los minutos transcurridos desde el inicio del día, dejando 60 minutos para poder realizar el enlace. La última condición en el cuerpo de la cláusula es que el viajero tenga suficiente tiempo para coger el vuelo.

```
connection_same_day(Origin,Destination,Date):-
    flight(Origin,Connection,Date,_,Date,time(Hl1,Ms1),_,_),
    flight(Connection,Destination,Date,time(Hs2,Ms2),Date,_,_,_),
    Hl1_in_minutes is Hl1 * 60 + Ms1 + 60,
    Hs2_in_minutes is Hs2 * 60 + Ms2,
    Hl1_in_minutes =< Hs2_in_minutes.
```

Ejercicio 13 *Consulta todos los vuelos con una única escala que pueden realizarse el día 10 de Noviembre de 2030.*

Ejercicio 14 *¿Qué ocurriría si la cuarta y última aparición de la variable* `Date` *en el cuerpo de la cláusula* `connection_same_day` *se reemplazara por otra variable* `Another_date`*?*

3.2. Búsqueda con retroceso

Hemos visto que el intérprete proporciona todas las soluciones que son deducibles de los hechos y reglas que definen predicados en un programa Prolog. Estas soluciones van apareciendo (se van deduciendo) una tras otra a medida que el usuario las solicita. Para ello se exploran todas las posibilidades haciendo uso tanto de los hechos como de las reglas. Esta parte de la práctica se dedica a observar mediante trazas la estrategia que utiliza el intérprete para localizar todas las soluciones.

Veamos primero una breve introducción a esta estrategia. Dada una consulta, el intérprete debe localizar una cláusula (hecho o regla) cuya cabeza unifique con la consulta[4]. Para evitar conflictos entre variables que puedan llamarse igual pero que pertenezcan a distintas cláusulas (o distintas unificaciones con una misma cláusula), cada vez que se unifica con una cláusula el intérprete proporciona una nueva copia de ésta (variante). Todas las variables de una variante son nuevas (están renombradas) y mantienen la misma relación que en la cláusula original.

Si una consulta unifica con un hecho, la solución viene dada simplemente por el resultado de la unificación. En el caso de que la consulta unifique con la cabeza de una regla, el resultado de la unificación se propaga a los predicados en el cuerpo de la regla y se lanzan cada uno de los

[4]Nota que podemos ver los hechos como reglas cuya cola es vacía y cuya cabeza es el propio hecho.

predicados del cuerpo como si fueran consultas. Si todos los predicados del cuerpo tienen éxito, el valor de las variables que ha hecho cierto el cuerpo de la cláusula determina la respuesta.

Debido a que los predicados de las consultas pueden unificar con varias cláusulas del programa, en general la ejecución de un programa lógico requiere realizar una búsqueda. Cada vez que tiene éxito una de estas unificaciones, se está haciendo una elección entre diferentes posibilidades que pueden llevar a una deducción correcta. El retroceso o vuelta atrás se produce cuando se ha realizado una elección que lleva al intérprete a concluir que no puede satisfacer una determinada consulta. Para intentar otras alternativas, se deshacen las instanciaciones de variable asociadas a la última unificación con el fin de probar a unificar con otra cláusula.

En Prolog, el proceso de búsqueda se concreta de la siguiente manera: se van eligiendo cláusulas de arriba a abajo en el orden en el que aparecen en el programa. Cuando se unifica con la cabeza de una regla, los predicados del cuerpo de ésta se van resolviendo (consultando) de izquierda a derecha. Cada predicado resuelto en este orden tiene aplicado el unificador resultante tanto de la unificación con la cabeza como las realizadas para resolver los predicados a su izquierda. Cada vez que el intérprete falla en resolver uno de estos predicados del cuerpo (tras buscar todas las posibilidades de unificación de arriba a abajo), da un paso atrás volviendo al predicado más próximo a la izquierda del que ha fallado. Deshace la última unificación que hizo para este predicado e intenta unificar con otra cláusula del programa que esté más abajo.

Observación de trazas y depuración

De acuerdo con el mecanismo explicado anteriormente, la ejecución de un objetivo puede conllevar las siguientes fases o acciones que se manifiestan en su traza de ejecución:

- Llamada (`Call`): se muestra el nombre del predicado y los valores de sus argumentos en el momento en el que se invocó el (sub)objetivo.
- Terminación con éxito (`Exit`): si el (sub)objetivo se satisface y termina con éxito, entonces se acumula a la respuesta parcialmente computada el valor de los argumentos que hacen satisfacible el (sub)objetivo.
- Terminación con fallo (`Fail`): en este caso, simplemente se muestra una indicación de fallo.
- Reactivación (`Redo`): se informa de una nueva invocación del mismo (sub)-objetivo, producida por el mecanismo de vuelta atrás para intentar resatisfacer el (sub)objetivo.

El intérprete SWI-Prolog proporciona predicados que permiten seguir las búsquedas y retrocesos: `debug`, `nodebug`, `trace` y `notrace`. Estos predicados inician el modo de depuración y permiten observar cómo el intérprete realiza la exploración en la base de conocimiento.

El predicado `trace` es interactivo y permite interaccionar en el proceso (puedes ver las opciones pulsando h durante la traza). Éste es el predicado que vamos a usar para observar y seguir la ejecución del intérprete. Para entrar en modo traza, basta con realizar la consulta `trace.`

Para ilustrar su uso consideraremos el ejemplo que aparece a continuación, en que se lanza la consulta `isRelated(golf,X)` sobre el programa del fichero `cars.pl` sin el predicado `isClassic` ni la cláusula `isRelated` que lo usa. Cada vez que se va a realizar una consulta, en la traza ésta va precedida de la palabra `Call [1]`. Fíjate que aparece la variable `_G2587` procedente de la variante de la cláusula debida al renombramiento automático. Si el intérprete consigue demostrar que la consulta es deducible a partir de las cláusulas del programa, se muestra precedida de la palabra `Exit` y con sus parámetros instanciados a los términos que la hacen cierta.

En [2] la variable _G2587 aparece instanciada a golf. Si el usuario pide que se continúe con la búsqueda (pulsando r tras obtener la primera respuesta en [3]), el intérprete da un paso atrás, deshace la última unificación y retoma la búsqueda de una nueva cláusula a partir de la última con la que unificó.

La continuación de la búsqueda aparece precedida de la palabra Redo. En [4] vuelve a aparecer la variable _G2587 y se busca la siguiente cláusula que unifique con la consulta pero a partir de la siguiente de la última que unificó. En [5] se unifica con la siguiente cláusula, pero sin éxito [6]. Los fallos en la búsqueda aparecen precedidos de la palabra Fail.

Si se insiste en pedir más soluciones a la consulta inicial, el intérprete no encuentra otro hecho isSameBrand con el que unificar y busca otra regla isRelated [7]. Para salir del modo traza usa la consulta notrace. y nodebug. para salir de modo depuración.

```
[trace]  ?- isRelated(touran,X).
   Call: (6) isRelated(touran, _G2587) ? creep          [1]
   Call: (7) isSameBrand(touran, _G2587) ? creep
   Call: (8) model(touran, _G2663) ? creep
   Exit: (8) model(touran, volkswagen) ? creep
   Call: (8) model(_G2587, volkswagen) ? creep
   Exit: (8) model(golf, volkswagen) ? creep
   Call: (8) touran\==golf ? creep
   Exit: (8) touran\==golf ? creep
   Exit: (7) isSameBrand(touran, golf) ? creep          [2]
   Exit: (6) isRelated(touran, golf) ? creep
X = golf ;                                              [3]
   Redo: (8) model(_G2587, volkswagen) ? creep          [4]
   Exit: (8) model(touran, volkswagen) ? creep          [5]
   Call: (8) touran\==touran ? creep                    [6]
   Fail: (8) touran\==touran ? creep
   Fail: (7) isSameBrand(touran, _G2587) ? creep
   Redo: (6) isRelated(touran, _G2587) ? creep          [7]
   Call: (7) isSameYear(touran, _G2587) ? creep
   Call: (8) since(touran, _G2663) ? creep
   Exit: (8) since(touran, 2003) ? creep
   Call: (8) since(_G2587, 2003) ? creep
   Exit: (8) since(touran, 2003) ? creep
   Call: (8) touran\==touran ? creep
   Fail: (8) touran\==touran ? creep
   Fail: (7) isSameYear(touran, _G2587) ? creep
   Redo: (6) isRelated(touran, _G2587) ? creep
   Call: (7) segment(touran, _G2663) ? creep
   Exit: (7) segment(touran, c) ? creep
   Call: (7) segment(_G2587, c) ? creep
   Exit: (7) segment(altea, c) ? creep
   Call: (7) touran\==altea ? creep
   Exit: (7) touran\==altea ? creep
   Exit: (6) isRelated(touran, altea) ? creep
X = altea ;
   Redo: (7) segment(_G2587, c) ? creep
   Exit: (7) segment(golf, c) ? creep
```

```
   Call: (7) touran\==golf ? creep
   Exit: (7) touran\==golf ? creep
   Exit: (6) isRelated(touran, golf) ? creep
X = golf ;
   Redo: (7) segment(_G2587, c) ? creep
   Exit: (7) segment(touran, c) ? creep
   Call: (7) touran\==touran ? creep
   Fail: (7) touran\==touran ? creep
   Redo: (7) segment(_G2587, c) ? creep
   Exit: (7) segment(mégane, c) ? creep
   Call: (7) touran\==mégane ? creep
   Exit: (7) touran\==mégane ? creep
   Exit: (6) isRelated(touran, mégane) ? creep
X = mégane ;
   Redo: (7) segment(_G2587, c) ? creep
   Exit: (7) segment(scénic, c) ? creep
   Call: (7) touran\==scénic ? creep
   Exit: (7) touran\==scénic ? creep
   Exit: (6) isRelated(touran, scénic) ? creep
X = scénic ;
   Redo: (7) segment(_G2587, c) ? creep
   Exit: (7) segment('3008', c) ? creep
   Call: (7) touran\=='3008' ? creep
   Exit: (7) touran\=='3008' ? creep
   Exit: (6) isRelated(touran, '3008') ? creep
X = '3008'.
```

Ejercicio 15 *Comprueba que esté cargada la base de conocimiento* `cars.pl` *(se puede usar* `listing.`*) y en el caso de no estarlo, usa* `consult(cars).` *Ejecuta la traza de la consulta* `isRelated(golf,X).` *excluyendo, y luego incluyendo, las cláusulas que se eliminaron en el ejemplo anterior (*`isClassic`*, y su* `isRelated` *correspondiente).*

Ejercicio 16 *Cerciórate de que esté cargada la base de conocimiento* `flights.pl`*. Ejecuta la traza de la consulta* `connection_same_day(Origin, Destination, Connection).`

Para ejemplos realistas, una traza detallada como la que hemos mostrado en el ejemplo anterior puede carecer de valor práctico, debido a la gran cantidad de información que tiene que procesar el programador. Por este motivo, los sistemas Prolog suministran predicados capaces de producir una traza selectiva: `spy(P)` especifica el nombre de un predicado del que se quiere visualizar una traza (i.e., cuando se lanza un objetivo solamente se muestra información del predicado `P`); `nospy(P)` detiene el "espionaje" del predicado denominado `P`. Puedes consultar el manual para obtener más información sobre éstos y otros comandos para la depuración de programas Prolog.

4. Aritmética simple en Prolog

Técnicamente, la igualdad es en Prolog un predicado como cualquier otro y puede ser escrito en notación prefija (por ejemplo, `=(X,2)`), si bien la notación infija es más estándar y legible.

Ejercicio 17 *Prueba las siguientes consultas y recapacita sobre lo que obtienes:*

```
?- 1 = 2.
?- X = 2.
?- Y = X.
?- X = 8, X = Y.
?- X = Y, X = 8.
```

Ejercicio 18 *Prueba ahora la siguiente consulta:*

```
?- 1 + 2 = 3.
```

El operador ‘‘+’’ *produce un resultado simbólico* ‘‘`1 + 2`’’. *Para hacer un cálculo aritmético, necesitas forzarlo. Esto se hace con el operador* `is`. *Prueba estas consultas:*

```
?- 2 is 1 + 1.
?- X is 3 * 4.
?- Y is Z + 1.
```

Observa el mensaje de error del último ejemplo: Prolog requiere que el lado derecho de la relación `is` *sea básico (*ground*), es decir, no puede contener variables. ¡De lo contrario Prolog podría tener que resolver algunas ecuaciones no triviales!*

Ejercicio 19 *Escribir un predicado factorial para poder responder con éxito a la consulta* `?- factorial(6,Y).` *¿Sirve este mismo predicado resolver la consulta* `?- factorial(X,24).`*?*

Ejercicio 20 *Considera el siguiente programa Prolog, en el que se usa la llamada* `put_code(32)`*, cuyo efecto es escribir un espacio en blanco en pantalla,*

```
tab(0).
tab(N) :- put_code(32), N is N - 1, tab(N).
```

y el objetivo: `?- tab(8).` *Selecciona la respuesta correcta de entre las siguientes.*

1. *se produce un error de ejecución porque el argumento de* `put` *ha de ser un carácter.*
2. *se produce un error de ejecución porque el primer argumento del predicado* `is` *ha de ser una variable no instanciada.*
3. *la ejecución tiene como efecto una tabulación de ocho espacios.*
4. *el objetivo falla.*

Ejercicio 21 *¿Cómo puedes conseguir que el programa anterior funcione como se esperaba? Sustituye el predicado* `put_code(32)` *por* `put_char('.')` *para visualizar mejor el resultado en pantalla.*

Apéndice

Clasificación de los términos de Prolog

- Simples
  - Variables: se representan mediante cadenas de caracteres formadas por letras, dígitos y el símbolo de subrayado, empezando necesariamente por una letra mayúscula o por un símbolo de subrayado (por ejemplo: `X`, `Result_1`, `_total3`). Las que empiezan con un símbolo de subrayado son variables anónimas y se usan cuando se necesita trabajar con variables cuyos posibles valores no interesan porque no van a ser utilizados.
  - Constantes
    - Constantes simbólicas: cadenas formadas por letras, dígitos y el símbolo de subrayado, que deben empezar necesariamente por una letra minúscula. Por ejemplo: `f`, `barcelona`, `book33a`, `white_book`. **Nota**: A estas constantes también se las conoce como átomos, aunque no se debe confundir esta terminología con las fórmulas atómicas, que son predicados, como por ejemplo: `model(clio,renault)`.
    - Números: se utilizan para representar tanto números enteros como reales.
      - Enteros: se representan con la notación habitual (0, 1, -320, 1520, etc.).
      - Reales: se pueden representar tanto en notación decimal como en notación científica. En ambos casos deberá haber siempre por lo menos un dígito a cada lado del punto.
- Compuestos: se construyen mediante un símbolo denominado functor (denotado por una constante simbólica) y seguido, entre paréntesis, por una serie de términos separados por comas (argumentos). Por ejemplo: `date(10,nov,2030)`, `time(13,05)`. **Nota**: al escribir un término compuesto no puede haber ningún espacio entre el functor y el paréntesis abierto previo a los argumentos.