Notas de Clase para IL

6. Fundamentos de la Programación Lógica

Rafel Farré, Robert Nieuwenhuis, Pilar Nivela,
Albert Oliveras, Enric Rodríguez, Josefina Sierra

3 de septiembre de 2009

1. Cálculo de respuestas mediante resolución

El modelo de base de datos más utilizado en la actualidad es el *relacional*. Su idea fundamental es expresar los datos mediante *relaciones* o *tuplas*, como en:

$$\begin{array}{l} padre(juan, pedro) \\ \quad \vdots \\ padre(maria, pedro) \\ \\ hermano(pedro, vicente) \\ \quad \vdots \\ hermano(pedro, alberto) \end{array}$$

A diferencia de otros modelos más antiguos como el jerárquico y el de red, el modelo relacional ofrece una gran flexibilidad en las consultas (normalmente con el lenguaje SQL) y en su administración, al hacer abstracción de la forma concreta en que se almacenan los datos (es decir, no es necesario tener en cuenta los detalles de implementación).

1.1. Bases de datos deductivas

Supongamos ahora que queremos tener también la relación *tio*, que exprese quién es tío de quién, por ejemplo $tio(juan, vicente)$. Una opción sería añadir a esta base de datos todas las relaciones de *tio*. Esto requeriría una cantidad cuadrática de espacio, y podría introducir inconsistencias entre relaciones, al ser la relación *tio* una consecuencia lógica de las otras dos. Una solución mejor puede ser el modelo de las bases de datos *deductivas*, que permite añadir *reglas deductivas* como:

$$\forall x \forall y \; (\quad tio(x,y) \quad \textbf{IF} \quad \exists z \; (\; padre(x,z) \wedge hermano(z,y) \;) \quad)$$

que puede escribirse como:

$$\forall x \forall y \; (\quad tio(x,y) \quad \leftarrow \quad \exists z \; (\; padre(x,z) \wedge hermano(z,y) \;) \quad)$$

y como:

$$\forall x \forall y \; (\quad tio(x,y) \quad \vee \quad \neg \exists z \; (\; padre(x,z) \wedge hermano(z,y) \;) \quad)$$

que es lógicamente equivalente a:

$$\forall x \forall y \; (\quad tio(x,y) \quad \vee \quad \forall z \; \neg (\; padre(x,z) \wedge hermano(z,y) \;) \quad)$$

y a:

$$\forall x \forall y \; (\quad tio(x,y) \quad \vee \quad \forall z \; (\; \neg padre(x,z) \vee \neg hermano(z,y) \;) \quad)$$

y a:

$$\forall x \forall y \forall z \; (\quad tio(x,y) \quad \vee \quad (\; \neg padre(x,z) \vee \neg hermano(z,y) \;) \quad)$$

y que, sin los cuantificadores universales, puede escribirse como la cláusula de Horn:

$$tio(x,y) \vee \neg padre(x,z) \vee \neg hermano(z,y)$$

1.2. Programas lógicos

La misma base de datos también puede escribirse como un *programa lógico*. Un programa en el lenguaje de programación *Prolog*, sería por ejemplo:

$$
\begin{array}{l}
padre(juan, pedro).\\
padre(maria, pedro).\\
hermano(pedro, vicente).\\
hermano(pedro, alberto).\\
tio(x, y) \ \texttt{:-} \ padre(x, z), \ hermano(z, y).
\end{array}
$$

A pesar de las diferencias sintácticas, se trata del mismo conjunto de cláusulas que antes. En este ejemplo, hay cuatro cláusulas de un solo literal y una cláusula con tres literales que, desde el punto de vista lógico, es $tio(x, y) \vee \neg padre(x, z) \vee \neg hermano(z, y)$. Todas las cláusulas de un programa Prolog son cláusulas de Horn con exactamente un literal positivo. Son las llamadas *cláusulas de programa* (*program clauses*). Ahora vamos a ver que las consultas a los programas se hacen mediante cláusulas de Horn con sólo literales negativos. Son las llamadas *cláusulas de objetivo* (*goal clauses*).

2. Cálculo de respuestas

Al programa lógico (o la base de datos deductiva) formado por la conjunción de las cinco cláusulas de arriba lo llamaremos F. Supongamos ahora que queremos saber si $F \models \exists u \exists v \ tio(u, v)$, es decir, si en nuestro programa/base de datos existe un sobrino u cuyo tío es v. Por resolución, esto se averigua viendo si $F \ \wedge \ \neg \exists u \exists v \ tio(u, v)$ es insatisfactible, o, equivalentemente si $F \ \wedge \ \forall u \forall v \ \neg tio(u, v)$ es insatisfactible, lo cual en forma clausal (numerada) puede escribirse como:

$$
\begin{array}{ll}
1: & padre(juan, pedro)\\
2: & padre(maria, pedro)\\
3: & hermano(pedro, vicente)\\
4: & hermano(pedro, alberto)\\
5: & tio(x, y) \vee \neg padre(x, z) \vee \neg hermano(z, y)\\
6: & \neg tio(u, v)
\end{array}
$$

donde la cláusula 6 es una cláusula objetivo (proviene de la negación de lo que queremos demostrar). Ahora, por resolución, podemos obtener la siguiente refutación:

$$
\begin{array}{lll}
7: & \neg padre(u, z) \vee \neg hermano(z, v) & (de \ 5 \ y \ 6)\\
8: & \neg hermano(pedro, v) & (de \ 7 \ y \ 1)\\
9: & \square & (de \ 8 \ y \ 3)
\end{array}
$$

Puesto que hemos obtenido la cláusula vacía, sabemos que efectivamente
$F \models \exists u \exists v \ tio(u, v)$.

Pero lo interesante es que en realidad sabemos más: podemos saber quiénes son este sobrino u y su tío v. Si inspeccionamos la refutación, vemos que u se ha *instanciado* con *juan*, y v con *vicente*. Hemos pues *calculado* la *respuesta* $(juan, vicente)$ para la pregunta "$F \models \exists u \exists v \ tio(u, v)$?".

Para ver más cómodamente qué respuesta se ha calculado para u y v, también es posible añadir un literal "fantasma" $resp(u, v)$, es decir, reemplazar la cláusula 6 por $\neg tio(u, v) \vee resp(u, v)$. Entonces la refutación se convierte en:

$$\begin{array}{lll} 7: & \neg padre(u, z) \vee \neg hermano(z, v) \vee resp(u, v) & (de\ 5\ y\ 6) \\ 8: & \neg hermano(pedro, v) \vee resp(juan, v) & (de\ 7\ y\ 1) \\ 9: & resp(juan, vicente) & (de\ 8\ y\ 3) \end{array}$$

y la cláusula vacía se manifiesta en forma de respuesta $resp(juan, vicente)$ a nuestra pregunta.

Podemos calcular tres respuestas más:

$$\begin{array}{lll} 10: & resp(juan, alberto) & (de\ 8\ y\ 4) \\ 11: & \neg hermano(pedro, v) \vee resp(maria, v) & (de\ 7\ y\ 2) \\ 12: & resp(maria, vicente) & (de\ 11\ y\ 3) \\ 13: & resp(maria, alberto) & (de\ 11\ y\ 4) \end{array}$$

2.1. Completitud para el cálculo de respuestas

Existe un teorema de completitud para el cálculo de respuestas (que aquí no demostraremos), y que esencialmente dice lo siguiente:

- Considera un conjunto F de cláusulas de Horn, y un objetivo, una conjunción de átomos de la forma $\exists x_1 \ldots \exists x_n\ A_1 \wedge \ldots \wedge A_m$.
- Sea σ una sustitución *respuesta* (*answer substitution*) de la forma

  $$\{\ x_1 = t_1,\ \ldots,\ x_n = t_n\ \}$$

  que representa una *respuesta correcta* para este programa y objetivo, es decir:

  $$F \models A_1\sigma \wedge \ldots \wedge A_m\sigma$$

- Entonces, por resolución a partir del conjunto de cláusulas

  $$F \quad \wedge \quad (\neg A_1 \vee \ldots \vee \neg A_m \vee resp(x_1, \ldots, x_n))$$

  hay una manera de obtener una cláusula vacía (o *cláusula respuesta*) de la forma $resp(s_1, \ldots, s_n)$, que representa una sustitución σ' de la forma $\{\ x_1 = s_1,\ \ldots,\ x_n = s_n\ \}$ que es al menos tan general como σ, es decir, que $\sigma = \sigma'\sigma''$ para alguna sustitución σ''.

Ejemplo: Si el programa F tiene dos cláusulas $p(a, x)$ y $p(b, b)$, y la pregunta es $\exists u \exists v\ p(u, v)$, una posible respuesta es $\{u = a, v = b\}$, porque $F \models p(a, b)$. Efectivamente, de las cláusulas $\{\ p(a, x),\ p(b, b),\ \neg p(u, v) \vee resp(u, v)\ \}$ obtenemos las cláusulas de respuesta $resp(a, x)$ y $resp(b, b)$. La primera de ellas nos da la respuesta $\{u = a, v = x\}$, que es más general que $\{u = a, v = b\}$.

3. La ejecución de programas Prolog: resolución SLD

Escribamos de nuevo nuestro ejemplo como un programa Prolog, en un fichero llamado `familia.pl`. Las variables de Prolog comienzan con mayúscula o el carácter subrayado "_". Los símbolos de función y de predicado comienzan con minúsculas:

```
padre(juan,pedro).
padre(maria,pedro).
hermano(pedro,vicente).
hermano(pedro,alberto).
tio(X,Y):- padre(X,Z), hermano(Z,Y).
```

Ahora podemos invocar un entorno de Prolog, por ejemplo en Linux con el comando `gprolog` [1]. Nos sale (algo así como) el siguiente mensaje:

```
GNU Prolog 1.2.18
By Daniel Diaz
Copyright (C) 1999-2004 Daniel Diaz
| ?-
```

que es el intérprete de comandos del Prolog. Ahora podemos decirle que compile nuestro programa:

```
| ?- [familia].
compiling familia.pl for byte code...
familia.pl compiled, 5 lines read - 943 bytes written, 39 ms

(4 ms) yes
| ?-
```

El intérprete está ahora preparado para que le hagamos consultas sobre nuestro programa. Las cláusulas de objetivo se escriben ahora sin la negación, y, como las cláusulas de programa, siempre acaban en un punto. Si preguntamos: `| ?- tio(U,V).` Nos contesta:

```
U = juan
V = vicente ?
```

Si queremos que nos dé más respuestas, contestamos con un punto y coma `;`

```
| ?- tio(U,V).

U = juan
V = vicente ? ;

U = juan
V = alberto ? ;
```

[1] GNU Prolog está disponible en `http://www.gprolog.org`

```
U = maria
V = vicente ? ;

U = maria
V = alberto

yes
| ?-
```

3.1. La resolución SLD

La estrategia de resolución de Prolog se llama *resolución SLD*. No vamos a entrar aquí en el significado exacto del acrónimo SLD (*Selection-rule driven Linear resolution for Definite clauses*). Basta con saber lo siguiente.

Dado un objetivo como `?- tio(U,V),` el Prolog busca la primera cláusula por orden de aparición cuyo literal positivo (la *cabeza* de la cláusula) unifique con el objetivo. En este caso es la cláusula `5. tio(X,Y):- padre(X,Z), hermano(Z,Y).` La resolución entre el objetivo y esta cláusula nos da un nuevo objetivo que es `padre(U,Z), hermano(Z,V).` En este nuevo objetivo, procedemos de izquierda a derecha, de la misma manera, unificando con la cláusula `1. padre(juan,pedro).` Pero aquí también unifica con la cláusula `2. padre(maria,pedro)`, por lo que se guarda ésta como una *alternativa para backtracking* para el presente objetivo. Concretamente, se guarda en una *pila de backtracking* que podemos representar como:

```
|                            |                          |
| padre(U,Z), hermano(Z,V). | 2. padre(maria,pedro).  |
_____________________________________________________
        objetivo                alternativa para
                                  backtracking
```

Después de resolver `padre(U,Z), hermano(Z,V)` con la cláusula `1. padre(juan,pedro)`, el nuevo objetivo es `hermano(pedro,V).` Éste se resuelve con la tercera cláusula `hermano(pedro,vicente)`, y se escribe la respuesta

```
U = juan
V = vicente ?
```

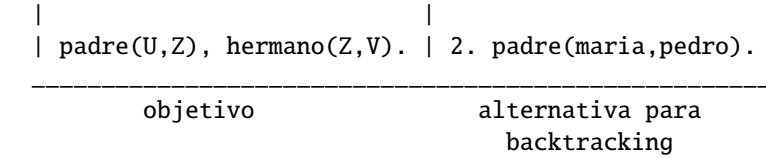
Además, puesto que también era aplicable `4. hermano(pedro,alberto)`, ésta también se empila como alternativa para backtracking y la pila queda:

```
|                            |                              |
| hermano(pedro,V).          | 4. hermano(pedro,alberto).  |
| padre(U,Z), hermano(Z,V). | 2. padre(maria,pedro).      |
_____________________________________________________________
        objetivo                alternativa para
                                  backtracking
```

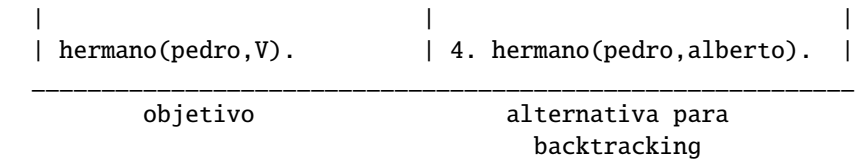
Si ahora pedimos más respuestas (con el punto y coma), el intérprete desempila el estado que hay en la cima de la pila de backtracking: hay que re-hacer el objetivo `hermano(pedro,V)`, esta vez con la cláusula `4. hermano(pedro,alberto).` Esto le permite inmediatamente escribir la respuesta:

```
U = juan
V = alberto ?
```

Puesto que, en nuestro programa, debajo de la cuarta cláusula no hay más alternativas para este objetivo, en este momento no se empila nada, y la pila queda otra vez así:



Si ahora pedimos de nuevo más respuestas, se desempila y se hace resolución sobre el objetivo `padre(U,Z), hermano(Z,V)` con la cláusula `2. padre(maria,pedro).` Después de esto la pila queda vacía. El nuevo objetivo es `hermano(pedro,V)`, y de nuevo podemos unificar con la tercera cláusula, guardando la cuarta como alternativa en la pila:



En este momento se escribe la respuesta:

```
U = maria
V = vicente ?
```

Si pedimos más respuestas, se desempila y se rehace el objetivo `hermano(pedro,V)`, esta vez con la cláusula `4. hermano(pedro,alberto)` y se escribe la respuesta:

```
U = maria
V = alberto ?
```

Después de esto la pila queda vacía: no existen más respuestas.

3.2. Ejemplos de Prolog. Unificación, listas.

Podemos hacer consultas al intérprete de Prolog sobre unificación, usando el predicado "=" y su negación "\=" (no confundir con la igualdad de la LPOI):

```
| ?- f(X,a)=f(b,Y).

X = b
```

```
Y = a
yes

| ?- f(f(X),a)=f(Y,Y).
no

| ?- f(f(X),a)\=f(Y,Y).
yes

| ?- f(f(X),a)=f(Y,Z).

Y = f(X)
Z = a
yes
```

Y también podemos utilizar la notación para *listas*, en la que el operador "|" separa los primeros elementos de la lista de la lista con los demás elementos, y `[]` denota la lista vacía:

```
| ?- [X,Y|L]=[a,b,c].

L = [c]
X = a
Y = b
yes

| ?- [X|L]=[Y,Z,a,g(b),c].

L = [Z,a,g(b),c]
Y = X
yes
```

Podemos ahora escribir un programa `listas.pl` que define la pertenencia a una lista, y también la concatenación de listas:

```
pert(X,[X|_]).             % o bien X es el primero, o bien
pert(X,[_|L]):- pert(X,L). % pertenece a la lista de los demás

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

Aquí el subrayado denota una variable cuyo nombre no nos importa y los `%` indican que el resto de la línea es un comentario. Nótese que programar en Prolog consiste en *declarar* o definir las cosas, en vez de *mandar* con instrucciones. Por eso se habla de *programación declarativa*, en contraposición con los lenguajes *imperativos* tradicionales. Ahora podemos hacer preguntas al programa sobre listas:

```
| ?- [listas].
```

```
compiling listas.pl for byte code...
listas.pl compiled, 4 lines read - 1570 bytes written, 32 ms

yes
| ?- pert(X,[a,b,c]).

X = a ? ;
X = b ? ;
X = c ? ;
no

| ?- concat([a,b,c],[c,d,e],L).

L = [a,b,c,c,d,e]
yes

| ?- concat(L1,L2,[a,b,c]).

L1 = []
L2 = [a,b,c] ? ;

L1 = [a]
L2 = [b,c] ? ;

L1 = [a,b]
L2 = [c] ? ;

L1 = [a,b,c]
L2 = [] ? ;
no

| ?- concat([X|L],[a,Y,b,c],[1,2,a,Z,b,Z]).

L = [2]
X = 1
Y = c
Z = c ? ;
no
```

Nótese que el predicado `pert` también podría haber sido definido alternativamente con `pert(X,L):- concat(_,[X|_],L).`

3.3. La programación recursiva y la inducción

Tanto en matemáticas como en programación, a menudo se trabaja con *definiciones recursivas*: se define un concepto en términos de un caso más sencillo de ese mismo

concepto.

Por ejemplo, sabemos que el *factorial* $n!$ de un número n es $n \cdot (n-1) \cdots 1$. Pero a menudo no es posible hacer este tipo de definiciones con puntos suspensivos, o no resultan suficientemente precisas (por ejemplo, 0! qué es?). Una definición recursiva del factorial dice, *como caso base*, que $fact(0) = 1$, y, *como caso recursivo*, que si $n > 0$ entonces $fact(n) = n \cdot fact(n-1)$.

Es obvio que las definiciones recursivas *están estrechamente relacionadas con las demostraciones por inducción*. Por ejemplo, podemos demostrar por inducción que para todo número natural k el factorial de k está bien definido con nuestra definición recursiva: para el caso $k = 0$ lo está, y para el caso $k > 0$ lo está, asumiendo que es así para $k - 1$.

En informática, las definiciones recursivas resultan extremadamente útiles, porque se pueden convertir de manera fácil en programas. Por ejemplo, en el lenguaje `C` o en `C++`, podemos definir la función

```
unsigned int factorial( unsigned int n ) {
  if (n==0) return(1);
  else return(n * factorial(n-1));
}
```

En Prolog pasa lo mismo. Hemos visto la definición recursiva de pertenencia a listas:

```
pert(X,[X|_]).              % caso base: X es el primero,
pert(X,[_|L]):- pert(X,L).  % caso recursivo: lista mas corta
```

Para construir un programa recursivo como éste, y al mismo tiempo convencernos de que es correcto, es necesario pensar por inducción. En este caso, la primer cláusula es el caso base: `X` pertenece a cualquier lista de la cual es el primer elemento. El caso inductivo, la segunda cláusula, dice que `X` también pertenece a una lista si pertenece a la lista obtenida al quitar el primer elemento.

Otra forma de convencernos de que un programa recursivo es correcto es intentando ejecutarlo "a mano" y ver que el comportamiento es el esperado. Pero eso resulta muy engorroso, es fácil equivocarse, y tampoco es posible hacerlo para entradas no-triviales. Por eso es muy preferible limitarse a pensar por inducción, directamente al escribir el programa[2]. Hagamos otro ejemplo, la definición recursiva que vimos de la concatenación de listas:

```
concat([],L,L).                              % caso base: []
concat([X|L1],L2,[X|L3]):- concat(L1,L2,L3). % al menos un elem
```

Este programa es correcto por inducción sobre la longitud k de la primera lista. Caso base: si $k = 0$ (lista vacía), es correcto por la primera cláusula; paso de inducción ($k > 0$): si hay un primer elemento `X` (lista no-vacía), el primer elemento de la concatenación debe ser `X`, y el resto de los elementos (la lista `L3`) se construye con la llamada recursiva `concat(L1,L2,L3)` que es correcta por hipótesis de inducción ya que la longitud de `L1` es $k - 1$.

[2]Como decía el ilustre investigador pionero en informática, E. Dijkstra, "la programación y la verificación han de ir de la mano".

3.4. Los aspectos extra-lógicos de Prolog

Hasta ahora los programas Prolog que hemos visto son puramente lógicos: su ejecución consiste puramente en la estrategia SLD de resolución. Por eso son capaces de contestar a preguntas formuladas de todas las maneras posibles. Ya lo hemos visto en el `concat`, y lo mismo pasa con el de `familia.pl`: además de calcular todas las parejas sobrino-tío, podemos preguntar cosas como

```
| ?- tio(S,alberto).
S = juan ? ;
S = maria
yes

| ?- tio(juan,T).
T = vicente ? ;
T = alberto
yes

| ?- tio(juan,alberto).
yes
```

Pero en la práctica, por razones de eficiencia (entre otras), no basta con la resolución como único mecanismo de cómputo.

Aritmética predefinida: Por ejemplo, es necesario disponer de aritmética predefinida. En Prolog, si un término está formado por operadores aritméticos y constantes (enteras o reales), entonces se puede *evaluar* mediante el predicado `is`. Si no es evaluable, se produce un error de ejecución. Entre muchos otros, existen los operadores `+, -, *, /, mod, //` (este último es la división entera) y los predicados `is, <, >, =<, >=`. Nótese la diferencia entre la unificación con "=" y el `is`:

```
| ?- X = 2+3.
X = 2+3
yes

| ?- X is 2+3.
X = 5
yes

| ?- X is 1/3.
X = 0.33333333333333331
yes

| ?- X is (7+3) mod 2.
X = 0
yes

| ?- 3 is 2+3.
```

```
no

| ?- 2 is X+1.
uncaught exception: error(instantiation_error,(is)/2)

| ?- X is f(1).
uncaught exception: error(type_error(evaluable,f/1),(is)/2)
```

Un ejemplo de programa que utiliza la aritmética es, por ejemplo, el siguiente predicado `fact(N,F)` que significa que el factorial de `N` es `F`:

```
fact(0,1).
fact(X,F):-  X1 is X - 1, fact(X1,F1), F is X * F1.
```

Nótese que este programa sólo puede calcular `F` a partir de `N`, pero no al revés: no puede calcular `N` a partir de `F`. Además, este programa bajo backtracking genera llamadas `fact(-1,F)`, `fact(-2,F)`, etc., por lo que se produce un error de ejecución si se le piden más resultados:

```
| ?- fact(6,F).
F = 720 ?
yes

| ?- fact(N,720).
uncaught exception: error(instantiation_error,(is)/2)

| ?- fact(6,F).
F = 720 ? ;
Fatal Error: local stack overflow
```

Entrada/salida: Otro aspecto extra-lógico es el de la entrada/salida. Por ejemplo, el predicado unario `write` desde el punto de vista lógico siempre se cumple, pero tiene el efecto secundario extra-lógico de escribir algo. Eso mismo pasa con el `nl` (salto de línea) o el `read`, etc, (también hay muchos otros predicados, gestión de ficheros, etc.). Podemos listar directamente todas las respuestas a una pregunta (sin usar el punto y coma) si usamos el `write` y forzamos el backtracking con el `fail`, un predicado que siempre falla (como `1 is 2` o algo similar):

```
| ?- tio(U,V), write([U,V]), nl, fail.
[juan,vicente]
[juan,alberto]
[maria,vicente]
[maria,alberto]
no
```

Consideremos ahora el generador de números naturales:

```
nat(0).
nat(N):- nat(N1), N is N1 + 1.
```

Este programa escribe "todos" (pero no acaba) los números naturales con la pregunta:

```
| ?- nat(N), write(N), nl, fail.
0
1
2
3
...
```

En base a él, podemos definir el mínimo común múltiplo (mcm) de dos números:

```
mcm(X,Y,M):- nat(M), M>0, 0 is M mod X, 0 is M mod Y.
```

u otra versión un poco más eficiente:

```
mcm(X,Y,M):- nat(N), N>0, M is N * X,   0 is M mod Y.
```

El operador de corte "!": Para obtener una versión del programa del factorial que no dé error de ejecución al pedirse más respuestas, podemos usar el *operador de corte*, que se denota por "`!`". En el siguiente programa:

```
fact(0,1):-!.
fact(X,F):-  X1 is X - 1, fact(X1,F1), F is X * F1.
```

desde el punto de vista lógico un corte "`!`" (como el que hay en la primera cláusula) siempre se satisface. Pero aquí además indica que, si se consigue unificar con `fact(0,1)`, entonces no hay que considerar la siguiente cláusula como alternativa: no hay que empilarla en la pila de backtracking como opción alternativa. Si ahora repetimos el experimento con esta nueva versión del `fact`, ya no nos da opción a pedir más respuestas:

```
| ?- fact(6,F).
F = 720
yes
| ?-
```

En general, el comportamiento del "`!`" se define como sigue. Sea un programa de la forma

```
p(...):- ...
...
p(...):- q1(...), ..., qn(...), !, r(...), ...
...
p(...):- ...
```

y considera un objetivo en curso de la forma `p(...), A1, ... Ak`. Asume además que la pila de backtracking está en estado E antes de comenzar a tratarse la llamada `p(...)` del objetivo en curso. Si la ejecución llega al "`!`", se elimina de la pila todas las alternativas para tratar `p(...)` y para tratar los objetivos `q1(...), ..., qn(...)`,

es decir, la pila vuelve a estar en estado E y se continúa con el objetivo `r(...), A1, ... Ak`.

Más ejemplos: Considera las siguientes definiciones:

```
pert_con_resto(X,L,Resto):- concat(L1,[X|L2], L    ),
                            concat(L1,   L2,  Resto).

long([],0).   % longitud de una lista
long([_|L],M):- long(L,N), M is N+1.

factores_primos(1,[]) :- !.
factores_primos(N,[F|L]):- nat(F), F>1, 0 is N mod F,
                N1 is N // F, factores_primos(N1,L),!.

permutacion([],[]).
permutacion(L,[X|P]) :- pert_con_resto(X,L,R), permutacion(R,P).

subcjto([],[]).  %subcjto(L,S) es: "S es un subconjunto de L".
subcjto([X|C],[X|S]):-subcjto(C,S).
subcjto([_|C],S):-subcjto(C,S).
```

Permiten hacer consultas como:

```
| ?- pert_con_resto(X,[a,b,c],R), write(X), write(R), nl, fail.
a[b,c]
b[a,c]
c[a,b]
no

| ?- long([a,b,c,d],N).
N = 4
yes

| ?- factores_primos(120,L).
L = [2,2,2,3,5]
yes

| ?- permutacion([a,b,c],P), write(P), write('  '), fail.
[a,b,c]  [a,c,b]  [b,a,c]  [b,c,a]  [c,a,b]  [c,b,a]
no

| ?- subcjto([a,b,c],S), write(S), write(' '), fail.
[a,b,c] [a,b] [a,c] [a] [b,c] [b] [c] []
no
```

En base a estas definiciones, podemos hacer por ejemplo el juego llamado *cifras*: dada una lista L de enteros y un entero N, encontrar una manera de obtener N a base de su-

mar, restar y multiplicar elementos de L (usando cada uno tantas veces como aparezca en L):

```
cifras(L,N):- subcjto(L,S), permutacion(S,P), expresion(P,E),
              N is E, write(E),nl,fail.

expresion([X],X).
expresion(L,E1+E2):- concat(L1,L2,L),  L1\=[],L2\=[],
                     expresion(L1,E1), expresion(L2,E2).
expresion(L,E1-E2):- concat(L1,L2,L),  L1\=[],L2\=[],
                     expresion(L1,E1), expresion(L2,E2).
expresion(L,E1*E2):- concat(L1,L2,L),  L1\=[],L2\=[],
                     expresion(L1,E1), expresion(L2,E2).
```

Este programa simplemente genera y comprueba todas las expresiones formadas a partir de subconjuntos permutados de L. Escribe miles de soluciones en pocos segundos:

```
| ?- cifras( [4,9,8,7,100,4], 380 ).
4 * (100 - 7) + 8
((100 - 9) + 4) * 4
4 * (9 + 100) - 7 * 8
7 * (8 * (9 - 4)) + 100,
...
```

Otro ejemplo, cuya programación en un lenguaje imperativo como `C`, `C++`, o `Java` ocuparía muchas líneas y costaría muchas horas, es el del cálculo de derivadas, donde `der(E,X,D)` significa que la derivada de la expresión `E` con respecto de `X` es `D`:

```
der(X,X,1):-!.
der(C,_,0):-               number(C).
der(A+B,X,DA+DB):-         der(A,X,DA),der(B,X,DB).
der(A-B,X,DA-DB):-         der(A,X,DA),der(B,X,DB).
der(A*B,X,A*DB+B*DA):-     der(A,X,DA),der(B,X,DB).
der(sin(A),X,cos(A)*DA):-  der(A,X,DA).
der(cos(A),X,-sin(A)*DA):- der(A,X,DA).
der(e^A,X,DA*e^A):-        der(A,X,DA).
der(ln(A),X,DA*1/A):-      der(A,X,DA).
...
```

Este programa simplemente enuncia las regla de derivación. Podemos usarlo, por ejemplo, así:

```
| ?- der( 2*x*x + 3*x, x, D), simplifica(D,D1).
D  = 2*x*1+x*(2*1+x*0)+(3*1+x*0)
D1 = 4*x + 3
yes
| ?-
```

Todas las definiciones Prolog mencionadas aquí (incluyendo las de `simplifica` de este último ejemplo) están disponibles en la página web de la asignatura.

4. Ejercicios

Notación: "dado N" significa que la variable N estará instanciada inicialmente. "Debe ser capaz de generar todas las respuestas posibles" significa que si hay backtracking debe poder generar la siguiente respuesta, como el `nat(N)` puede generar todos los naturales.

1. (dificultad 1) Demuestra por inducción que son correctos los programas para `pert_con_resto`, `long`, `permutación` y `subcjto`.

2. (dificultad 1) Escribe un predicado Prolog `prod(L,P)` que signifique "`P` es el producto de los elementos de la lista de enteros dada `L`". Debe poder generar la `P` y también comprobar una `P` dada.

3. (dificultad 1) Escribe un predicado Prolog `pescalar(L1,L2,P)` que signifique "`P` es el producto escalar de los dos vectores `L1` y `L2`". Los dos vectores vienen dados por las dos listas de enteros `L1` y `L2`. El predicado debe fallar si los dos vectores tienen una longitud distinta.

4. (dificultad 2) Representando conjuntos con listas sin repeticiones, escribe predicados para las operaciones de intersección y unión de conjuntos dados.

5. (dificultad 2) Usando el `concat`, escribe predicados para el último de una lista dada, y para el inverso de una lista dada.

6. (dificultad 3) Escribe un predicado Prolog `fib(N,F)` que signifique "`F` es el `N`-ésimo número de Fibonacci para la N dada". Estos números se definen como: $fib(1) = 1$, $fib(2) = 1$, y, si $N > 2$, como: $fib(N) = fib(N-1) + fib(N-2)$.

7. (dificultad 3) Escribe un predicado Prolog `dados(P,N,L)` que signifique "la lista `L` expresa una manera de sumar `P` puntos lanzando `N` dados". Por ejemplo: si `P` es 5, y `N` es 2, una solución sería `[1,4]`. (Nótese que la longitud de `L` es `N`). Tanto `P` como `N` vienen instanciados. El predicado debe ser capaz de generar todas las soluciones posibles.

8. (dificultad 2) Escribe un predicado `suma_demas(L)` que, dada una lista de enteros `L`, se satisface si existe algún elemento en `L` que es igual a la suma de los demás elementos de `L`, y falla en caso contrario.

9. (dificultad 2) Escribe un predicado `suma_ants(L)` que, dada una lista de enteros `L`, se satisface si existe algún elemento en `L` que es igual a la suma de los elementos anteriores a él en `L`, y falla en caso contrario.

10. (dificultad 2) Escribe un predicado `card(L)` que, dada una lista de enteros `L`, escriba la lista que, para cada elemento de `L`, dice cuántas veces aparece este elemento en `L`. Por ejemplo, `card( [1,2,1,5,1,3,3,7] )` escribirá

    `[[1,3],[2,1],[5,1],[3,2],[7,1]]`.

11. (dificultad 2) Escribe un predicado Prolog `está_ordenada(L)` que signifique "la lista `L` de números enteros está ordenada de menor a mayor". Por ejemplo, con `?-está_ordenada([3,45,67,83]).` dice `yes`

    Con `?-está_ordenada([3,67,45]).` dice `no`.

12. (dificultad 2) Escribe un predicado Prolog `ordenación(L1,L2)` que signifique "`L2` es la lista de enteros `L1` ordenada de menor a mayor". Por ejemplo: si `L1` es `[8,4,5,3,3,2]`, `L2` será `[2,3,3,4,5,8]`. Hazlo en una línea, utilizando sólo los predicados `permutación` y `está_ordenada`.

13. (dificultad 2) ¿Qué número de comparaciones puede llegar a hacer en el caso peor el algoritmo de ordenación basado en `permutación` y `está_ordenada`?

14. (dificultad 3) Escribe un predicado Prolog `ordenación(L1,L2)` basado en el método de la inserción, usando un predicado `insercion(X,L1,L2)` que signifique: "`L2` es la lista obtenida al insertar `X` en su sitio en la lista de enteros `L1` que está ordenada de menor a mayor".

15. (dificultad 2) ¿Qué número de comparaciones puede llegar a hacer en el caso peor el algoritmo de ordenación basado en la inserción?

16. (dificultad 3) Escribe un predicado Prolog `ordenación(L1,L2)` basado en el método de la fusión (*merge sort*): si la lista tiene longitud mayor que 1, con `concat` divide la lista en dos mitades, ordena cada una de ellas (llamada recursiva) y después fusiona las dos partes ordenadas en una sola (como una "cremallera"). Nota: este algoritmo puede llegar a hacer como mucho $n \log n$ comparaciones (donde n es el tamaño de la lista), lo cual es demostrablemente óptimo.

17. (dificultad 4) Escribe un predicado `diccionario(A,N)` que, dado un alfabeto `A` de símbolos y un natural `N`, escriba todas las palabras de `N` símbolos, por orden alfabético (el orden alfabético es según el alfabeto `A` dado). Ejemplo: `diccionario( [ga,chu,le],2)` escribirá:

    ```
    gaga gachu gale chuga chuchu chule lega lechu lele.
    ```

    Ayuda: define un predicado `nmembers(A,N,L)`, que utiliza el `pert` para obtener una lista `L` de `N` símbolos, escríbe los símbolos de `L` todos pegados, y provoca backtracking.

18. (dificultad 3) Escribe un predicado `palíndromos(L)` que, dada una lista de letras `L`, escriba todas las permutaciones de sus elementos que sean palíndromos (capicúas).

    Ejemplo: `palindromos([a,a,c,c])` escribe `[a,c,c,a]` y `[c,a,a,c]`.

19. (dificultad 4) ¿Qué 8 dígitos diferentes tenemos que asignar a las letras `S,E,N,D,M,O,R,Y`, de manera que se cumpla la suma `SEND+MORE=MONEY`? Resuelve el problema en Prolog con un predicado `suma` que sume listas de dígitos. El programa debe decir "no" si no existe solución.

20. (dificultad 4) Escribe el predicado `simplifica` que se ha usado con el programa de calcular derivadas.

21. (dificultad 4) Tres misioneros y tres caníbales desean cruzar un río. Solamente se dispone de una canoa que puede ser utilizada por 1 ó 2 personas: misioneros ó caníbales. Si los misioneros quedan en minoría en cualquier orilla, los caníbales se los comerán. Escribe un programa Prolog que halle la estrategia para que todos lleguen sanos y salvos a la otra orilla.