Modificación dinámica del programa

- Prolog está relacionado con la inteligencia artificial entre otros motivos por la capacidad de los programas Prolog de modificarse a sí mismos
  - Si un programa es un conjunto de reglas que representan conocimiento, un programa puede **aprender** modificándose a sí mismo...
- Aunque la complejidad de muchas de estas ideas es mayor de la que se pensó en su momento, estas técnicas siguen siendo interesantes y útiles.
- En cualquier caso, la modificación dinámica de un programa **debe hacerse con cuidado**, pues los programas resultantes pueden ser muy difíciles de mantener.
- Pero en determinados casos es muy conveniente.

Modificación dinámica del programa

- Un programa Prolog es un conjunto de reglas (hechos, cláusulas). Estas reglas están almacenadas en la memoria de forma similar a los intérpretes que hemos visto en semanas anteriores.
- Existen predicados para modificar el conjunto de reglas del programa:
  - `asserta(C)` introduce la cláusula `C` **al principio** del conjunto de reglas del predicado correspondiente a `C`.
  - `assertz(C)` introduce la cláusula `C` **al final** del conjunto de reglas del predicado correspondiente a `C`.
  - `retract(C)` elimina **la primera cláusula** del predicado que unifica con `C`, unificando las variables de `C`. En *backtracking*, elimina las siguientes cláusulas del predicado.
  - `retractall(C)` elimina **todas las cláusulas** del predicado correspondiente a `C`.
- En cualquiera de estos predicados, `C` puede ser un **hecho** (nombre de predicado seguido de sus argumentos) o una **cláusula** (utilizando el operador `':-'`).
- Existen otros predicados (`recorda/1`, `abolish/1`,...).

Modificación dinámica del programa (cont.)

- **Ejemplos**:

```
?- asserta(p(a,b)).
true.
?- assertz(p(a,c)).
true.
?- asserta(p(A,C)).
true.
?- listing(p/2).
:- dynamic p/2.
p(_, _).
p(a, b).
p(a, c).
true.
?- retractall(p(a,b)).
true.
```

```
?- asserta(p(a,b)).
true.
?- assertz(q(r,s)).
true.
?- asserta(p(X,Y):-q(X,Y)).
true.
?- listing(p/2).
:- dynamic p/2.
p(A, B) :- q(A, B).
p(a, b).
true.
?- p(X,Y).
X = r, Y = s ;
X = a, Y = b.
```

Modificación dinámica del programa (cont.)

- Las cláusulas añadidas al programa tienen efecto en la **siguiente** ejecución del predicado que está siendo modificado:
- **Ejemplo**:

```
?- assertz((p(A):- assertz(p(A)),fail)).
true.
?- p(a).
false.
?- listing(p).
:- dynamic p/1.
p(A) :-
    assertz(p(A)),
    fail.
p(a).
true.
```

- **Ejercicio**: ¿Qué ocurre si evaluamos el siguiente objetivo?

```
?- p(X),p(b).
```

Modificar dinámicamente el programa puede hacerlo inmantenible.

Modificación dinámica del programa (cont.)

- Uno de los usos tradicionales de este tipo de técnicas es para **almacenar conocimiento** que ya ha sido calculado anteriormente.
  Por ejemplo, el predicado siguiente:

```
tabla(L) :-
    member(X,L), member(Y,L), V is X*Y,
    assertz(mult(X,Y,V)), fail.
```

- Si evaluamos el siguiente objetivo:

```
?- tabla([1,2,3,4,5,6,7,8,9]).
```

  ¿Cuál es el resultado?

Modificación dinámica del programa (cont.)

- Uno de los usos tradicionales de este tipo de técnicas es para **almacenar conocimiento** que ya ha sido calculado anteriormente.
  Por ejemplo, el predicado siguiente:

```
tabla(L) :-
    member(X,L), member(Y,L), V is X*Y,
    assertz(mult(X,Y,V)), fail.
```

- Si evaluamos el siguiente objetivo:

```
?- tabla([1,2,3,4,5,6,7,8,9]).
```

  ¿Cuál es el resultado?
- El objetivo falla, pero **como efecto lateral** obtenemos una colección de hechos *mult(X,Y,V)* que contienen la **tabla de multiplicar**.
- Esta técnica para recorrer las soluciones de un objetivo se denomina **bucle de fallo**.

Modificación dinámica del programa (cont.)

- Una versión modificada del predicado de cálculo de fibonacci:

```
:- dynamic tab_fib/2.
tab_fib(0,1).
tab_fib(1,1).

fibTabulado(N,F):- tab_fib(N,F), !.
fibTabulado(N,F):-
    N1 is N-1, N2 is N-2,
    fibTabulado(N1,F1), fibTabulado(N2,F2),
    F is F1+F2, assert(tab_fib(N,F)).
```

- La declaración `:- dynamic tab_fib/2.` indica que el predicado `tab_fib` de aridad 2 es **dinámico**: puede cambiar durante la ejecución (por medio de assert/retract).
- El predicado `fibTabulado(N,F)` calcula el valor del N-ésimo término de la serie utilizando tabulación o caching o memoization, es decir, utilizando la tabla `tab_fib`.

Modificación dinámica del programa (cont.)

- El problema de estas técnicas es que su uso produce **efectos laterales** y **globales**, que no desaparecen tras la resolución ni el *backtracking*.
- Los programas pueden no tener un sentido declarativo independiente de la ejecución.
- Se puede perder la "transparencia referencial": idénticas llamadas al mismo predicado pueden proporcionar resultados diferentes.
- Conclusión: **debe hacerse un uso muy controlado** de ellos, habitualmente en forma de hechos:
  - Para almacenar conocimiento obtenido previamente.
  - Para almacenar propiedades globales.
- **Ejercicio:** Diseña un predicado `contador(X)` que en sucesivas llamadas unifique el argumento con números naturales distintos comenzando en 1.
- **Ejercicio:** Diseña utilizando `assert` y `retract` un predicado `copy_term(X,Y)` que a partir de un término `X` proporcione otro término igual `Y` pero con todas las variables renombradas.

Precedencia de operadores

- Anteriormente hemos visto que en Prolog existen operadores predefinidos: `is/2`, `^/2`, los operadores aritméticos, etc.
- Si no existieran, el objetivo `X is 3*4+Y/2` se debería escribir `is(X,+(*(3,4),/(Y,2)))`.
- Los operadores tienen tres características fundamentales:
  - La **posición del operador**: prefija (por ejemplo, `-3`), infija (`2 + 3`), o postfija.
  - La **precedencia**: `2 + 3 * 4` se lee como `2 + (3 * 4)`.
  - La **asociatividad**: define cómo se interpretan expresiones como `8 - 5 - 2` (asociativo *por la derecha*: `8 - (5 - 2)`).
- Las reglas de precedencia se pueden ignorar utilizando paréntesis.