Minimanualillo de SWI-Prolog

Roberto Torres de Alba

22 de marzo de 2006

Índice

1. Introducción

Este pequeño manual pretende ser una introducción al uso de SWI-Prolog poniéndolo en relación con los conceptos de la asignatura de Programación Lógica de las Ingenierías Técnicas de Gestión y de Sistema. La versión sobre la que versa este manual es la última que se puede encontrar, a día de hoy, en la página *www.swi-prolog.com*, que es la 5.6.6 para Windows NT/2000/XP.

2. Comandos básicos

Una vez instalado SWI-Prolog y procediendo a su ejecución observamos el *shell* visto como un número más los caracteres "?-". Desde ahí es de donde vamos a ajecutar todos los objetivos. Los siguientes apartados explican de forma muy general todo aquello que es fundamental conocer a la hora de usar este intérprete.

1. Consultar un programa Prolog: la manera más fácil de consultar un programa Prolog es usando el menú *file/consult* y navegando por los directorios. También existe la posibilidad de hacerlo desde el *shell* escribiendo el nombre del programa, sin la extensión ".pl", encerrada por corchetes "[ ]" y seguida de punto. Ej:

   ```
   [mi_prog].
   ```

2. Ayuda: SWI-Prolog posee una ayuda gráfica fácil de usar. Basta con poner en el *shell* "help.". Si se quiere consultar un tema o un predicado en concreto basta con escribir el tema de la siguiente forma: help(Tema).

3. Ejecución de objetivos: para ejecutar un objetivo simplemente lo escribimos en el *shell* (seguido de punto). Si el objetivo no tiene éxito SWI-Prolog responderá "no". Si ha tenido éxito y el objetivo tenía variables entonces devolverá la unificación de esas variables que ha producido el éxito y el programa esperará una acción del usuario. Ahora debemos escribir un único carácter. Si buscamos más respuestas escribiremos ";" (punto y coma), si no pulsamos *enter*. Para una lista de comandos escribimos "h".

4. Depuración: Para entrar en el modo *trace* hay que escribir "trace.". Al lado del *shell* debería aparecer la palabra "[trace]". Para salir de este modo hay que escribir "notrace." y "nodebug.". Existe la posibilidad de utilizar un depurador gráfico escribiendo "guitracer."

3. Sintaxis de Prolog

Cualquier programa en Prolog tiene que estar escrito en un fichero de texto plano (sin formato). La manera más sencilla es usar el Bloc de Notas. Dicho archivo debe poseer la extensin ".pl" para indicar que contiene código fuente de Prolog.

Un programa Prolog está formado con un conjunto de hechos y de reglas junto con un objetivo. El archivo del código fuente de Prolog contendrá el conjunto de hechos y de reglas y el objetivo será lanzado desde el *shell* de SWI-Prolog.

Existen ciertas consideraciones sobre la sintaxis de Prolog:

1. Variables:
   El identificador de una variable tendrá que tener su primera letra en mayúsculas.
   Ej: X, Atapuerca, Cobaltina, RADgtfCdf

2. Constantes:
   La primera letra de una constante deberá estar escrita en minúsculas.
   Ej: a, incienso, roberto, rADgtfCdf
   También se consideran constantes a los números,
   Ej: `1, 5.02, 0.7`
   las palabras entre comillas simples
   Ej: `'a', 'A', 'a a'`
   y la lista vacía [ ].

3. Funciones:
   Al igual que las constantes, su primera letra debe ser una minúscula. Deberá estar seguido de un conjunto de términos (otras funciones, variables o constantes) encerrados entre paréntesis.
   Ej: f(c,X), conc_arbol(Hijo_Izq, Raiz, Hijo_Der), rADgtfCdf(RADgtfCdf, rADgtfCdf)

4. Predicados:
   Su sintaxis es la misma que la de las funciones aunque, por su localización dentro de la cláusula (es decir, dentro del programa Prolog), el compilador los identificará como tal, y no como funciones.
   Ej: f(c,X), conc_arbol(Hijo_Izq, Raiz, Hijo_Der), rADgtfCdf(RADgtfCdf, rADgtfCdf)
   También existe la posibilidad de tener predicados 0-arios.

5. Hechos:
   Son cláusulas de Horn que poseen un único predicado en la cabeza y ninguno en el cuerpo. Tienen la siguiente forma en sintaxis de lógica de primer orden:

   $$P \leftarrow$$

   En Prolog no se escribe la flecha sino que se pone un punto al final:

   $$p.$$

   donde p es un predicado y tiene que seguir su sintaxis. Ej:

   ```
   padre(aaron, maria).
   compositor(shostakovich).
   shostakovich(compositor).
   ```

6. Reglas:
   Son cláusulas de Horn que poseen un único predicado en la cabeza y uno o más en el cuerpo. Tienen la siguiente pinta:
   $P \leftarrow Q1, Q2, Q3$ escritos en sintaxis clausular o
   $P \leftarrow Q1 \wedge Q2 \wedge Q3$ escritos en sintaxis de lógica de primer orden.
   En Prolog la flecha se sustituye por ":-", las conectivas conjuntivas se escriben como comas "," y la regla termina en punto:

   ```
   p :- q1, q2, q3.
   ```

   donde, al igual que los hechos, p y $q1$, $q2$ y $q3$ son predicados. Ej:

   ```
   cuadrado(X) :- poligono(X), numero_lados(X,4).
   ```

7. Objetivos:
   Son cláusulas de Horn que no poseen ningún predicado en su cabeza:
   $\phi \leftarrow Q1, Q2, Q3$
   Los predicados se escriben separados por comas y terminados en punto. Sólo pueden sen lanzados desde el shell de SWI-Prolog.

```
?- padre(X, Y),padre(Y, Z).
```

4. Ayuda

En la ayuda de SWI-Prolog, la definición de los predicados tiene su propia sintaxis. El nombre y la aridad determinan unívocamente a un predicado. Esto en Prolog se escribe *pred/arid* donde *pred* es el nombre del predicado y *arid* su aridad. Después del nombre del predicado se escribe un número de variables, dependiendo de la aridad, con un símbolo delante de la variable:

+ : significa que esa variable debe estar instanciada, es decir, que no puede ser una variable sin estar unificada con nada, en el momento es que se llega a ese predicado. Desde el punto de vista del programador se puede ver como un parámetro de entrada. A pesar de que en lógica de primer orden ello no tendría mucho sentido, en SWI-Prolog se dan cosas así debido a que, por ejemplo, dicho predicado puede estar relacionado con una llamada a C (como *sort/2*) o con operaciones aritméticas (como *is/2*). Ej:

```
?- sort(X,[1,2,3,4]).
ERROR: sort/2: Arguments are not sufficiently instantiated
```

- : identico a + pero esta vez como predicado de salida. Pongamos lo que le pongamos va a intentar unificar con ello. Ej:

```
?- sotr([5,4,3,2,1], L).

L = [1, 2, 3, 4, 5]

?- sort([5,4,3,2,1], [C|R] ).

C = 1
R = [2, 3, 4, 5]

?- sort([5,4,3,2,1], [2,1,3,5,4] ).

No
```

?: se considera de entrada o de salida. Esto sigue el procedimiento normal de resolución de la programación lógica.

Para ver la ayuda sobre el predicado *pred* escribimos *help(pred)*. Si además nos referimos a un predicado concreto también tenemos que escribir su aridad: *help(pred/2)*. Podemos indicar una sección del manual: la seccion 2.4 se escribira como 2-4: *help(2-4)*.

Si no tenemos información tan concreta podemos usar el predicado *apropos/1*. A este predicado se le pasa el nombre aproximado de lo que buscamos. Nos devolvera predicados, funciones o secciones del manual que contengan ese nombre. Sí le importa las mayúsculas o minúsculas (cuidado si la primera es mayusculas porque, lógicamente, pensará que el argumento es una variable). También podemos proporcionarle frases escritas entre comillas simples. Ej:

```
apropos(file).
apropos('file').
apropos('File').
apropos('the file').
```

Los predicados *explain/1* y *explain/2* dan cierta información sobre el argumento, como su función o referencias aunque es demasiado escueto.

Esta información está recogida en la sección 2.6 del manual.

5. Consultando y modificando programas

5.1. Consultar programas

Cargar y compilar un programa en Prolog se conoce como consultar. Existe varias formas. La más fácil consiste en usar el menú *File→Consult*, aunque esta opción sólo está disponible en el entorno Windows.

También podemos usar el predicado *consult/1*. *consult/1* puede ser invocado de varias formas. La más fácil consiste en llamar a *consult/1* con el nombre del programa sin la extensión. Es indiferente el uso de mayúsculas o minúsculas excepto para la primera letra, que debe ser minúscula para que Prolog no piense que es una variable. Ej:

```
consult(practica3_chunga).
```

En este caso buscará en el directorio actual y en una serie de directorios que tiene para buscar.

Otra forma consiste en escribir la ruta del archivo completa entrecomillada con comillas simples. La barra que separa los directorios en Windows debe estar inclinada hacia la derecha (y no como en Windows que está escrita hacia la izquierda). Ej:

```
consult('c:/hlocal/practica3_suspensa').
consult('c:/hlocal/practica3_suspensa.pl').
```

Otra alternativa a este predicado es el uso de los corchetes. Su funcionamiento es identico al de *consult/1* pero poniendo el nombre del archivo entre corchetes. Ej:

```
consult(practica3_chunga).
[practica3_chunga].
['practica3_chunga'].
['practica3_chunga.pl'].
['c:/hlocal/practica3_chunga.pl'].
```

Para consultar en qué directorio estamos usamos el predicado *pwd/0*. Para cambiar la ruta usamos el predicado *cd/1*. Se usa escribiendo entre comillas simples la ruta absoluta o relatica. Ej:

```
1 ?- pwd.
c:/hlocal

yes
2 ?- cd('..').

yes
3 ?- pwd.
c:

yes
```

Existe más información detallada en la sección 4.3 del manual y, una más útil en la 4.3.2.

5.2. Errores y avisos

Cuando se consulta un archivo pueden aparecer mensajes de error o de aviso. Los más comunes son:

1. Error sintáctico: el intérprete mostrará el error que se ha producido, donde ha sido y por qué se ha parado. Ya que la sintaxis de Prolog es sencilla suele acertar la mayoría de las veces.
2. Error de archivo no encontrado: hay que fijarse en el nombre del archivo, la ruta que se ha puesto explícitamente y los directorios por defecto en donde busca, como puede ser el directorio de trabajo.
3. Aviso de variable *singleton*: una variable *singleton* es una variable que solamente aparece una vez en una cláusula. El compilador interpreta que puede haber un error al escribir esa variable pero, por no tratarse de un error, da un aviso. Si la cláusula está bien hecha y posee una variable singleton significa que el valor de esa variable no importa en dicha cláusula ya que va a unificar con cualquier cosa y no tiene relevancia posterior. Para que el programa sea más claro y no salga el aviso podemos usar, en su lugar, una variable anónima. Las variables anónimas se escriben como un subrayado "_", unifican con cualquier cosa pero no entre ellas. Ej:

   ```
   % multiplica(in, in, out)
   multiplica(0, _, 0).
   ```

4. Aviso de que las cláusulas de un mismo predicado no están juntas: este aviso salta cuando se escriben cláusulas de un predicado entre cláusulas de otro.

5.3. Mostrar base de clausulas

Para ver el contenido de la base de cláusulas usamos el predicado *listing/0*. Este predicado muestra todas las cláusulas que tenemos actualmente. Para ver sólo aquellas cláusulas que pertenecen a un predicado usamos *listing(+Pred)* donde *Pred* es el predicado que buscamos mirar. También podemos escribir *listing(+Pred/Arid)* donde *Arid* es la aridad del predicado *Pred*.

5.4. Modificando la base de clausulas

Podemos añadir y eliminar cláusulas usando la familia de predicados *assert* y *retract*. Con *asserta/1* insertamos la cláusula al principio de la lista de cláusulas de ese predicado. Con *assert/1* y *assertz/1* hacemos lo mismo pero al final. Con *retract/1* y *retractall/1* las eliminamos.

6. Depurador

6.1. Depurador en modo texto

El modo depurador comienza cuando escribimos el predicado *debug/0*. Deber aparecer "[debug]" antes de la interrogación en el shell de Prolog. Para salir de ese modo basta con usar *nodebug/0*. Para conocer el estado del depurador usamos *debugging/0*.

Podemos establecer puntos de paradas para el depurador, que en Prolog se conocen como puntos espías. Dichos puntos son establecidos con el predicado *spy/1*. Con *spy(+Pred)* establecemos un punto de espionaje en las cláusulas cuya cabeza sea *Pred*. Ej:

```
dios_hindu(ganesha).
dios_hindu(krishna).
dios_hindu(shiva).
?- spy(dios_hindu).
```

Para eliminar un punto espía usamos *nospy/1* o para eliminar todos usamos *nospyall/0*. Una vez que se ha alcanzado un predicado que lleva asociado un espía la ejecucin del programa se para en los llamados puertos (ports). Existen 6 y son:

- Call: pasamos por él cuando se llama a la cláusula.
- Unify: cuando se ha unificado la cabeza de la cláusula con el objetivo actual.
- Exit: cuando salimos del objetivo.
- Redo: cuando, por recursión, probamos otra cláusula de un mismo predicado.
- Fail: cuando esa cláusula falla.
- Exception: cuando el predicado *throw/1* lanza esa excepción.

En el *shell* de SWI-Prolog veremos el nombre del puerto seguido del objetivo con la unificación que lleve hasta ese momento más una interrogación. Eso indica que se espera un gesto del usuario y será la escritura de un único carácter sin necesidad de seguirlo por *enter*. Pulsando "h" obtendremos la lista de comandos. Los más útiles son:

- Enter (o espacio o "c"): continua hasta el siguiente puerto.
- h: muestra las opciones disponibles.
- a: aborta la ejecución.
- L: muestra la cláusula.
- +: establece un punto de espionaje.
- -: elimina un punto de espionaje.

El predicado *trace/0* hace lo mismo que el predicado *debug* pero sin necesidad de establecer un punto espía para pararse en un puerto, sino que lo realiza desde el principio de la ejecución. Cuando llamamos a *trace/0* aparece "[trace]" a la izquierda de ?- indicando que hemos pasado a ese modo. Para salir de él simplemente llamamos a *notrace/0*.

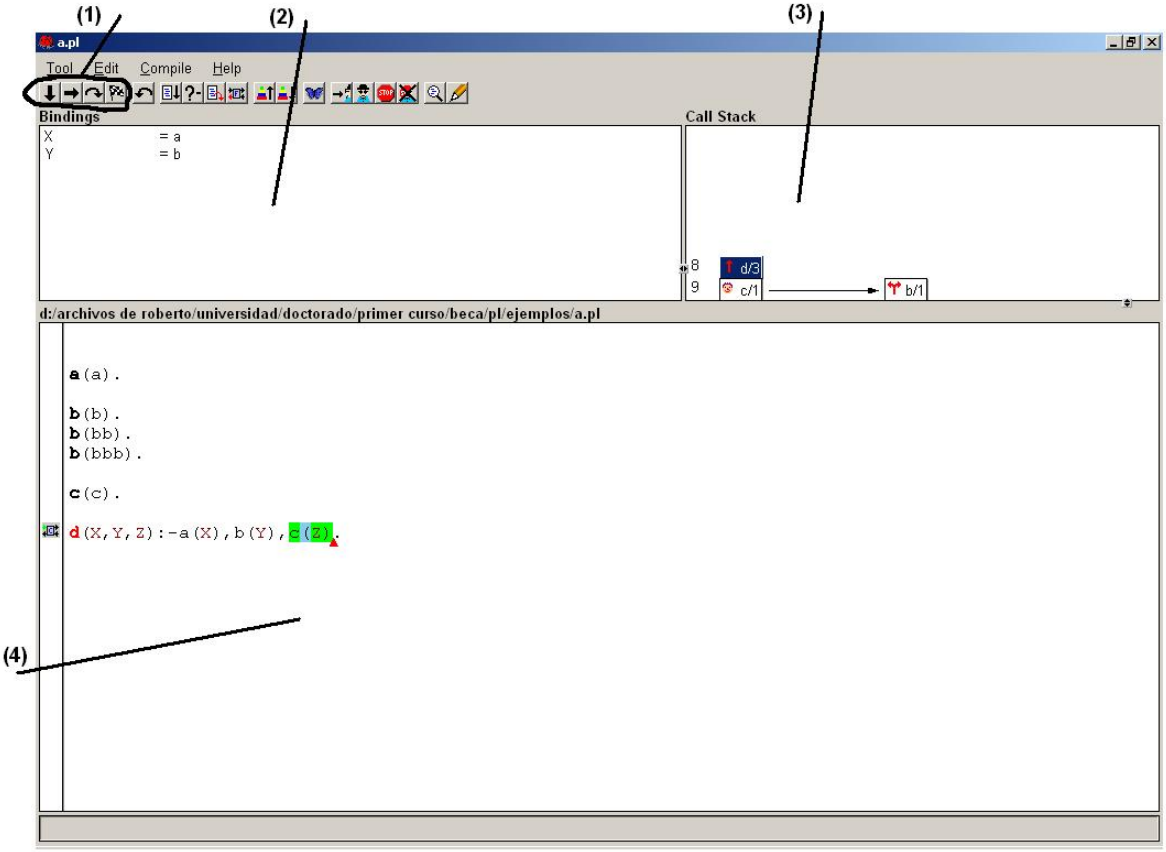
Existe muchas secciones del manual que hablan sobre el depurador. Yo recomiendo las secciónes 2.9 y 4.38 por proporcionar una visión general.

6.2. Depurador gráfico

Existe también un depurador gráfico. Para que funcione se debe escribir *guitracer.* y cuando queramos dejar de usarlo escribimos *noguitracer.*.

Funciona de la misma manera que el depurador en modo texto excepto que, cuando la ejecución se para en algún punto, aparece una ventana.

La ventana que aparece es esta:



(1) Estos botones nos sirven para avanzar en la ejecución del programa. En concreto la flecha recta hacia la derecha se usa para avanzar paso a paso.

(2) Esta parte muestra las unificaciones que llevamos en este momento en esta cláusula. En este ejemplo, hemos unificado X con a e Y con b. Todavía falta la Z de $c(Z)$ porque es el predicado que se va a llamar ahora.

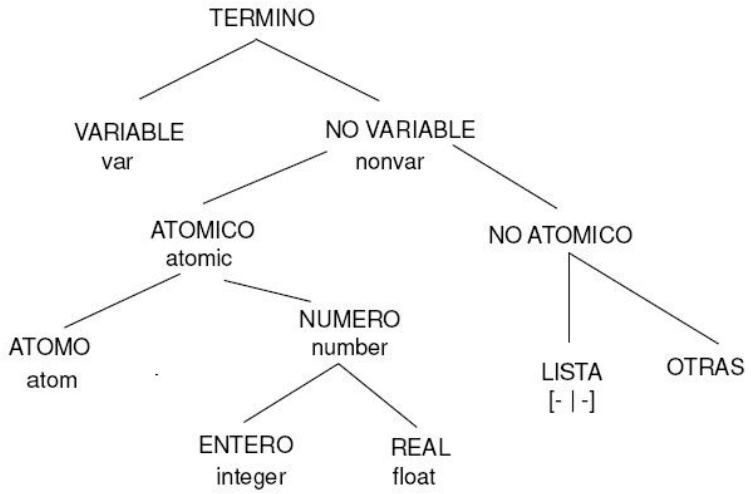
(3) Aquí se muestra la pila de objetivos que sirve para la recursión. En este ejemplo se ve que estamos ejecutando el predicado *d/3* y que ahora vamos a llamar a *c/1*. También vemos que queda pendiente hechos del predicado *b/1*, en concreto *b(bb).* y *b(bbb).*

(4) En esta ventana aparece el programa Prolog. Se indica que cláusula se esta ejecutando ahora y en color verde a qué objetivo se va a llamar. En color rojo aparecen las clausulas que fallan.

7. Manejo de términos

7.1. Jerarquía

Para facilitar la tarea de la programación en SWI-Prolog existe una pequeña jerarquía de términos



Debajo de cada categoria de término se ha escrito el predicado metalógico que lo define. Ej:

```
?- var(X).

X = _G230

Yes

?- X=a, var(X).

No
```

Todos los predicados pueden ser encontrados en la sección 4.5 del manual.

7.1.1. Átomos

Los términos átomos son aquellos que se corresponden con las constantes del vocabulario en lógica de primer orden. Se comprueban que están bien definidos mediando el predicado *atom/1*.

Existen dos formas de definir constantes. La primera es escribiendo el nombre de la constante con la primera letra en minúsculas. Ej:

```
?- atom(a).

Yes

?- atom(fEdErIc0).

Yes
```

La segunda forma es más flexible. Si buscamos que nuestra constante contenga espacios, caracteres raros o la primera letra en mayúsculas lo escribimos entre comillas simples. Ej:

```
?- atom('a').
```

```
Yes

?- atom('a a').

Yes

?- atom('A').

Yes

?- atom('2').

Yes
```

7.1.2. Números

Es difícil la definición de predicados para números en el que sus argumentos sean de entrada y de salida, tal y como se hace en los predicados en programación lógica. Por eso los argumentos de los predicados aritméticos sólo poseen una dirección en SWI-Prolog.

El más importante de ellos es el predicado *is/2* que se usa de forma infija. En su argumento izquierdo usamos una variable libre (u otro valor con el que queremos que unifique) y en el argumento derecho una expresión aritmética. Ej:

```
?- N is 2 + 2.

N = 4

?- N is 5 / 7.

N = 0.714286

?- 4 is 2 + 1.

No

?- N is X + 2.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Otros predicados de este tipo son las relaciones `<`, `>`, `=<`, `>=`, `=\=` y `=:=` que se usan también en forma infija.

Si se busca más información sobre cómo maneja SWI-Prolog los números hay que consultar la sección 4.26 del manual.

7.2. Escritura y lectura de términos

Prolog posee predicados para leer o escribir términos por pantalla o desde un archivo. El predicado más general para la escritura de terminos es *write_term/2*. Otros predicados más sencillos son *write/1*, *writeln/1* o *writeq/1*. Ej:

```
?- write(f(a)).
f(a)

Yes
?- write('a a').
a a

Yes
?- write(f(a,X)).
f(a, _G279)

X = _G279

Yes
```

También ayudan a formatear la salida predicados como *nl/0* (que escribe un salto de línea) y *tab/1* (que escribe una cantidad de espacios).

Existen versiones de estos predicados con un argumento más añadido que corresponde con el "stream" de un archivo. Su función es la misma pero lo hacen en el archivo de texto asociado con el stream.

Para la lectura de términos existen predicados análogos: *read_term/2* y *read/1* junto con predicados que llevan como argumento un flujo de un fichero.

Hay algo importante que decir sobre la lectura de términos. Para que Prolog sepa cual es el final de un término se debe acabar con un punto. Hay que tener cuidado con la escritura de un término en un archivo si queremos volverlo a leer inmediatamente. Hay que escribir el punto explícitamente (*write('.')*) porque *write/1* no lo hacer. Ej:

```
?- read(T).
|    f(a).

T = f(a)

Yes
?- read(f(T)).
|    f(a).

T = a

Yes
?- read(g(T)).
|    f(a).

No
```

Más predicados pueden ser encontrados en la sección 4.19.

8. Listas

Existe un módulo que se carga por defecto para el manejo de listas y es el que vamos a tratar ahora. Para manejar listas SWI-Prolog proporciona los dos términos constructores de la lista: [ ], la constante lista vaca; y $[X|R]$ el operador (función infija) concatenar por la cabeza, donde R debe ser una lista a su vez. Para una definición más formal de lista consultar el predicado *is_list/1*. También se proporciona predicados para el manejo de listas. La lista completa está en la sección 4.28 y en el apéndice A.1 (11.1 si usamos el comando help en vez del .pdf) del manual. Nosotros vamos a ver algunos por encima:

***is_list(+Term)*:** cierto si *Term* es una lista.

***length(?List, ?Int)*:** *Int* es el número de elementos de la lista *List*.

***sort(+List, -Sorted)*:** *Sorted* es la lista ordenada de los elementos de *List* sin duplicados.

***append(?List1, ?List2, ?List3)*:** *List3* es la concatenación de *List1* y *List2*

***member(?Elem, ?List)*:** *Elem* es elemento de *List*.

***nextto(?X, ?Y, ?List)*:** *Y* está después de *X* en la lista *List*.

***delete(+List1, ?Elem, ?List2)*:** *List2* es la eliminación de todos los elementos que unifican simultáneamente con *Elem* de *List1*.

***nth0(?Index, ?List, ?Elem)*:** *Elem* es el *Index*-ésimo elemento de *List*, comenzando por el 0.

***reverse(+List1, -List2)*:** *List2* es *List1* pero con el orden de los elementos cambiado.

9. Conjuntos

SWI-Prolog también posee un módulo para el manejo de conjuntos. Un conjunto está implementado como una lista sin repeticiones aunque no es necesario conocer dicha implementación ya que podemos usarlo como un Tipo Abstracto de Datos con el predicado *list_to_set/2*. Podemos consultar los predicados relativos a conjuntos en el apéndice A.1.1 (u 11.1.1):

***is_set(+Set)*:** *Set* es un conjunto

***list_to_set(+List, -Set)*:** *Set* es el conjunto de elementos de *List* pero sin duplicados.

***intersection(+Set1, +Set2, -Set3)*:** *Set3* es la intersección de *Set1* con *Set2*.

***subtract(+Set, +Delete, -Result)*:** elimina de *Set* todos los elementos del conjunto *Delete*.

***union(+Set1, +Set2, -Set3)*:** *Set3* es la unión de *Set1* y *Set2*.

***subset(+Subset, +Set)*:** todos los elementos de *Sebset* están en *Set*.

10. Manejo de archivos

Para abrir un fichero tenemos el predicado *open/3* (también existe *open/4* que es igual que *open/3* pero con una lista de opciones adicionales). El uso de *open/3* es muy parecido al de C ya que se adapta a un estándar de ISO. A *open/3* se le pasa el nombre y/o la ruta del archivo escrito entre comillas simples y el modo de escritura. Devuelve en una variable el flujo (*stream*) donde podremos escribir. El modo debe ser una de estas constantes: *read*, *write*, *append* y *update*. *append* abre el fichero para escritura y se posiciona al final y *update* hace lo mismo pero se posiciona al principio (sin borrar nada a priori), por lo que cada vez que se escriba sustituirá algo de texto Ej:

```
open('declaracion_amor.txt', write, S).
open('c:/novias_secretas/declaracion_amor.txt', write, S).
open(declaracion_amor, read, S).
```

Para cerrar el flujo asociado con el fichero usamos *close/1*. Para leer y escribir tenemos la familia de los predicados *read/2* y *write/2*, como se ha visto anteriormente. El primer argumento será el flujo y el segundo el término (aunque mandemos texto al archivo, éste debe estar escrito como un término).

Más información en la sección 4.16 aunque nos hemos centrado en la 4.16.1.

11. Todas las soluciones a un objetivo

Existen predicados que no se paran cada vez que llegan a una hoja éxito en el árbol de vuelta atrás sino que realizan todo el árbol que está por debajo de un objetivo dado y guardan las soluciones (unificaciones) en una lista.

findall(+Template, +Goal, -Bag): realiza todo el árbol de *backtracking* del objetivo *Goal*. Cada vez que llega a un éxito unifica las variables con la plantilla *Template* e introduce el resultado en la lista *Bag*.

```
5 ?- listing.

dios_egipcio(amon).
dios_egipcio(anubis).
dios_egipcio(apis).
dios_egipcio(ra).

Yes
6 ?- findall(X,dios_egipcio(X), L).

X = _G362
L = [amon, anubis, apis, ra]

yes
```

bagof(+Template, +Goal, -Bag): idéntico a *findall/3* excepto que si el objetivo posee variables libres entonces primero probará la unificación con esas variables libres y después realizará el árbol de vuelta atras para cada una de esas unificaciones, parando cada vez.

```
?- listing.

vida(carlos_III, 1716, 1788).
vida(carlos_IV, 1748, 1819).
vida(fernando_VII, 1784, 1833).

yes
?- bagof([A,B], vida(A, B, C),L).

A = _G371
B = _G374
```

```
C = 1788
L = [[carlos_III, 1716]] ;

A = _G371
B = _G374
C = 1819
L = [[carlos_IV, 1748]] ;

A = _G371
B = _G374
C = 1833
L = [[fernando_VII, 1784]] ;

No
```

Si no queremos que se pare cada vez que encuentre una unificación debemos usar el símbolo ^ como en el ejemplo:

```
?- bagof([A,B], C^vida(A, B, C),L).

A = _G383
B = _G386
C = _G391
L = [[carlos_III, 1716], [carlos_IV, 1748], [fernando_VII, 1784]] ;

No
```

setof(+Template, +Goal, -Set): igual que *bagof/3* pero la lista *Bag* es devuelta ordenada y sin duplicados.

En las secciones 4.31 y 4.29 existe explicaciones de estos predicados.