Conferencia 6: Programación Declarativa
M.Sc Dafne García de Armas
Curso 2018-2019

Tema: Meta programación

En esta conferencia se mostrarán algunas de las características avanzadas del lenguaje Prolog como los **predicados predefinidos para la clasificación y construcción de términos**, las facilidades que brinda para la **metaprogramación**, su relación con las bases de datos, los predicados predefinidos para la extracción de un conjunto de respuestas así como el trabajo modular en Prolog.

1. Predicados predefinidos para la clasificación y construcción de términos

Prolog es un lenguaje relacional, es decir, es un lenguaje basado en el concepto de relación o predicado de la Lógica de Primer Orden. Como se ha visto, a través de ejemplos de programas, los argumentos de los predicados pueden ser ocupados por términos descriptivos, es decir, términos construidos a partir de símbolos de funciones que son constructores n-arios que sirven para definir un valor en Prolog.

Básicamente, un término puede ser una variable, un átomo, una constante numérica y un término compuesto. Una variable es un valor que no ha sido unificado aún (e.g X). Un átomo es una constante textual utilizada para nombrar los términos compuestos, para representar una constante o un texto (e.g 'ana'). Las constantes numéricas se utilizan en su notación habitual para representar números enteros o reales (e.g. -3, 4.5). Un término compuesto, también denominado estructura, consiste de un nombre seguido por n argumentos (la aridad del término), cada uno de los cuales es un término (e.g. persona ('Ana', 28,' doctora')).

Aunque Prolog es un lenguaje sin tipos (todos los datos pertenecen a un mismo dominio sintáctico, el universo de Herbrand) es posible reconocer la "categoría" a la que pertenece un cierto dato (valor) analizando su forma sintáctica. Como se verá a continuación, Prolog cuenta con una colección de predicados que permiten tanto clasificar como crear términos.

Clasificación de términos

- **var(Term):** tiene éxito si Term es una variable no unificada (libre).
- **nonvar(Term):** tiene éxito si Term es una variable unificada (ligada).
- **atom(Term):** tiene éxito si Term se ha unificado a una constante (átomo).
- **integer (Term):** tiene éxito si Term se ha unificado a un número entero.
- **float (Term):** tiene éxito si Term se ha unificado a un número real.
- **number(Term):** tiene éxito si Term se ha unificado a un número entero o real.
- **atomic(Term):** tiene éxito si Term se ha unificado a un objeto simple (atómico), i.e. un símbolo constante, un número o una cadena.
- **compound(Term):** tiene éxito si Term se ha unificado a un término compuesto.
- **ground(Term):** tiene éxito si Term no contiene variables sin unificar, o sea, variables libres.
- **callable(Term):** tiene éxito si Term se ha unificado a un átomo o un término compuesto, luego puede ser tomado como argumento por los predicados call/1, functor/3 y =../2.

Análisis y construcción de términos. Meta predicados

Para empezar a construir meta intérpretes con mayor granularidad necesitamos, además de meta variables, predicados que son meta predicados, es decir, predicados que toman como argumentos cláusulas que definen predicados en el programa objeto, ya sean estos predicados predefinidos del

intérprete o predicados definidos en el programa objeto cuyas cláusulas se hallan en la Base de Datos o de Conocimiento (BC) que mantiene el intérprete.

A continuación se introducen algunos de los meta predicados más importantes que se aparecen predefinidos en el intérprete de Prolog. Para obtener mayores especificaciones sobre el empleo de estas metas predicados y sus variantes deben ser consultados la ayuda del intérprete que se está utilizando en las clases prácticas, el SWI Prolog.

call(Obj): invoca el objetivo Obj y tiene éxito si Obj tiene éxito. Se utiliza para lanzar objetivos dentro de una regla o de un programa que solo se conocen en tiempo de ejecución. Si Obj es una variable deberá estar unificada en el momento en que vaya a ejecutarse. Ejemplo: la definición anterior del predicado not/1 es equivalente a esta:

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

clause/2: clause(Cabeza, Cuerpo)
Predicado que permite tener acceso a las cláusulas en la BD que mantiene el intérprete. Interpretación: clause/2 busca en la BD una cláusula cuya cabeza unifique con Cabeza (Cabeza no puede ser nunca una variable) y suministra en su segundo argumento Cuerpo el cuerpo de la cláusula con la que logró la unificación al cual le ha aplicado el umg resultado de la unificación (Cuerpo puede ser una variable). Si se unifica con un hecho, la variable Cuerpo se instancia con el átomo true.

```
?- clause(padre(luis,jose),B).
B = true;
False

?- clause(abuelo(X,Y),B).
B = (padre(X, _G1020), (madre(_G1020, Y);padre(_G1020, Y))).
```

functor/3: functor(Término, Functor, Aridad)
Tiene éxito si T es un término con functor principal F y aridad A. Si T es una variable pero F y A no lo son, entonces T unificará con un nuevo término con functor F y contendrá A variables como argumentos. Si T es un átomo o un número, entonces F unificará con T y A con el número 0. El predicado functor/3 fallara si T y (F o A) no están unificadas. Ejemplos:

```
?- functor(padre(jose,luis),F,A).
F = padre
A = 2

?- functor(ana,F,A).
F = ana
A = 0

?- functor(T,padre,3).
T = padre(_G431,_G432,_G433)

?- functor(T,F,2).
ERROR: functor/3: Arguments are not sufficiently instantiated
```

arg /3: arg (N, T, V)
Tiene éxito si el valor V unifica con el N-ésimo argumento del término T. Si T no es una variable pero N y V si lo son, entonces V unifica con cada uno de los argumentos de T. Ejemplos:

```
?- arg(1,padre(luis,jose),luis).
true.

?- arg(1,padre(luis,jose),V).
V = Luis

?- arg(2,padre(luis,X),jose).
X = jose

?- arg(A,padre(luis,jose),V).
A = 1
V = luis ;
A = 2
V = jose
```

T =.. L: Tiene éxito si L unifica con la lista [X|Y], donde X es el functor principal del término T y el resto Y contiene los argumentos de T. Tanto T como L pueden ser variables pero no al mismo tiempo.

Ejemplos:

```
?- padre(luis,jose) =.. L.
L = [padre,luis,jose]


?- T =.. [member,X,[1,2,3]].
T = member(X,[1,2,3])
```

El siguiente ejemplo ilustra las facilidades que brindan algunos de estos predicados para la manipulación simbólica.

En el siguiente ejemplo se utilizan los predicados call/1 y =../2 para definir el predicado map/3 que, dado el nombre de una función F y una lista L, devuelve la lista que contiene el resultado de aplicar a todos los elementos de L la función F.

```
map(_,[],[]).
map(F,[X|Y],[V|R]):- T =..[F,X,V], call(T), map(F,Y,R).
```

Ahora, considerando las siguientes funciones:

```
cuadrado(X,Y):- Y is X**2.
cubo(X,Y):- Y is X**3.

incrementa(X,Y):- Y is X+1.

decrementa(X,Y):- Y is X-1.
```

Es posible usar el mismo predicado para obtener diferentes procesadores de listas:

```
?- map(cuadrado,[1,2,3,4],L1), map(incrementa,L1,L2).
L1 = [1,4,9,16],
L2 = [2,5,10,17] ;
false

?- map(cubo,[1,2,3,4],L1), map(decrementa,L1,L2).
L1 = [1,8,27,64],
L2 = [0,7,26,63] ;
false
```

Ejemplo: Definición del predicado cumplen_todos(L,C) que, dada una lista L y un predicado unario C, determina si cada elemento de L cumple la condición C.

```
cumplen_todos([],_).
cumplen_todos([X|Y],C):- T=..[C,X], T, cumplen_todos(Y,C).
```

La definición anterior puede utilizarse para comprobar si una lista contiene solo variables, átomos, números, etc. mediante los predicados predefinidos de catalogación de términos:

```
?- cumplen_todos([X,Y],var).
True

?- cumplen_todos([madre,padre],atom).
True

?- cumplen_todos([1,2,3,4.5],number).
True

?- cumplen_todos([1,2,3,4.5],integer).
False
```

También con cualquier otro predicado unario definido por uno mismo, por ejemplo, los predicados par/1 e impar/1:

```
par(X):- R is X mod 2, R =:= 0.
impar(X):- \+ par(X).
```

Ahora se puede comprobar si una lista solo contiene números pares o impares:

```
?- cumplen todos([2,4,6,8,10],par).
true
?- cumplen_todos([1,3,5,7,9],impar).
true
?- cumplen_todos([2,4,5,8,10],par).
False
```

Prolog y los Sistemas de Bases de Datos

Una relación en un sistema de base de datos relacional se define extensionalmente mediante el conjunto de tuplas que la constituyen. Esta misma relación queda definida en un programa Prolog mediante un conjunto de hechos. Además, Prolog permite definir intencionalmente relaciones entre entidades dando definiciones de las mismas. Por ejemplo, en el programa Familia el predicado padre/2 es una relación definida extensionalmente mediante un conjunto de hechos, mientras que el predicado abuelo/2 es una relación definida intencionalmente mediante dos cláusulas.

Luego, un programa lógico puede ser considerado una base de datos relacional muy especial. En la terminología de Prolog, se denomina Base de Datos (BD) al conjunto de hechos y reglas que han sido cargados en el espacio de trabajo del sistema Prolog.

Gestión de la Base de Datos interna de Prolog

Para que un programa Prolog pueda ejecutarse es preciso que los hechos y las reglas que lo componen se hayan cargado en el espacio de trabajo del sistema Prolog. Las reglas que están almacenadas en un fichero y todavía no han sido cargadas en el espacio de trabajo no pueden intervenir en la ejecución del programa. Un sistema Prolog solo reconoce aquello que está en su espacio de trabajo. Por consiguiente, los efectos y las acciones que se realicen durante la ejecución de un programa y un objetivo dependen de lo que allí se encuentre almacenado.

En lo que sigue, se describen una serie de predicados que permiten gestionar la base de datos interna del sistema Prolog y, por lo tanto, modificar el comportamiento de un programa.

Carga de programas

consult(Fich): añade a la base de datos interna de Prolog todas las cláusulas que contenga el fichero Fich. Esto también puede lograrse escribiendo el nombre del fichero entre corchetes, o sea, [Fich]. Si se desea añadir a la base de datos las cláusulas de varios ficheros es posible hacer de una vez [Fich1,Fich2,...].

Consulta

listing(Nom): imprime todas las cláusulas de la base de datos interna de Prolog cuyo símbolo de predicado en la cabeza coincida con Nom.

listing: imprime todas las cláusulas de la base de datos interna de Prolog.

Adición y Eliminación

Los predicados de adición y eliminación permiten modificar dinámicamente la definición de un predicado en la base de datos interna de Prolog.

asserta(C): añade la cláusula C como el primer hecho o regla de la definición del predicado correspondiente.

assertz(C): añade la cláusula C como el _ultimo hecho o regla de la definición del predicado correspondiente.

assert(C): similar a assertz.

retract(C): elimina la cláusula C de la base de datos.

retractall(Head): elimina toda cláusula de la base de datos cuya cabeza unifique con Head.

Existen otros predicados de adición y eliminación (abolish/1, recorda/3, erase/1, etc.) que pueden ser consultados en el manual de SWI-Prolog.

Para poder modificar dinámicamente la definición de un predicado se requiere que este se haya declarado de tipo dinámico mediante la directiva:

:- dynamic Nombre_Predicado/Aridad.

Esta directiva informa al intérprete que la definición del predicado puede cambiar durante la ejecución, mediante el uso de los predicados de adición y eliminación anteriores. En el siguiente ejemplo se informa que los predicados madre/2, padre/2 y abuelo/2 del programa Familia son dinámicos:

:- dynamic madre/2, padre/2, abuelo/2.

Base de Datos Deductivas

Cuando estamos en presencia de una cláusula definida que es un objetivo, nos encontramos con una solicitud de información a una base de datos. Como el intérprete es en esencia un demostrador lógico un programa Prolog puede considerarse como un sistema de base de datos deductivo.

Un sistema de base de datos deductivo, es un sistema de base de datos pero con la diferencia de que permite hacer deducciones a través de inferencias basado, principalmente, en reglas y hechos que son almacenados en la base de datos. Este tipo de base de datos surge debido a las limitaciones de las bases de datos relacionales de responder a consultas recursivas y de deducir relaciones indirectas de los datos almacenados en la base de datos.

Las bases de datos deductivas combinan el modelo relacional para representar los datos con el modelo de la programación lógica y del lenguaje Prolog para representar las reglas. Debido a esto, los lenguajes para crear reglas y consultar bases de datos deductivas tienen una sintaxis semejante a Prolog. Entre estos lenguajes, se destaca el lenguaje Datalog.

Entre sus aplicaciones se encuentran los sistemas de apoyo a la toma de decisiones, sistemas expertos, sistemas de planificación, entre otras.

Metaprogramación

Se conoce como metaprogramación a la escritura de programas, denominados meta programas, que escriben o manipulan, como datos, otros programas (o a s__ mismos). Ejemplo de meta programas son los analizadores sintácticos, que analizan la corrección sintáctica de programas escritos en determinado lenguaje de programación, los analizadores lexicales, intérpretes y compiladores que actúan a nivel sintáctico y/o semántico sobre otros programas.

En principio, un meta programa puede ser escrito en cualquier lenguaje de programación, sin embargo, no en todos se ofrecen facilidades para la metaprogramaci_on2. El lenguaje de los meta programas y el de los programas objetos que aquellos manipulan no tiene que ser necesariamente el mismo, pero para un objetivo tan relevante de la metaprogramación como lo es el manipular semánticamente a programas, la identidad

de ambos resulta un factor conveniente. Esta identidad se logra en los lenguajes de programación en los que no hay diferencia sintáctica entre programa y dato. Satisface Prolog dicha identidad? Como se ha visto, en Prolog hay solo una estructura sintáctica para representar tanto las definiciones de un programa como las estructuras que representan los datos: la cláusula. Luego, como no hay diferencia entre dato y programa, Prolog cumple con la identidad, resultando muy adecuado para escribir meta programas en él.

Meta variables

Una facilidad básica de la metaprogramación la constituye el concepto de meta variable. Las variables en Prolog pueden actuar como *meta variables*, es decir, son variables que pueden tomar como valor un término el cual el intérprete evalúa como un objetivo si el operador principal del término es un predicado y el intérprete tiene acceso a las cláusulas que lo definen, tanto si se trata de un predicado predefinido en el sistema o definido por el usuario en un programa.

Las variables se usan (las detecta el intérprete) como meta variables al colocarlas (ocurrir) como objetivos en el cuerpo de una cláusula. El intérprete evalúa como un objetivo un término que, siendo el valor de una metavariable, lo interpreta como un "llamado" a un predicado.

Ejemplo: en la definición del predicado predicado not/1 la variable X es una metavariable, pues ha sido colocada como sub-objetivo en el cuerpo de la primera cláusula y por tanto así será evaluada por el intérprete.

```
not(X):- X, !, fail.
not(X).
```

Se deja al lector que aprecie como actúa el intérprete cuando X toma como valor un término cuyo operador principal es un predicado previamente definido, por ejemplo, en el programa Familia:
:- not(padre(ana, luis)).

Los intérpretes de Prolog incluyen tradicionalmente un predicado que explícitamente establece la evaluación de una variable como una metavariable, el predicado call/1. El anterior programa es equivalente al siguiente:

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

El lenguaje de programación funcional LISP fue el primer lenguaje que incluyo facilidades para la metaprogramación.

Aplicaciones de la metaprogramación

Además de la construcción de analizadores sintácticos y semánticos, de los programas escritos en un lenguaje de programación dado, otra de las aplicaciones de la metaprogramación es la construcción de metaintérpretes para extender y/o modificar la capacidad básica de un intérprete, es decir, su capacidad de representación y de proceso para solucionar problemas y además su estrategia de navegación en el espacio de búsqueda.

Debido a que Prolog permite acceder directamente al código del programa en tiempo de ejecución, es fácil escribir un intérprete de Prolog en Prolog mismo. El metaintérpretes más simple para Prolog, definido en su propio lenguaje, se muestra en el siguiente programa:

metaS(Obj):- call(Obj).
Como puede verificarse, este metaintérpretes no introduce ninguna extensión significativa al intérprete ya que la evaluación de Obj tiene lugar totalmente en el intérprete de Prolog. Los metaintérpretes comienzan a ser extensiones significativas de un intérprete cuando exhiben cierta granularidad, es decir, cierto grado de detalle del proceso de evaluación de un objetivo que se hace visible y por tanto se reprograman de manera diversa y significativa en el metaintérpretes. Siguiendo esta idea, otro metaintérpretes de Prolog muy popular es el denominado \vainilla" que utiliza la unificación predefinida de Prolog, pero habilita un acceso al motor de búsqueda que permite modificarlo fácilmente. El metaintérpretes \vainilla" es muy importante porque muchos sistemas expertos derivan de él.

```
metaV((Obj1,Obj2)):- !, metaV(Obj1), metaV(Obj2).
metaV(Obj):- functor(Obj, Predicado, Aridad), functor(ObjN, Predicado, Aridad), predicate_property(ObjN, built_in), !, call(Obj).
metaV(Obj):- clause(Obj, Cuerpo), metaV(Cuerpo).
```

Note que la primera cláusula actúa como regla de selección. La segunda cláusula evalúa los objetivos cuyos predicados son predefinidos en Prolog, mientras que la tercera evalúa los objetivos cuyos predicados son definidos en un programa. Analice qué define este metaintérpretes explícitamente y qué deja implícito a realizar por el intérprete.

Programación de Segundo Orden

Como ya se ha dicho en conferencias anteriores, un programa es de *primer orden* si solo manipula y procesa los datos del dominio de un problema y no conjuntos de dichos datos como un todo. Por ejemplo el programa familia es de primer orden pues sus datos son las personas (datos del dominio del problema) y no son conjuntos de personas. La programación estándar es programación de primer orden: en sus programas se procesan individualmente datos de un dominio a los cuales se suele hacer referencia llamándolos “ciudadanos de primera clase”, pero no aplicando operaciones a conjuntos de estos datos (no confundir esto con el hecho de que un programa puede aplicar sus procedimientos a conjuntos de datos leyéndolos uno a uno de un archivo). La programación de primer orden no puede manipular conjuntos ni tampoco crearlos de manera declarativa, tampoco puede manipular de manera efectiva procedimientos funcionales, es decir, procedimientos que toman funciones como argumentos y las aplican a datos de su dominio.

La programación de segundo orden es necesaria dada la necesidad de manipular conjuntos en muchos problemas de una manera natural, es decir, declarativa. La programación de segundo orden además de manipular los datos de un dominio permite manipular conjuntos de esos datos, referirse a sus propiedades y crearlos. Nótese que las funciones y las relaciones no son sino tipos particulares de conjuntos, luego la programación de segundo orden permite generar programas que manipulan relaciones y funciones como datos u objetos de “primera clase”.

Nuevamente la programación declarativa es el marco ideal para la programación de segundo orden y dicho sea de paso de orden superior en general. Una muestra en Prolog de programación de segundo orden la ofrece el siguiente programa basado en un “procedimiento primitivo” denominado findall/3, que ofrecen todos los intérpretes de Prolog.

Predicados findall, setof y bagof

Como se ha visto, el mecanismo de control de Prolog produce respuestas una a una, siguiendo una estrategia en profundidad con retroceso cronológico que recorre las ramas del árbol de búsqueda de izquierda a derecha. Cada vez que se encuentra una respuesta, el intérprete la muestra y se detiene en espera de la indicación del usuario. Si el usuario desea más respuestas, el intérprete de Prolog intenta encontrarlas ayudado de la técnica de retroceso. En el contexto de las bases de datos deductivas, este tipo de estrategias recibe el nombre de \una respuesta cada vez" (tuple-at-time). No obstante, en ocasiones puede ser de interés obtener todas las respuestas a un objetivo (set-at-time). Prolog proporciona para ello una serie de predicados predefinidos que permiten acumular en una lista todas las respuestas del objetivo planteado.

Estos predicados no se detienen cada vez que encuentran una respuesta (una rama triunfo) sino que recorren todo el árbol de derivación del objetivo dado y guardan todas las soluciones (src) para el objetivo en una lista.

findall(F,O,L): crea la lista L con todas las unificaciones de las variables que aparecen en F correspondientes a todas las respuestas, obtenidas por retroceso cronológico, para el objetivo O. Si el árbol de derivación para O no tiene ninguna rama triunfo (no tiene solución), la lista L unifica con la lista vacía.

Ejemplos:

```
?- findall(X, permutacion([1,2,3],X), P).
P = [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]].

?- findall(X,(member(X,[1,2,3,4,5,6]),impar(X)), L).
L = [1,3,5].

?- findall((X:Y), padre(X,Y), P).
P = [luis:jose, luis:alicia, jose:ana].

?- findall(X, padre(X,Y), P).
P = [luis, luis, jose].
```

bagof(F,O,L): crea la lista L con todas las unificaciones de las variables que aparecen en F correspondientes a todas las respuestas, obtenidas por retroceso cronológico, para el objetivo O. Si O tiene variables libres, además de las que aparecen en F, bagof retornará una lista de respuestas obtenidas para cada una de las posibles unificaciones de las variables libres. Si O no tiene soluciones bagof falla. Ejemplos:

```
?- bagof(X, padre(X,Y), P).
Y = alicia,
P = [luis] ;
Y = ana,
P = [jose] ;
Y = jose,
P = [luis].

?- bagof(X, (member(X,[2,4,6]),impar(X)), I).
false.
```

Nota: Ver conferencia #3 el programa familia

Si no se desea iterar por todas las posibles unificaciones de las variables libres que no aparecen en F, estas pueden ligarse al objetivo con la construcción: Var^Obj. Ejemplo:

?- bagof(X, Y^padre(X,Y), P).
P = [luis, luis, jose].

Observe la diferencia entre esta respuesta y la obtenida en el ejemplo anterior. Note además que esta respuesta también se obtiene haciendo uso de un findall (ver último ejemplo para findall).

setof(F,O,L): equivalente a bagof/3 pero la lista L se devuelve ordenada y sin duplicados.

Modularidad en Prolog

La modularidad es un concepto fundamental de la programación y se considera en esta conferencia que todo estudiante conoce las ventajas de utilizar módulos en la construcción de un programa. Por lo tanto nos restringiremos a relatar las características más generales de los módulos en Prolog. La implementación de los módulos en un intérprete tiene características específicas. En nuestro curso serán ejemplificadas con respecto al intérprete SWI Prolog ya escogido.

Un programa Prolog se carga por el intérprete en un espacio que se denomina la base lógica o *base de datos* (BD) del intérprete. El programador puede con su programa dividir este espacio en *módulos*, una zona del espacio que tiene un nombre, el nombre del módulo, en la que se hallan cláusulas que definen diferentes predicados.

Todo módulo tiene una *interfaz*, donde se declaran:

a. Los predicados definidos dentro del módulo que son exportables a otros módulos.
b. Los predicados que el módulo importa que pertenecen a la categoría de los predicados declarados exportables por otros módulos.
c. Un cuerpo donde están las cláusulas de los predicados que el módulo define y exporta, así como las cláusulas que definen cualquier otro predicado de uso local al módulo.

Ejemplo (a):

```
:- module(familia, [abuelo/2, abuela/2]).  % definición del módulo familia y %
lista de los predicados que se exporta el módulo
.
.
.
```

Ejemplo (b)

```
:- module(familia, [abuelo/2, abuela/2]).  %  definición del módulo familia

:- import(computación). % importando todos los predicados que exporta el
% modulo computación

:- import([computacion:profesor/2, estudiante/3]).
% importando del módulo computación sólo los predicados
% profesor/2 y estudiante/3
.
.
.
```

Ejemplo (c):

```
:- module(familia, [abuelo/2, abuela/2]).

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

abuela(X,Y):-madre(X,Z),padre(Z,Y).
abuela(X,Y) :- madre(X,Z),madre(Z,Y).
```

De hecho se puede programar sin considerar la modularidad, ahora bien, siempre existe un módulo por defecto, el módulo **user**, al cual pertenecen todas las cláusulas de un programa que no contiene módulos. El módulo **user** es el módulo en el que se escriben objetivos en el intérprete. De la misma manera, todos los predicados que no aparezcan definidos dentro de un módulo en un programa pertenecen al módulo **user** cuando el programa es cargado en la BD del intérprete.

A continuación un ejemplo elaborado sobre las especificaciones del SWI Prolog:

```
% Modulo permutación: define diferentes métodos de permutar una lista.

:-module(permutación, [permutacion1/2, permutacion2/2]).

permutacion1([],[]).
permutacion1([X|Y],[U|V]):- elimina(U, [X|Y], Z),permutacion1(Z, V).

permutacion2([], []).
permutacion2([X|Y], Z):- permutacion2(Y, W), inserta(X, W, Z).

elimina(X,[X|Y],Y).
elimina(X,[Y|Z],[Y|W]):-elimina(X, Z, W).

inserta(X, L, [X|L]).
inserta(X, [Y|Z],[Y|W]):-inserta(X, Z, W).
```

Otro fichero con otro módulo dentro:

```
% módulo de ordenación de listas: define diferentes métodos de ordenar listas

:-module(ordena_lista,[ ordper1/2]).

:-import(permutacion).

% ordenación por permutación1/2.

ordper1(X,Y):- permutacion1(X,Y),ordenado(Y).

ordenado([]).
ordenado([X]).
ordenado([X,Y|Z]):- X =< Y, ordenado([Y|Z]).
```

Si se cargan ambos módulos en la BD del intérprete, entonces se puede someter a evaluación el siguiente objetivo, donde se ejemplifica cómo hacer referencia a un predicado definido en un módulo (ordena_lista) desde otro módulo (user) sin haberlo importado:

```
?- ordena_lista:ordper1([2,1],X).
```

```
X = [1,2].
```

Todos los predicados definidos en un módulo, así como los que el módulo importa se consideran visibles en dicho módulo y por tanto pueden ser usados sin referenciar el módulo donde han sido definidos. Sólo los predicados que un módulo exporta son importables y por ende visibles por otros módulos.

En la BD del intérprete son visibles los predicados predefinidos (built-in) del sistema, es decir, son visibles a todo módulo o programa en la BD. Con algunas excepciones, según el intérprete en uso, un módulo puede redefinir un predicado del intérprete.

Para modificar dinámicamente los predicados incluidos en cualquier módulo presente en la BD se utilizan assert/1 y retract/1 haciendo énfasis en el módulo que se quiere modificar.

```
assert( familia:madre(alicia,dora) )
%Inserta en el módulo familia la cláusula: madre(alicia,dora)

retract( familia:madre(alicia,dora) )
% Elimina del módulo familia la cláusula madre(alicia,darío).
```

Para saber qué módulo estamos utilizando se utiliza current_module/1 y para cambiar de módulo actual se emplea module/1.

Módulos reservados

SWI-Prolog contiene dos módulos especiales. El primero es el módulo system que contiene todos los predicados predefinidos en el sistema (member/2, append/3, etc.). El segundo es el módulo user que inicialmente está vacío (no contiene predicados definidos) y constituye por defecto el espacio de trabajo del usuario. El módulo user importa el módulo system, luego todos sus predicados están disponibles y también pueden ser redefinidos antes de ser usados. Cualquier módulo definido por el usuario, aunque no lo declare explícitamente, importa el módulo user.

Conclusiones

1. Los lenguajes declarativos son los idóneos para la construcción de metaprogramas, es decir, de programas que operan de alguna manera sobre otros programas.

2. La metaprogramación permite fundamentalmente:
   a) la programación de herramientas para actuar sobre programas (analizadores sintácticos, lexicales, compiladores).
   b) Construir metaintérpretes que modifiquen de manera esencial la conducta del intérprete del lenguaje en el cual se construyen, cambiando, por ejemplo, determinados mecanismos de control que usa el intérprete o haciendo extensiones conservadoras más ricas, tanto expresivas como procedimentales del intérprete de base.

3. Prolog logra la identidad entre instrucción y dato en el lenguaje representándolos por igual como el tipo de expresión lógica denominada término, por lo que se pueden construir metaprogramas en él.

4. Porlog incorpora el concepto de metavariable: variable cuyo valor es un predicado y el de metapredicado: predicado cuyos parámetros son predicados.

5. Se pueden construir metaintérpretes en Prolog para extender la conducta de su intérprete.

6. Un programa Prolog puede ser visto como una base de datos deductiva y cuando el intérprete mantiene en memoria dicha base de datos se pueden efectuar modificaciones dinámicas a los predicados presentes en ella.

7. Prolog brinda facilidades para la programación de segundo orden, al incluir entre sus predicados de sistema las definiciones de findall/3, bagof/3, setof/3.

8. Prolog permite el trabajo modular a través de un conjunto de directivas para este fin.

Con todas estas prestaciones de Prolog, el lector cuenta con más herramientas para definir programas mucho más interesantes que los que hasta ahora conocía. Invitamos a consultar la ayuda del SWI Prolog para obtener más datos y profundizar en otros predicados presentes en las librerías de este intérprete.