Metaintérprete Vanilla. Ejercicios.

Metaintérprete original. Vanilla básico (el que se adjunta en el examen)

```
solve(true):-!.
solve((A,B)) :-!, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

Cosas importantes que he ido descubriendo según reflexionaba algún ejercicio:

1. solve(Meta, *) es un predicado que definimos nosotros y que poco tiene que ver con Prolog. Podemos hacer con él lo que queramos.
2. true es una constante predefinida de Prolog.
3. **clause (A,B) es un predicado predefinido:** dada una meta que "ponemos en A", **busca en** el conjunto de claúsulas **-nuestro programa interpretado por el intérprete** :D, es decir, valor(w1,1), conectado(w1,w2), etc.- **con quien unifica A y pone en B el cuerpo de la regla**; si no es una regla, es un hecho, entonces pone la claúsula vacía, que resulta ser siempre true.

Ejercicio 1 (Examen 2013, regla de cómputo).

Modificar el meta intérprete vanilla para obtener un intérprete que utilice como **regla de cómputo "1er literal a la derecha".**

```
solve(true):-!.
solve((A,B)) :-!, solve(B), solve(A).
solve(A) :- clause(A, B), solve(B).
```

Podemos ver que en la regla resaltada en negrita, se ha cambiado en el cuerpo A por B con respecto al metaintérprete orginal. Estamos modificando la regla de cómputo de nuestro intérprete, ya que cuando tiene un par de literales (A,B), le estamos indicando (en la segunda regla) que primero se resuelve B y luego A. Y con esto estamos diciendo que cualquier programa a resolver por nuestro intérprete, el primer literal a coger es de la derecha.

Ejercicio 2 (Examen 2013, profundidad limitada).

Modificar el meta intérprete vanilla para obtener un intérprete que realice una **búsqueda en profundidad limitada**. La profundidad máxima será un argumento adicional que se instanciará en la llamada.

```
solve_pmax(true, _) :- !.
solve_pmax((A,B), Prf) :- !, solve_pmax(A, Prf), solve_pmax(B, Prf).
solve_pmax(A, Prf) :- Prf > 0, clause(A,B), Nprf is Prf-1, solve_pmax(B, Nprf).
```

Hay que fijarse en dos cosas:

1. La primera y segunda regla vienen a ser las de siempre, pero como ahora solve() tiene un argumento más (el máximo de profundidad), hay que añadirlo para que no dé errores de sintaxis.
2. En la tercera regla, lo único que hay distinto con respecto a la otra es que introducimos literales para contar. El que más a la izquierda está condiciona la terminación. El tercer literal resta uno al nivel que haya; es decir, si has introducido como tope 3 niveles, se irá restando uno cada vez que se haga una llamada recursiva, hasta que Prf sea 0, el literal Prf > 0 será falso y por tanto la búsqueda cesará.

Ejercicio 3. (Examen 2013, Prueba=())

Extender el meta intérprete vanilla básico para o**btener un meta intérprete que genere la prueba de las metas resueltas con éxito.** La prueba será un argumento adicional que no se instancia en la llamada.

```
builtin(A is B).      builtin(A = B).       builtin(A >= B).
builtin(read(X)).     builtin(A > B).       builtin(A < B).
builtin(A =:= B).     builtin(A =< B).      builtin(functor(T, F, N)).
builtin(write(X)).
```

```
solve(true,true) :- !.
solve((A, B), (ProofA, ProofB)) :- !, solve(A, ProofA), solve(B, ProofB).
solve(A, (A:-builtin)):- builtin(A), !, A.
solve(A, (A:-Proof)) :- clause(A, B), solve(B, Proof).
```

En este, pasa como en el anterior: al tener un parámetro a mayores en solve(), necesitamos modificar todas las reglas para admitir este nuevo parámetro. Pasan cosas que podemos considerar interesantes:

1. Las consultas se hacen con ?- solve (loquehayquedemostrar(tal), Prueba). En “Prueba”, se van almacenando todas las reglas con las que se ha unificado con éxito, de manera que queda descrito el camino que se ha seguido para alcanzar la solución.
2. Hay que introducir una nueva regla (la tercera, builtin). Pienso que es porque las funciones predefinidas hacen algo sin más, es decir, no siguen el proceso búsqueda/cómputo, no son true o false, son builtin. Por tanto, al “tener que resolver” un predicado predefinido, la prueba será algo así:

   ```
   ?- solve(write('Hola'),Prueba).
   Hola
   Prueba = (write('Hola'):-builtin).
   ```

Ejercicio 4 (Pintar la traza).

Modificar el meta intérprete vanilla para obtener un intérprete que muestre la traza de las metas que va resolviendo, mostrando el nivel de las mismas.

```
builtin(A is B).      builtin(A = B).       builtin(A >= B).
builtin(read(X)).     builtin(A > B).       builtin(A < B).
builtin(A =:= B).     builtin(A =< B).      builtin(functor(T, F, N)).
builtin(write(X)).

solve_traza(Meta)  :- solve_traza(Meta, 0).
solve_traza(true, Prf) :- !.

solve_traza((A,B), Prf) :- !, solve_traza(A, Prf), solve_traza(B, Prf).
solve_traza(A, Prf):- builtin(A), !, A, display(A, Prf), nl.

solve_traza(A, Prf) :- clause(A,B),
                       display(A, Prf),
                       nl, Prf1 is Prf +1,
                       solve_traza(B, Prf1).

display(A, Prf):- Espacios is 3*Prf, tab(Espacios), write(Prf), tab(1),
write(A).
```

En este también tenemos cosas interesantes, se juega un poco con todo lo anterior. Vamos a poner la salida que da Prolog a preguntas con el programa valor(w1,1), conectado(w1,w2), etc.:

```
1 ?- solve_traza(valor(w3,X)).
0 valor(w3,_G374)
  1 conectado(w3,w2)
  1 valor(w2,_G374)
    2 conectado(w2,w1)
    2 valor(w1,1)
X = 1 ;
    2 valor(w1,_G374)
false.
```

1. Empezamos por como se consulta: en este caso, podemos ver como tenemos un solve(Meta) con un sólo parámetro, lo que queremos demostrar. Por otra parte, para pintar el nivel, necesitamos jugar con un solve(Meta) que aparte de tener el parámetro a demostrar, tenga también el valor de la profundidad solve(Meta, Profundidad).

    1.1 Por ello está la primera regla: el proceso comienza sustituyendo la consulta introducida por un predicado que indice que la meta que hemos puesto es la raíz; el nivel 0.

1.2 Después, unifica con la cuarta regla, pero como valor(w3, Variable) no es un predicado predefinido, falla, y como no ha llegado al corte, hace backtrack, y entra en la regla larga. Esta regla añade profundidad, y utiliza display(), que pinta el nivel tirando de predicados predefinidos, la meta actual y la profundidad en la que nos encontremos (valor que utiliza para calcular cuanto tiene que identar).

No entiendo para que sirve la constante "nl" que está escrita tanto en la cuarta como en la quinta regla

Ejercicio 5 (definición de nuevos operadores).

Modificar el meta intérprete vanilla para que acepte el nuevo lenguaje base introducido en las páginas 19 y 20 de este guión (se refiere a definir & y --->).

```
:-op(40, xfy, &).
:-op(50, xfy, --->).

solve(true):-!.
solve((A & B)) :-!, solve(A), solve(B).
solve(A) :- (B ---> A), solve(B).

light(L)&
ok(L)&
live(L)
     ---> lit(L).

connected_to(W,W1)&
live(W1)
     ---> live(W).

true ---> live(outside).
true ---> light(l1).
true ---> light(l2).
true ---> down(s1).
true ---> up(s2).
true ---> up(s3).
true ---> connected_to(l1,w0).
up(s2) & ok(s2) ---> connected_to(w0,w1).
down(s2) & ok(s2) ---> connected_to(w0,w2).
up(s1) & ok(s1) ---> connected_to(w1,w3).
down(s1) & ok(s1) ---> connected_to(w2,w3).
```

```
true ---> connected_to(l2,w4).
up(s3) & ok(s3) ---> connected_to(w4,w3).
true ---> connected_to(p1,w3).
ok(cb1) ---> connected_to(w3,w5).
true ---> connected_to(p2,w6).
ok(cb2) ---> connected_to(w6,w5).
true ---> connected_to(w5,outside).
true ---> ok(_).
```

Este ejercicio es sencillo. La novedad aquí es el predicado predefinido op(Prioridad, Tipo, Nombre):

1. Se escribe :-op(tal,tal,tal) porque así obligamos a que se ejecute, pero a la vez a que nada unifique con él.
2. El número que se pone hay que ponerlo entre 0 y 1200, e indica la prioridad del operador, es decir, por donde empieza a evaluar si lo juntamos con otros operadores (A ---> A+B*A)
3. Una vez definidos, se concreta que hacen con reglas. En este caso, la segunda regla del programa especifica cómo funciona "&". Y como es equivalente a la ",", es decir, es un AND lógico, lo único que dice es que hay que resolver A y B.
4. La tercera regla especifica como funciona el operador "--->": dado que se define la implicación lógica, y la parte de la izquierda ha de ser siempre "true", nos basta con resolver B para determinar el valor de verdad de la expresión.

Ejercicio 6. (Metas diferidas)

Considerar el siguiente metaintérprete Vanilla:

```
:-op(40, xfy, &).
:-op(50, xfy, --->).
solve(true):-!.
solve((A & B)) :-!, solve(A), solve(B).
solve(A) :- (B ---> A), solve(B).
```

Ampliar el anterior meta intérprete con metas diferidas: dsolve:

- dsolve(Meta, D1, D2) es cierta si D1 es la finalización de D2 y la meta es consecuencia lógica de la base de conocimiento y las metas diferidas D2.
- Adaptar la base de conocimiento del asistente al diagnostico (practica II) declarando como diferibles los átomos ok(_).

Ejemplos de llamadas:

```
2 ?- dsolve(live(p2), [], D).
D = [ok(cb2)] ;
false.
3 ?- dsolve(lit(l2), [ok(cb2)], D).
D = [ok(cb1), ok(s3), ok(l2), ok(cb2)]
```

Solución:

```
:-op(40, xfy, &).
:-op(50, xfy, --->).

dsolve(true,D,D):-!.

dsolve((A & B),D1, D3) :-!, dsolve(A, D1, D2), dsolve(B, D2, D3).

/* La meta A es cierta con las metas diferidas D si A se puede
diferir   */
dsolve(A,D,[A|D]) :- delay(A).

dsolve(A, D1, D2) :- (B ---> A), dsolve(B, D1, D2).
light(L)&
ok(L)&
live(L)
     ---> lit(L).



connected_to(W,W1)&
live(W1)
     ---> live(W).

true ---> live(outside).
true ---> light(l1).
true ---> light(l2).
true ---> down(s1).
true ---> up(s2).
true ---> up(s3).
true ---> connected_to(l1,w0).
```

```
up(s2) & ok(s2) ---> connected_to(w0,w1).
down(s2) & ok(s2) ---> connected_to(w0,w2).
up(s1) & ok(s1) ---> connected_to(w1,w3).
down(s1) & ok(s1) ---> connected_to(w2,w3).
true ---> connected_to(l2,w4).
up(s3) & ok(s3) ---> connected_to(w4,w3).
true ---> connected_to(p1,w3).
ok(cb1) ---> connected_to(w3,w5).
true ---> connected_to(p2,w6).
ok(cb2) ---> connected_to(w6,w5).
true ---> connected_to(w5,outside).
/* true ---> ok(_). */

delay(ok(X)).
```