PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

2. Sintaxis

Grado en Ingeniería Informática

URJC

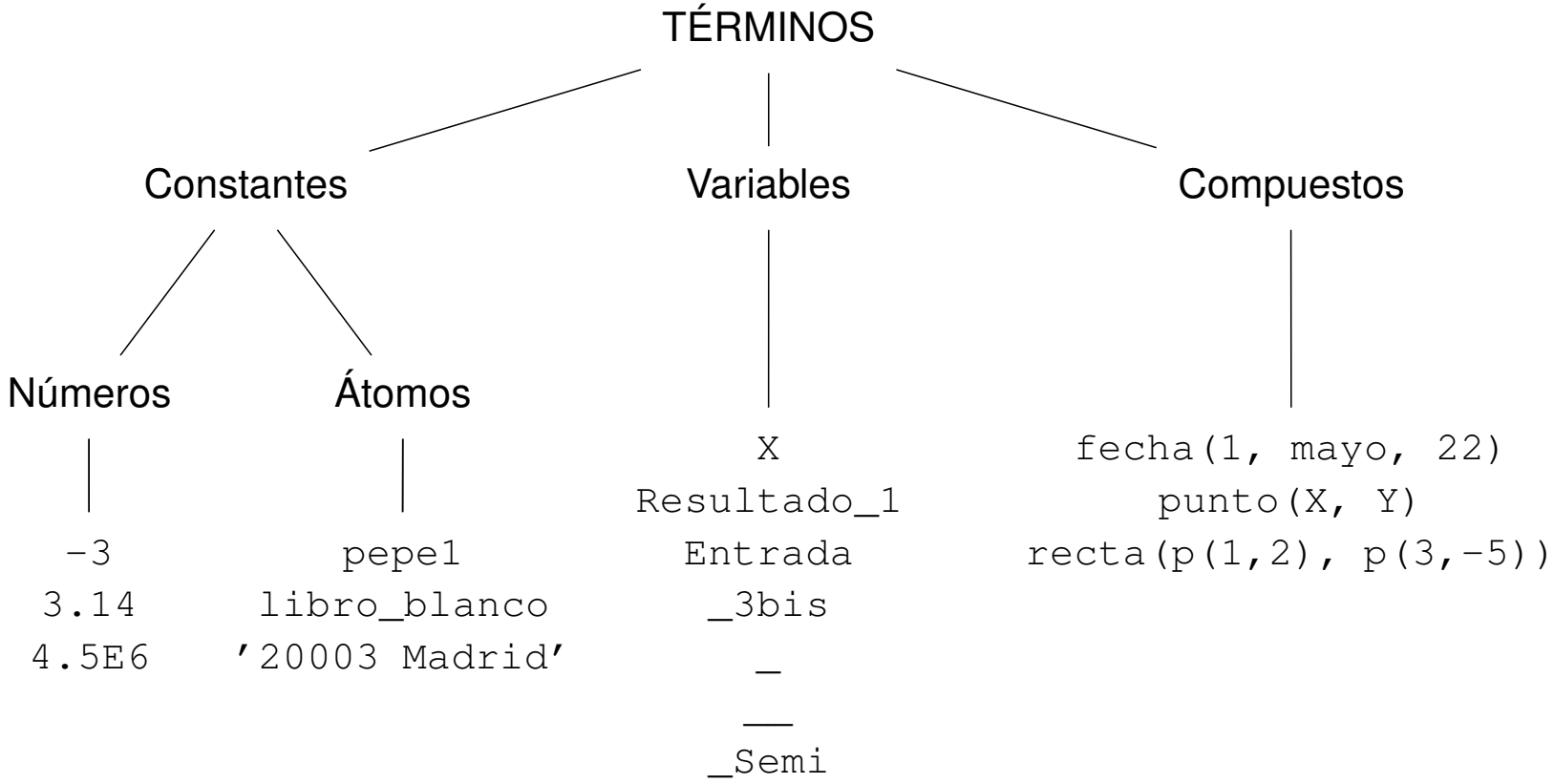
Ana Pradera

Contenido

ELEMENTOS BÁSICOS DE PROLOG

- Términos: sirven para representar *objetos* y constituyen los argumentos de los predicados.
- Predicados: tienen términos como argumentos y describen *propiedades o relaciones* entre sus argumentos.
- Programas: están compuestos por **cláusulas de Horn positivas**, fórmulas lógicas que involucran predicados y por lo tanto establecen propiedades o relaciones entre objetos. Actúan como premisas en los razonamientos.
- Consultas a programas: son fórmulas lógicas existenciales, denominadas **cláusulas de Horn negativas**, que involucran a uno o más predicados de un programa y que actúan como posibles conclusiones en los razonamientos.

TÉRMINOS



Los términos compuestos no admiten espacios entre su nombre y "(" : por ejemplo "punto (X,Y)" produciría un error sintáctico.

- **Números**: notación usual, formato decimal o exponencial.
- **Átomos**:
  - cadenas de letras, dígitos o subrayado, empezando por letra minúscula.
  - o bien cadenas de caracteres entre comillas simples.
- **Variables**: cadenas de letras, dígitos o subrayado, empezando por letra mayúscula o subrayado. La variable _ se denomina *variable anónima* y la variable ___, junto con todas las que empiezan por subrayado seguido de letra mayúscula, son *variables semianónimas*. Las particularidades de estas variables se discuten más adelante.
- **Términos compuestos (estructuras)**: constan de un nombre, que se denomina functor y se representa mediante un *átomo*, seguido, entre paréntesis, por una serie de argumentos, separados por comas, que son, a su vez, *términos* (constantes, variables o compuestos), por lo que los términos son *recursivos*. La aridad de un término compuesto es su número de argumentos.

PREDICADOS

- Tienen la misma sintaxis que los términos compuestos. Todo predicado tiene:
  1. Un nombre o functor, que se representa mediante un *átomo* y se puede sobrecargar (ver ejemplo a continuación).
  2. Cero o más argumentos, entre paréntesis y separados por comas, representados mediante *términos*.
- Establecen propiedades (si tienen menos de dos argumentos) o relaciones entre sus argumentos (cuando hay al menos dos).

Ejemplos

- `llueve` (predicado sin argumentos).
- `animal(serpiente)` (la serpiente es un animal).
- `abuela(pepa, pepon)` (pepa es abuela de pepon).
- `abuela(pepa)` (pepa es abuela). % sobrecarga de `abuela`

Convenios para describir predicados en libros y manuales

- `nombre_predicado/n`, siendo `n` la aridad (número de argumentos) del predicado.
- `nombre_predicado(<uso>Var_1, ...., <uso>Var_n)`, siendo `<uso>`:
  - `<uso> = +` : el argumento correspondiente debe estar, en el momento de usar el predicado, instanciado con un término *no variable* (parámetro de entrada).
  - `<uso> = -` : el argumento correspondiente *no* debe estar instanciado, es decir, debe ser una variable (parámetro de salida).
  - `<uso> = ?` : el argumento puede estar tanto instanciado como no instanciado (parámetro que se puede usar tanto para entrada como para salida).

PROGRAMAS

- Programa PROLOG = conjunto finito de cláusulas de Horn positivas, que pueden ser de dos clases: hechos o reglas.
- Hechos. Los hechos se escriben en PROLOG mediante

```
B.    % acabado en punto
```

donde `B` es un *predicado*, y se corresponden con fórmulas lógicas de la Lógica de Predicados o Lógica de Primer Orden (LPO) de la forma

$$\forall X_1 \ldots \forall X_p(B)$$

siendo $X_1, \ldots, X_p$, $p \geq 0$, las variables que, en caso de que las haya, aparecen en B.

- Reglas. Las reglas se escriben en PROLOG como

```
B :-
    B1,
    ...,
    Bm.     % acabado en punto
```

siendo `B, B1,...,Bm` *predicados*, y se corresponden con fórmulas lógicas de la LPO de la forma

$$\forall X_1 \ldots \forall X_p \Big((B1 \wedge \ldots \wedge Bm) \rightarrow B \Big)$$

donde $X_1, \ldots, X_p$, $p \geq 0$, son las variables que aparecen en `B, B1,..., Bm`, en caso de que las haya.
`B` es la cabeza de la regla y `B1,..., Bm` constituye su cuerpo.
Si `B` $\in$ {`B1,..., Bm`}, se dice que la regla es recursiva.

Significado de hechos y reglas

- Significado de un hecho

```
B.
```

  Para cualesquiera objetos $X_1, \ldots, X_p$ (variables que aparecen en `B`, si las hay), se cumple `B`.

- Significado de una regla

```
B :-
    B1,
    ...,
    Bm.
```

  Para cualesquiera objetos $X_1, \ldots, X_p$ (variables que aparecen en `B, B1,..., Bm`, si las hay), *si* se cumplen `B1` y ... y `Bm`
  ($, = \wedge =$ *y* $=$ conjunción) *entonces*
  ($\rightarrow =$ si...entonces $=$ implicación) también se cumple `B`.

Ejemplo (Programa “Abuelas/os”, hecho en LPO en PL1, 1/2)

```
% progenitor(?X,?Y)
% cierto si X es progenitor/a (padre o madre) de Y
% progenitor se describe mediante 3 HECHOS

    progenitor(pepa, pepito).
    progenitor(pepito, pepita).
    progenitor(pepito, pepon).

% abuelo(?X,?Z): cierto si X es abuela/o de Z
% abuelo/2 se describe mediante la siguiente REGLA

    abuelo(X,Z) :-
        progenitor(X,Y),
        progenitor(Y,Z).
```

Ejemplo (Programa “Abuelas/os” 2/2)

```
% abuelo(?X): cierto si X es abuela/o
% abuelo/1 se describe mediante la siguiente REGLA
    abuelo(X) :-
        abuelo(X,_).  % o abuelo(X,Y).
```

Observación

- Definición de `abuelo/1`: para cualesquiera `X` e `Y`, si se cumple `abuelo(X,Y)` entonces se cumple `abuelo(X)`.
  En LPO: $\forall X \forall Y(abuelo(X, Y) \rightarrow abuelo(X))$.
- La variable `Y` se puede reemplaza por la *variable anónima* _ puesto que su valor no interesa (esto evita además el aviso del intérprete “singleton variable [Y]”).

Ejemplo (Programa “Amistades/Enemistades”)

```
% enemigo(?X,?Y): cierto si X es enemiga/o de Y
% enemigo se describe mediante 4 HECHOS
    enemigo(abel, cain).
    enemigo(cain, blas).
    enemigo(cain, dolores).
    enemigo(blas, abilio).

% amigo(?X,?Y): cierto si X es amiga/o de Y
% amigo se describe mediante 2 REGLAS
    amigo(abilio, X) :-
        amigo(abel, X).

    amigo(X,Y) :-
        enemigo(X,Z),
        enemigo(Z,Y).
```

Ejercicios (Programa “Amistades/Enemistades”)

1. *Describa en Lógica de Primer Orden (LPO) y en lenguaje natural el conocimiento expresado por cada una de las dos reglas que definen el predicado* `amigo`.
2. *Escriba en LPO y en* PROLOG *el conocimiento expresado por las siguientes frases:*
   - *Los que son enemigos de Caín y de Abel también son enemigos de Abilio.*
   - *Dolores es amiga de cualquiera que sea enemigo de Abilio o de Blas.*
   - *Caín es enemigo de todos.*
3. *Describa en LPO y en lenguaje natural el conocimiento que describe el siguiente hecho:*

   ```
   amigo(abilio, X).
   ```

4. *Defina en* PROLOG *el predicado* `tiene_enemigos(?X)`, *cierto si* `X` *tiene algún enemigo.*

Soluciones propuestas:

1.

- La primera regla de `amigo` se corresponde con la fórmula de la LPO

$$\forall X \Big(amigo(abel, X) \rightarrow amigo(abilio, X) \Big)$$

  es decir, para cualquier persona X, si Abel es amigo de X, entonces Abilio también es amigo de X, o bien, equivalentemente, Abilio es amigo de todos los amigos de Abel.

- La segunda regla de `amigo` se corresponde con la fórmula de la LPO

$$\forall X \forall Y \forall Z \Big((enemigo(X, Z) \wedge enemigo(Z, Y)) \rightarrow amigo(X, Y) \Big)$$

  es decir, para cualesquiera personas X, Y, Z, si X es enemigo de Z y a su vez Z es enemigo de Y, entonces X es amigo de Y, o bien, equivalentemente, todo el mundo es amigo de los enemigos de sus enemigos.

2.

- En LPO:

$$\forall X \Big((enemigo(X, cain) \wedge enemigo(X, abel)) \rightarrow enemigo(X, abilio) \Big)$$

La fórmula anterior da lugar a la siguiente regla en PROLOG:
enemigo(X, abilio) :- enemigo(X, cain), enemigo(X, abel).

- En LPO:

$$\forall X \Big((enemigo(X, abilio) \vee enemigo(X, blas)) \rightarrow amigo(dolores, X) \Big)$$

Lo anterior es lógicamente equivalente a

$$\begin{aligned} & \forall X \Big((enemigo(X, abilio) \rightarrow amigo(dolores, X) \Big) \\ \wedge \quad & \forall X \Big((enemigo(X, blas) \rightarrow amigo(dolores, X) \Big) \end{aligned}$$

que da lugar a las dos siguientes reglas en PROLOG:
amigo(dolores, X) :- enemigo(X, abilio).
amigo(dolores, X) :- enemigo(X, blas).

- En LPO

$$\forall X \; enemigo(cain, X)$$

fórmula a la que le corresponde el hecho en PROLOG
enemigo(cain, X).

3. El hecho *amigo*(*abilio*, *X*) se corresponde con la fórmula lógica $\forall X$ *amigo*(*abilio*, *X*), es decir, Abilio es amigo de todo el mundo. Observe que aunque el hecho *amigo*(*abilio*, *X*) es correcto, la variable *X* solo aparece una vez, por lo que, para evitar avisos del intérprete (*singleton variables: [X]*), la variable *X* se podría reemplazar por la variable anónima _, dando lugar al hecho *amigo*(*abilio*, _).

4. Dada una persona cualquiera *X*, la propiedad *tiene_enemigos*(*X*) se cumplirá si existe al menos otra persona, pongamos *Y*, tal que se cumple la relación *enemigo*(*Y*, *X*). Por lo tanto, para cualesquiera personas *X* e *Y*, si resulta *enemigo*(*Y*, *X*), entonces se puede concluir *tiene_enemigos*(*X*). Lo anterior, escrito en LPO, da lugar a la fórmula

$$\forall X \forall Y \ (\textit{enemigo}(Y, X) \rightarrow \textit{tiene\_enemigos}(X))$$

La fórmula anterior se corresponde con la siguiente regla en PROLOG:

```
tiene_enemigos(X) :- enemigo(Y, X).
```

o bien, para evitar avisos del intérprete (*singleton variables: [Y]*), la *Y* se puede reemplazar por la variable anónima _ puesto que no juega ningún papel:

```
tiene_enemigos(X) :- enemigo(_, X).
```

Ejemplo (Programa en lógica “pura” para sumar naturales)

- Un ejercicio clásico para ilustrar el uso y la potencia de PROLOG es la implementación de un programa para *sumar números naturales* con lógica “pura”, es decir, sin usar las facilidades aritméticas de PROLOG (que se estudian más adelante).
- La suma es una *función* matemática que recibe como entrada dos naturales y devuelve su suma. Para implementar una función en PROLOG, que solo dispone de predicados (relaciones), basta con definir un predicado con los mismos argumentos que la función pero añadiendo un argumento adicional para el resultado (colocado, por convenio, al final). En este caso:

```
suma(?X,?Y,?Z)
cierto si Z es la suma de X e Y
```

- Para trabajar en lógica “pura” se necesita un mecanismo para representar números naturales. Uno sencillo consiste en usar un símbolo, por ejemplo `0`, para denotar el número natural cero, y otro, por ejemplo `s`, para construir el término compuesto `s(X)` representando el sucesor del número `X`, de forma que s(0) representa el número 1, s(s(0)) representa el 2, etc.
- Por otro lado, dado que PROLOG trabaja con recursividad, hay que partir de una definición *recursiva* de la suma:

$$\text{Para cualquier } X \in \mathbb{N}, \quad X + 0 = X \tag{1}$$

$$\text{Para cualesquiera } X, Y \in \mathbb{N}, \quad X + (Y + 1) = (X + Y) + 1 \tag{2}$$

(1) establece que la suma de cualquier número con cero es él mismo, mientras que (2) asegura que si se conoce la suma de dos números, $X + Y$, entonces también se conoce la suma del primero con el sucesor del segundo, $X + (Y + 1)$, que no es otra cosa que el sucesor del valor $X + Y$, es decir, $(X + Y) + 1$.

- Lo anterior, utilizando el predicado y los símbolos definidos previamente, se expresa mediante las siguientes fórmulas:

$$\forall X \; suma(X, 0, X) \tag{1}$$

$$\forall X \, \forall Y \forall Z \Big(suma(X, Y, Z) \rightarrow suma(X, s(Y), s(Z)) \Big) \tag{2}$$

En PROLOG:

```
% suma(?X,?Y,?Z): cierto si Z es la suma de X e Y

% (1) Caso base (expresado mediante un hecho)
      suma(X,0,X).

% (2) Caso recursivo (expresado mediante una regla)
      suma(X,s(Y),s(Z)) :-
          suma(X,Y,Z).
```

Ejercicios (Programación en lógica “pura”)

1. *Suponga que la primera cláusula de* `suma` *fuese* `suma(0,X,X).` *en lugar de la dada. ¿Cómo sería el resto del programa?*
2. *Escriba sendos programas lógicos “puros” capaces de decidir si un número natural es par (impar). Es decir, implemente los predicados* `par(?X)` *e* `impar(?X)`*, ciertos si* `X` *es par (impar) sin usar aritmética y manejando los números naturales mediante la constante cero y la función sucesor* `s(X)` *como se ha hecho para la suma.*
3. *Use el predicado* `suma/3` *para definir mediante lógica “pura” el predicado sobre números naturales* `producto(?X,?Y,?Z)`*, cierto si* `Z` *es el producto de* `X` *por* `Y`*. Parta de la siguiente definición recursiva del producto:*

$$\text{Para cualquier } X \in \mathbb{N}, \quad X \times 0 = 0$$

$$\text{Para cualesquiera } X, Y \in \mathbb{N}, \quad X \times (Y + 1) = (X \times Y) + X$$

Soluciones propuestas:

1. El cambio propuesto hace que la recursión se lleve a cabo en el primer argumento en lugar de en el segundo como se hace en el código original. Este cambio no tiene mayor importancia, dado que la suma es conmutativa, siempre que se adapte la segunda cláusula de la siguiente forma:

```
suma(s(X),Y,s(Z)) :- suma(X,Y,Z).
```

2.

```
par(0). % 0 es par
 % para todo X, si X es par, entonces s(s(X)) también lo es.
par(s(s(X))) :- par(X).

impar(s(0)). % 1 es impar
 % para todo X, si X es impar, entonces s(s(X)) también lo es.
impar(s(s(X))) :- impar(X).
```

Una implementación alternativa para `impar`, basada en el predicado `par`:

```
 % para todo X, si X es par, entonces s(X) es impar.
impar(s(X)) :- par(X).
```

3.

```
 % el producto de cualquier natural por 0 es 0.
producto(_, 0, 0).
 % si se suma X al producto de X por Y, el resultado es X por s(Y)
producto(X, s(Y), Z) :- producto(X, Y, ProdXY), suma(ProdXY, X, Z).
```

CONSULTAS

Consultas para la activación de programas

Las consultas = cláusulas de Horn negativas = cláusulas objetivo = cláusulas meta tienen la siguiente sintaxis:

```
?- A1, ... ,An.
```

---

siendo `A1,...,An` *predicados*, y se corresponden con fórmulas lógicas de la forma

$$\exists X_1 \ldots \exists X_q (A1 \wedge \ldots \wedge An)$$

siendo $X_1, \ldots, X_q$, $q \geq 0$, las variables que, en caso de que las haya, aparecen en `A1,...,An`. Representan la pregunta *¿Es cierto que existen objetos* $X_1, \ldots, X_q$ *que cumplen A1 y ... y An? En caso afirmativo, ¿qué objetos son?* ($, = \wedge = y =$ conjunción)

Ejemplos (Algunas consultas al programa “Abuelas/os”)

- `?- abuelo(pepa, pepito).` *False*
- `?- abuelo(pepa, pepita).` *True*
- `?- abuelo(pepa, N).` 2 soluciones: *N=pepita* y *N=pepon*
- `?- abuelo(pepa, _).` *True (variable anónima, no se computa)*
- `?- abuelo(A, pepon).` *A=pepa*
- `?- abuelo(A, N).` 2 soluciones: [*A=pepa,N=pepita*] y [*A=pepa,N=pepon*]
- `?- abuelo(X).` *X=pepa*
- `?- progenitor(A, P), progenitor(P, pepita).`
  A=pepa, P=pepito
- `?- progenitor(X,pepito), progenitor(X,pepon).` *False*
- `?- progenitor(X, pepon), progenitor(X, pepita).`
  X=pepito
- `?- abuelo(A, pepon), progenitor(P, pepon).`
  A=pepa, P=pepito

Observación (Uso de variables anónimas/semianónimas)

Ojo con el uso de la variable **anónima** (_) y las **semianónimas** (___ o cualquiera que empiece por _ seguido de una mayúscula):

- `?- progenitor(pepa, P), progenitor(P, Z).` (v. normal)

  2 soluciones: [`P=pepito`, `Z=pepita`] y [`P=pepito`, `Z=pepon`]

- `?- progenitor(pepa, _P), progenitor(_P, Z).` (v. semi)

  La variable semianónima `_P` no se reporta pero sí tiene que unificar ("casar") consigo misma:

  2 soluciones: `Z=pepita` y `Z=pepon` (`_P = pepito` pero no se incluye)

- `?- progenitor(pepa, _), progenitor(_, Z).` (v. anónima)

  La variable anónima ni reporta valor ni tiene por qué unificar ("casar") en sus distintas apariciones, por lo que, dado que `progenitor(pepa, _)` se cumple para alguien, `Z` será cualquiera para el que se cumpla `progenitor(_, Z)`:

  3 soluciones: `Z=pepito`, `Z=pepita` y `Z=pepon`

Ejemplos (Algunas consultas al programa “suma”)

- `?- suma(0, s(0), s(0)).` *True*
- `?- suma(s(0), s(0), X).` X = s(s(0))
- `?- suma(s(0), s(0), X), suma(X, s(s(0)), Z).`

  X = s(s(0)), Z = s(s(s(s(0))))

- `?- suma(0,0,Z), suma(0,s(s(0)),Z).` *False*
- `?- suma(s(s(0)), Y, s(0)).` *False*
- `?- suma(s(0), _, s(s(s(0)))).` *True*
- `?- suma(s(0), Y, s(s(s(0)))).` Y = s(s(0))

versatilidad de PROLOG: ¡“suma” también sirve para restar!

- `?- suma(X, Y, s(s(0))).`

  3 soluciones: [X = s(s(0)),Y = 0], [X = s(0),Y = s(0)] y [X = 0, Y = s(s(0))]

versatilidad de PROLOG: ¡“suma” descompone en sumandos!

Ejercicios (Consultas al programa “Amistades/Enemistades”)

1. *Exprese en LPO y en lenguaje natural qué se pretende averiguar con las siguientes consultas en* PROLOG*:*
   - `?- amigo(abel, X).`
   - `?- amigo(abel, _).`
   - `?- enemigo(X, X).`
   - `?- amigo(dolores, X), enemigo(X, dolores).`
   - `?- amigo(X,_Y), amigo(X,_Z), enemigo(_Y,_Z).`
2. *Escriba en Lógica de Primer Orden y en* PROLOG *las siguientes preguntas:*
   - *¿Tienen Abel y Caín algún amigo común? En caso afirmativo, ¿quién o quiénes?*
   - *¿Es Caín amigo de alguno de los enemigos de Abel? En caso afirmativo, ¿quién o quiénes son?*
   - *¿Tiene Abel enemigos?*
   - *¿Qué parejas de personas existen tales que cada una de ellas es amiga de la otra?*

Soluciones propuestas:

1.

- LPO: $\exists X\ amigo(abel, X)$. ¿Es Abel amigo de alguien? En caso afirmativo, ¿de quién o quiénes?
- LPO: $\exists \_\ amigo(abel, \_)$. ¿Es Abel amigo de alguien? En este caso el uso de la variable anónima hace que se espere solo una respuesta booleana.
- LPO: $\exists X\ enemigo(X, X)$. ¿Hay alguien que sea enemigo de sí mismo? En caso afirmativo, ¿quién o quiénes?
- LPO: $\exists X\ (amigo(dolores, X) \wedge enemigo(X, dolores))$. ¿Es Dolores amiga de alguno de sus enemigos? En caso afirmativo, ¿de quién o quiénes?
- LPO: $\exists X \exists \_Y \exists \_Z\ (amigo(X, \_Y) \wedge amigo(X, \_Z) \wedge enemigo(\_Y, \_Z))$. ¿Hay alguien que tenga al menos dos amigos que son enemigos entre sí? En caso afirmativo, ¿quién o quiénes? (las variables semianónimas no se computarán).

2.

- LPO: $\exists A\ (amigo(abel, A) \wedge amigo(cain, A))$
  PROLOG: `?- amigo(abel, A), amigo(cain,A).`
- LPO: $\exists X\ (amigo(cain, X) \wedge enemigo(X, abel))$
  PROLOG: `?- amigo(cain, X), enemigo(X, abel).`
- LPO: $\exists \_\ enemigo(\_, abel)$
  PROLOG: `?- enemigo(_, abel).`
- LPO: $\exists A \exists B\ (amigo(A, B) \wedge amigo(B, A))$
  PROLOG: `?- amigo(A,B), amigo(B,A).`

RESUMEN

	REGLAS	HECHOS
PROLOG	`B :- B1,...,Bm.`	`B.`
LPO	$\forall X_1 \ldots \forall X_p((B_1 \wedge \ldots \wedge B_m) \rightarrow B)$	$\forall X_1 \ldots \forall X_p(B)$
Uso	en un programa	en un programa

	OBJETIVOS (METAS)
PROLOG	`?- A1,...,An.`
LPO	$\exists X_1 \ldots \exists X_q(A_1 \wedge \ldots \wedge A_n)$
Uso	consulta

- `B`, `Bi` y `Aj` representan *predicados* y X_k *variables*.
- Hechos, reglas y consultas *terminan siempre con un punto*.
- Predicados: *empiezan por minúscula*.
- Variables: *empiezan por mayúscula o subrayado*.

Interpretación procedural de PROLOG

1. Predicados = *procedimientos*, con argumentos = *parámetros*.
2. Reglas = *definiciones de procedimientos*:

   Una regla

   ```
   B :- B1, B2,..., Bm.
   ```

   se puede interpretar de la siguiente forma: computar el procedimiento `B` equivale a computar en primer lugar el procedimiento `B1`, a continuación el procedimiento `B2`, etc hasta `Bm`.
3. Consultas = *llamadas a procedimientos*:

   Una consulta

   ```
   ?- A1, A2,..., An.
   ```

   se puede interpretar como una llamada al procedimiento `A1`, seguida de una llamada al procedimiento `A2`, etc hasta `An`.

Ejercicios (Sintaxis de PROLOG)

- *Ejercicio nº 1 de la Práctica de PROLOG nº 1 (puede consultar las instrucciones generales para la realización de las prácticas aquí).*
- *Ejercicio nº 2 de la Práctica de PROLOG nº 1.*

Las soluciones propuestas para ambos ejercicios, comentadas, están disponibles en Práctica de PROLOG nº 1 con soluciones.

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq