Práctica II

Prolog II: Meta Intérprete *Vanilla*

Contenido

1. Meta intérpretes.
2. Meta intérprete *vanilla* para cláusulas definidas.
3. Extensión *vanilla* predicados predefinidos.
4. Extensión *vanilla* pruebas.
5. Modificación del lenguaje base.
6. Ejercicios.

1. Meta intérpretes

Meta intérprete

- Intérprete de un lenguaje escrito en el propio lenguaje
- Interés
  - Acceso al proceso de cómputo del lenguaje
  - Desarrollo de entornos de programación integrados

Meta intérprete más sencillo

```
solve(A):- call(A).
```

O bien:

```
solve(A):-A.
```

- Sin interés, pues no permite acceder a los elementos del proceso de cómputo.
- Más interesante: hacer explícita la regla de cómputo y la regla de búsqueda

2. Meta interpreta *vanilla* para cláusulas definidas

Meta intérprete *vanilla* (I)

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

- Lectura Declarativa
  - La meta vacía es cierta.
  - la meta conjuntiva (A, B) es cierta si A es cierta y B es cierta.
  - La meta A es cierta si existe una cláusula A:-B y B es cierta.

Meta intérprete *vanilla* (II)

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

- Lectura Operacional
  - La meta vacía está resuelta.
  - Para resolver la meta (A, B) resolver primero A y después B (Regla de cómputo).
  - Para resolver la meta A, seleccionar una cláusula cuya cabeza unifique con A y resolver el cuerpo, usando la regla de búsqueda de Prolog.

Meta intérprete *vanilla* (III)

Mejor:

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

¿Por qué?

Meta intérprete *vanilla* (IV)

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

- Limitado a Prolog "puro":
  - Sin modificación de la reevaluación: corte, fail, repeat...
  - Sin negación por fallo (programas definidos).
  - Sin asociación de procedimientos: predicados predefinidos.

Ejemplo base de conocimiento "propagación señal "

```
valor(w1, 1).
conectado(w2, w1).
conectado(w3, w2).
valor(W,X):-conectado(W,V), valor(V,X).

1 ?- solve(valor(W,X)).
W = w1,
X = 1 ;
W = w2,
X = 1 ;
W = w3,
X = 1 ;
false.
```

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

3. Extensión *vanilla* predicados predefinidos

Extensión *vanilla* con predicados predefinidos

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

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

Ejemplo predefinidos

```
1 ?- solve(write('¡¡¡Esto funciona!!!')).
¡¡¡Esto funciona!!!
true.
```

4. Extensión *vanilla* pruebas

Extensión “vanilla” pruebas

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

Prueba propagación señal

```
1 ?- solve(valor(w1,X),Prueba).
X = 1,
Prueba = (valor(w1, 1):-true) .

2 ?- solve(valor(w2,X),Prueba).
X = 1,
Prueba = (valor(w2, 1):- (conectado(w2, w1):-true),
   (valor(w1, 1):-true)) .

3 ?- solve(valor(w3,X),Prueba).
X = 1,
Prueba = (valor(w3, 1):- (conectado(w3, w2):-true),
   (valor(w2, 1):- (conectado(w2, w1):-true),
   (valor(w1, 1):-true))) .
```

5. Modificación lenguaje base

- Lenguaje base: expresiones que pueden ser manejadas por el meta intérprete
- Metalenguaje: lenguaje del intérprete
- Hasta ahora, el mismo
  - Cláusulas definidas
  - Predicados predefinidos interpretados "*como*" en Prolog
- Modificaremos el lenguaje base
  - Separar claramente ambos
  - *Sintactic sugaring*

Ejemplo base de conocimiento "propagación señal "

```
true ---> valor(w1, 1).
true ---> conectado(w2, w1).
true ---> conectado(w3, w2).
conectado(W,V) & valor(V,X) ---> valor(W,X).
```

- Necesitamos definir `--->` y `&` como operadores Prolog:

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

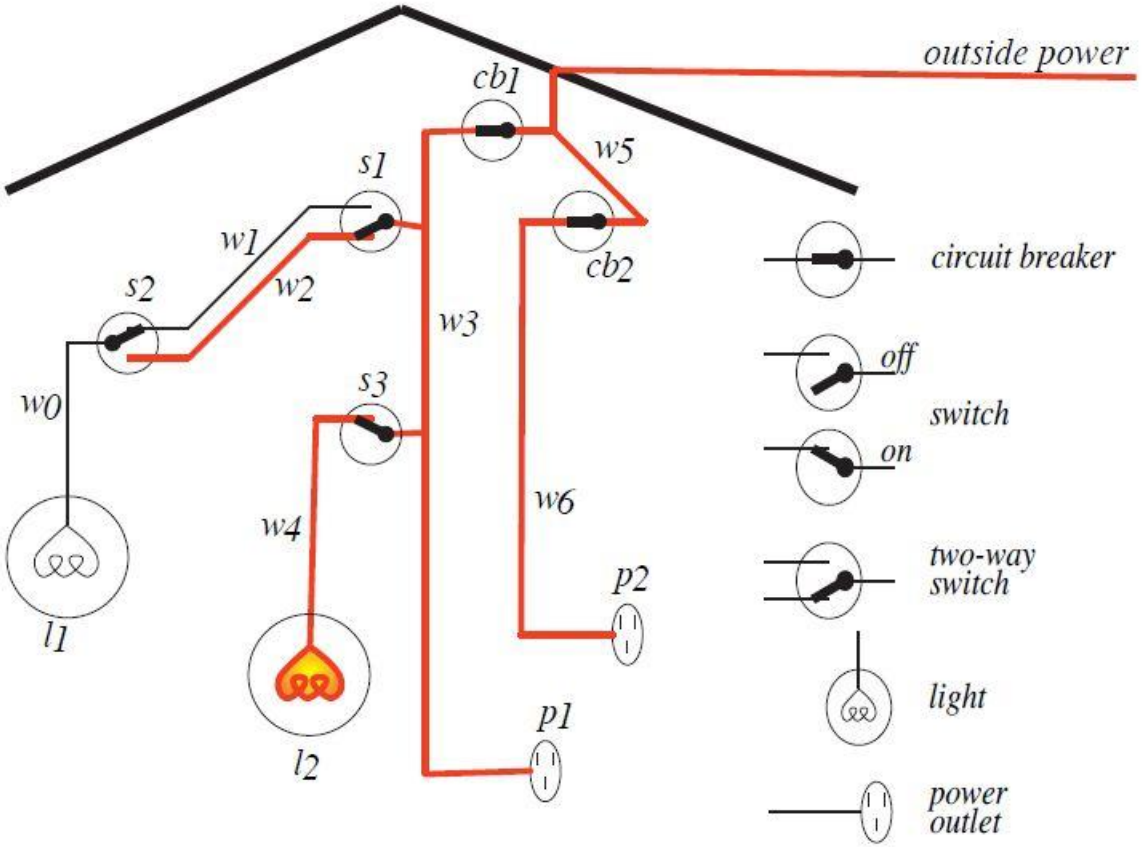
Ligera modificación del meta intérprete

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

solve(true):-!.
solve((A & B)) :-!, solve(A), solve(B).
LA ULTIMA CLÁUSULA
```

- Ver ejercicio 4

Dominio: asistente al diagnóstico



David Poole, Alan Mackworth. *Artificial Intelligence: Foundations of Computational Agents,* Cambridge University Press, 2010

Modelar el dominio en el lenguaje base

- Si una bombilla funciona correctamente y le llega tensión, entonces se enciende:

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

Modelar el dominio

- Si un cable está conectado a otro al que le llega tensión, entonces tiene tensión:

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

Modelar el dominio

- El cable externo tiene tensión:
  `true ---> live(outside).`
- l1 es una bombilla:
  `true ---> light(l1).`
- El interruptor s1 está abierto:
  `true ---> down(s1).`
- El interruptor s2 está cerrado:
  `true ---> up(s2).`

Modelar el dominio

- Si el interruptor s2 está abierto y funciona correctamente entonces el cable w0 está conectado al cable w1:

```
up(s2) & ok(s2) ---> connected_to(w0,w1).
```

- Si el diferencial cb2 funciona correctamente entonces el cable w6 está conectado al cable w5:

```
ok(cb2) ---> connected_to(w6,w5).
```

- El enchufe p2 está conectado al cable w6:

```
true ---> connected_to(p2,w6).
```

Ejercicios (I)

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

2. 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.

3. 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. Por ejemplo:

Ejercicios (II)

```
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.
```

Ejercicios (III)

4. Modificar el meta intérprete *vanilla* para que acepte el nuevo lenguaje base.

5. Completar la base de conocimiento que modela el ejemplo de asistente al diagnóstico propuesto por Poole y Mackworth.

6. Modificar el meta intérprete *vanilla* que genera pruebas para que acepte el nuevo lenguaje base y obtener la prueba de `lit(l2)`.

   (sugerencia: construir la prueba de «`B ---> A`» mediante «`A por Prueba`», siendo `Prueba` la prueba de `B`, definiendo el operador `:-op(40, xfy, por).`)

Bibliografía

- David Poole, Alan Mackworth. *Artificial Intelligence: Foundations of Computational Agents,* Cambridge University Press, 2010.

  Disponible en http://artint.info/html/ArtInt.html