1. Sintaxis de Prolog

Términos:

- **Constantes**: enteros (Ejs: 3, 4), átomos (Ejs: juan, pi) (en minúscula).
- **Variables**: Ejs: X, Casa (en mayúscula)
- **Estructuras**: functor, seguido de uno o más argumentos, es decir, si f es un functor de aridad n, $f(t_1, ..., t_n)$ es un término si t_i es un término $\forall i$. Un término se llama *cerrado* si no posee variables. Observar que no se permiten predicados anidados.

Fórmulas atómicas: De la misma forma que en primer orden, si P es un símbolo de predicado de aridad n, luego $P(t_1, ..., t_n)$ es una fórmula atómica $\forall t_i$.

Tipos de cláusulas:

- **Cláusula de programa**: conforman el programa lógico. Tienen la forma de `L :- B1,...,BS`. Y se interpreta como: L es verdadero si valen B1,... y BS. Si S=0, se dice que la cláusula de programa es un hecho. Se puede pensar el `:-` como una implicación $\leftarrow$.
- **Cláusula objetivo**: tienen la forma de `B1,...,BS` y es lo que queremos probar.

2. Sustitución y Unificación

Llamemos $Term$ al conjunto formado por todos los términos y todos los predicados. Una sustitución en una función $\sigma : Variables \rightarrow Term$ tal que $\sigma(X) \neq X$ para finitos X (i.e. tiene dominio finito).

Notación: si el dominio de σ es $X_1, ..., X_n$, se nota $\sigma = \{X_1 \rightarrow \sigma(X_1), ..., X_n \rightarrow \sigma(X_n)\}$. Por ejemplo, $\sigma = \{X \rightarrow f(a), Y \rightarrow X\}$.

Se puede extender σ a la función $\widehat{\sigma} : Term \rightarrow Term$, de la siguiente manera:

- $\widehat{\sigma}(X) = \sigma(X)$ si X es una variable.
- $\widehat{\sigma}(g(t_1, ..., t_n)) = g(\widehat{\sigma}(t_1), ..., \widehat{\sigma}(t_n))$ si t_i es un término y g un functor o un predicado.
- $\widehat{\sigma}(c) = c$ si c es una constante.

(i.e. sólo interesan las variables).

A partir de ahora vamos anotar $\widehat{\sigma}$ directamente como σ. Por ejemplo, si tomamos la sustitución σ del ejemplo anterior: $\sigma(g(X, Y)) = g(\sigma(X), \sigma(Y)) = g(f(a), X)$.

El problema de *unificación* se define de la siguiente manera: Dado un conjunto de ecuaciones $S = \{s_1 =^? t_1, ..., s_n =^? t_n\}$, donde s_i y t_i son términos, queremos saber si existe una sustitución σ tal que $\sigma(s_i) = \sigma(t_i) \forall 1 \leq i \leq n$. Si esta sustitución existe, se dice que σ es un **unificador** de S.

Tener en cuenta que la igualdad utilizada es puramente sintáctica(por ejemplo $3 \neq 2 + 1$).

Se dice que S es *unificable* si existen una o más sustituciones que cumplan lo dicho anteriormente (unificadores).

Es claro que pueden existir más de un unificador para un conjunto S. Por ejemplo, si $S = \{X =^? Y\}$, los siguientes son unificadores de S: $\sigma_1 = \{X \to Y\}$, $\sigma_2 = \{Y \to X\}$, $\sigma_3 = \{X \to a, Y \to a\}$.

Lo que se intenta buscar es el **unificador más general**(mgu) de S. Una sustitución σ_1 es **más general** que σ_2 si existe un σ_3 tal que $\sigma_2 = \sigma_3(\sigma_1)$. Si ahora volvemos al ejemplo, podemos ver que σ_1 y σ_2 son más generales que σ_3. Además, tanto σ_1 como σ_2 son mgu de S.

El problema de encontrar el mgu de un conjunto S es decidible y existen algoritmos eficientes para encontrarlos.Veamos otros ejemplos de conjuntos de ecuaciones:

- $S = \{a =^? b\}$ no unifica. En general $S = \{t =^? t\}$ unifica.
- $S = \{f(X) =^? g(Y)\}$ no unifica.
- $S = \{f(X) =^? X\}$ no unifica.

3. Regla de Resolución

Dado un programa lógico P (conjunto de cláusulas de programa) y un goal G, se quiere saber si dicho goal es consecuencia lógica de P. La regla de resolución que se utiliza es:

$$\frac{B_1,...,B_i,...,B_n \qquad C{:-}A_1,...,A_p}{\sigma(B_1,...,B_{i-1},A_1,...,A_p,...B_{i+1},...,B_n)} \qquad \text{si } \sigma \text{ es mgu de } B_i \text{ y } C$$

La conclusión de la regla de resolución es el nuevo goal a resolver. Prolog resuelve el goal comenzando desde B_1 de izquierda a derecha en DFS. El orden en el que recorre el programa P (buscando unificar las cabezas de las reglas) es de arriba hacia abajo. Tener en cuenta que el orden de las cláusulas (y subcláusulas) en el programa **influye en el resultado**. Esta es una cuestión mas bien implementativa.

Veamos un ejemplo. Sea el siguiente programa P

```
superHeroe(señorTostada).
esValiente(X) :- superHeroe(X).
```

El goal es `esValiente(señorTostada)`. Apliquemos la regla de resolución:

$$\frac{esValiente(señorTostada) \qquad esValiente(X){:-}superHeroe(X).}{superHeroe(señorTostada)} \qquad \sigma_1 = \{X \to señorTostada\}$$

$$\frac{superHeroe(señorTostada) \qquad superHeroe(señorTostada).}{\square} \qquad \sigma_2 = \{\}$$

Al evaluar un goal, los resultados posibles son los siguientes:

- **Yes**. Se aplicó la regla de resolución hasta alcanzar la cláusula vacía.
- **No**. Se llegó a una cláusula del goal que no unifica con ninguna cabeza de regla.
- **Loop**. El proceso de aplicación de la regla de resolución no termina.

4. Ejemplos

Ejemplo de **Loop**: el programa P es

```
m(X) :- m(X),n(X)
```

Observación: Prolog utiliza unificación, no tipado. Es decir, una variable puede unificar con cualquier valor y siempre pueden compararse dos valores con =.

Un ejemplo más complejo:

```
mediatico(X) :- programaBasura(Y), participa(X,Y).
mediatico(X) :- amigoDeTinelli(X).
programaBasura(cantandoPorUnSueño).
participa(moria, cantandoPorUnSueño).
amigoDeTinelli(sofovich).
```

Goals:

```
mediatico(pettinato) ⇒ No.
mediatico(moria) ⇒ Yes.
mediatico(X) ⇒ X = moria; X = sofovich; No.
```

Veamos que sí importa el orden de las cláusulas con otro ejemplo:

```
descendiente(X,Y) :- hijo(X,Y).
descendiente(X,Y) :- hijo(Z,Y), descendiente(X,Z).
hijo(bart, homero).
hijo(homero, abuelo).
```

Resultado:

```
?- descendiente(bart,W).
W = homero ;
W = abuelo ;
No
```

```
?- descendiente(A,B).
A = bart B = homero ;
A = homero B = abuelo ;
A = bart B = abuelo ;
No
```

Redefinidiendo el programa:

```
descendiente(X,Y) :- descendiente(X,Z), hijo(Z,Y).
descendiente(X,Y) :- hijo(X,Y).
hijo(bart, homero).
```

```
hijo(homero, abuelo).
```

Resultado:

```
?- descendiente(bart,W).
Loop
```

Y si lo modificamos otra vez:

```
descendiente(X,Y) :- hijo(X,Y).
descendiente(X,Y) :- descendiente(X,Z), hijo(Z,Y).
hijo(bart, homero).
hijo(homero, abuelo).
```

Resultado:

```
?- descendiente(bart,W).
W = homero ;
W = abuelo ;
Loop
```

Ejemplos con functores: Vamos a representar los naturales con cero y suc(X). Notar que el comportamiento de los functores se definen a través de los predicados que los utilizan.

```
nn(cero).
nn(suc(X)) :- nn(X).
suma(cero, X, X).
suma(suc(X), Y, suc(Z)) :- suma(X, Y, Z).
```

No se estila en Prolog chequear que los argumentos de la suma sean naturales. Se utiliza el hecho de que Prolog sea no tipado como una ventaja.

5. Reversibilidad e Instanciación

Reversibilidad: una relación no distingue entre input y output, y eso significa que el goal puede "preguntar" en más de una dirección. Por ejemplo, en el predicado *suma*, se puede evaluar `suma(X, cero,suc(cero))`.

Decimos que un parámetro está *instanciado* si es un término cerrado en la sustitución actual.

Por ejemplo si se intenta resolver:

```
mediatico(X) :- programaBasura(Y), participa(X,Y).
```

Y ya se ha resuelto `programaBasura(Y)`, al evaluar el predicado `participa(X,Y)`, la variable Y ya

se encuentra instanciada. Notación: si un predicado $P(..., X, ...)$ supone que el parámetro X está instanciado, se nota como $P(..., +X, ...)$. Si es indistinto, $P(..., -X, ...)$. Esta notación no es sintaxis de Prolog, se utiliza a modo de documentación.

6. Aspectos Extralógicos

Prolog utiliza un motor de operaciones aritméticas que permite realizar operaciones matemáticas por afuera del motor lógico. Esto simplifica el trabajo con dominios conocidos (naturales, racionales, etc).

Decimos que un término es una expresión aritmética si es un número natural, una variable instanciada en una expresión aritmética, o una expresión de la forma $t_1 + t_2$, $t_1 - t_2$, $t_1 * t_2$, etc. Donde cada t_i es una expresión aritmética. Esto significa que todas las variables de una expresión aritmética **se encuentran instanciadas**.

Hay que tener especial cuidado en trabajar con expresiones aritméticas, porque en caso contrario se genera una excepción en Prolog (que no es lo mismo que No).

operador =. `X = Y` responde `Yes` si X unifica con Y (esto no es un aspecto extralógico). Ejemplos: responde `Yes` para `5 = 5`, `a = a`, `X = 2+3`; pero responde `No` para `5 = 2 +3`.

operador is. `X is E`. E siempre debe ser una expresión aritmética. Primero evalúa E y responde `Yes` si X unifica con la evaluación de E. Ejemplos: 5 is 2+3, 2+3 is 5, X is 2+3 (asignación).

predicados relacionales. `<, =:=, =<, >=, =/=`. Ambos argumentos deben ser expresiones aritméticas. Se evalúan ambos argumentos y se compara el resultado de la evaluación.

Ejemplo:

```
factorial(0, 1).
factorial(N, F) :- N > 0, N1 is N-1, factorial(N1, F1), F is N*F1.
```

Ver lo siguiente:

- `factorial(3,X)` es X = 6.
- `factorial(X,2)` da `ERROR`. El predicado no es reversible.
- ¿Qué pasa si quitamos la cláusula `N > 0`?

7. Listas

Las listas se construyen de la siguiente manera:

- `[]`
- `[X, Y, ... | L]`, donde X, e Y son elementos y L es una lista
- `[X, Y, Z, ...]` donde X, Y y Z son elementos de la lista

Ejemplos: `[]`, `[1]`, `[1 | []]`, `[1, 2 | [3]]`, `[1,2,3]`, `[[], 3, soyGalleta]`

```
lista([]).
lista([X|XS]) :- lista(XS).

longitud([], cero).
longitud([X|XS], suc(Z)) :- longitud(XS, Z).

esto esta MAL!
longitud([], cero).
longitud([X|XS], Z) :- longitud(XS, Y), Z is Y+1.

esto esta BIEN!! :)
longitud([], 0).
longitud([X|XS], Z) :- longitud(XS, Y), Z is Y+1.

esto esta MAL!!
longitud([], 0).
longitud([X|XS], Z) :- Y is Z-1, longitud(XS, Y).
```