Introducción al paradigma lógico

Vamos a empezar a hablar de un nuevo paradigma, el paradigma lógico. Recordemos que los paradigmas nos dan un marco conceptual para resolver problemas; en el caso de la programación, es para poder crear programas.

Entonces empezamos haciéndonos la pregunta de siempre: *¿Qué es un programa?* Revisemos la definición…
"Es un ingenio que resuelve nuestro problema". Vimos que hay dos formas de encararlo:

- **Imperativo/procedimental**: yo le digo la secuencia de pasos a ejecutar y voy guardando estados intermedios en posiciones de memoria que llamo variables. Esto respeta fielmente el modelo matemático de Von Neumann.
- **Declarativo**: no hay algoritmo. Sólo definiciones (el qué) y hay alguna magia detrás de todo esto que lo termina resolviendo. Si esa magia es transparente ➔ me puedo concentrar en lo que es esencial al problema (ese es el beneficio de la abstracción).

Veamos entonces cómo podemos encuadrar la resolución de problemas en una perspectiva de Lógica. En el paradigma lógico, en general usamos soluciones declarativas.
¿Para qué se inventó la Lógica? Para poder hacer razonamientos correctos sin necesidad de saber de lo que estás hablando (casi un precursor de la televisión).

¿Qué declaro? Lo que hago a continuación es declarar **conocimiento** a través de predicados (afirmaciones que hago respecto a algo).
Tenemos un silogismo típico de los primeros cursos de Lógica Proposicional

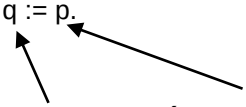
Todo hombre es mortal. (cuantificación universal)
Sócrates es humano (cuantificación particular)
Ergo, Sócrates es mortal. (cuantificación particular)

Esto en lógica, se escribe:
∀(x)(humano(x)=>mortal(x))
humano(socrates)
entonces mortal (socrates)

Esto va a formar parte de nuestra base de conocimientos de PROLOG:

`mortal(X):-humano(X).`	Regla
`humano(socrates).`	Hecho

Fíjense que lo que en lógica se escribe p => q, en sintaxis PROLOG se escribe al revés:
q := p.



consecuente ← antecedente(s). Si se cumplen los antecedentes, entonces se cumple el consecuente.
Esta forma de escribir los predicados se llama *cláusula de Horn*.

¿Qué preguntas puedo hacer?

- Sócrates, ¿es mortal?

```
?- mortal(socrates).
Yes
```

- Maradona, ¿es mortal?

```
?- mortal(maradona).
false.
```

- ¿Qué mortales conocés?

```
?- mortal(X).
X = socrates
true.
```

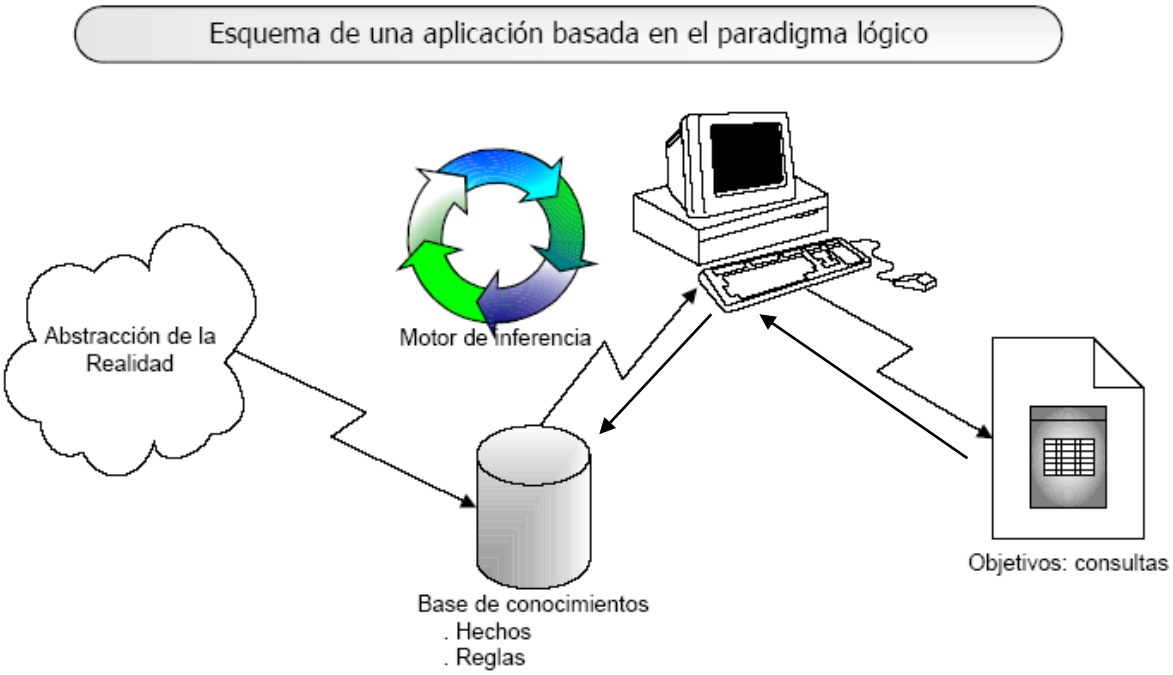
- ¿Hay algún mortal en la sala?

```
?- mortal(_).
true.
```

Ahora, ¿dónde hago las preguntas y dónde pongo el conocimiento?

Modelo de una aplicación en Lógico

Dibujamos en el pizarrón este gráfico:



¿Dónde escribo la base de conocimientos? En un archivo .pl[1]

Además de un lugar donde guardar los conocimientos, en un lenguaje lógico yo puedo efectuar consultas (preguntas, objetivos). Esas consultas pueden involucrar variables o no.

¿Qué es una variable? Dentro del paradigma lógico una variable es una incógnita, algo que está sin resolver. Es el caso de mortal(X). X comienza con mayúscula, lo que me indica que no es un valor fijo. Podría hacer
`mortal(Quien)` si lo quisiera hacer más legible.

¿Quién termina resolviendo que Sócrates es mortal? Un **motor de inferencia** que trabaja con algún tipo de algoritmo mágico (ese algoritmo tiene un nombre y es el Principio de Resolución de Robinson, para que le pregunten a Google si están interesados). Durante la cursada vamos a evitar saber qué pasa ahí abajo, de la

[1] uso cualquier editor de texto, fíjense en el swiki que les da un par de tips:
http://www.pdep.com.ar Software > SWI Prolog

misma manera que no intentamos saber cómo se terminan reduciendo las expresiones en funcional. Bien, acá en Lógico lo único que necesitamos saber es que si yo le hago una consulta
`mortal(Quien)` y en la base de conocimientos además de Sócrates tengo a Platón, entonces la consulta me responderá:
Quien = socrates ;
Quien = platon.
Recordamos: Platon con mayúscula → es una variable (una incógnita)
platon con minúscula → es un valor (un individuo)

¿Cómo encuentra las soluciones? No nos interesa. Sólo vamos a decir que con un mecanismo que se llama backtracking, así es como encuentra todas las unificaciones posibles para una variable.
Unificación de una variable = ligar un valor a esa incógnita
X = socrates, X es la incógnita, socrates el valor. Vamos a volver con este concepto en breve.

Dos cosas:

- La declaratividad: no vamos a tener un algoritmo, sino que va a estar implícito en un motor (cuando yo hago las consultas no tengo que programar la lógica que encuentra el conjunto de soluciones posibles).
- No existe la asignación / no hay efecto colateral: una variable no es una posición de memoria que almacena estados intermedios. Una variable es una incógnita, no tiene sentido que yo le haga
  X = X + 1, porque esa condición nunca se puede cumplir (es una contradicción en sí misma como regla lógica).

Bien. Dentro de la base de conocimientos ya hemos visto dos tipos de cláusulas:

- Los hechos, que son axiomas (son ciertos porque existen, si no existen se presumen falsos)
- Las reglas (p => q, que se escriben con la forma clausal de Horn como q :- p).

Variable anónima

en la cuarta consulta, yo pregunté si alguien era mortal. La traducción para la consulta lógica:
∃x / p(x) (existe x tal que p de x se cumple, donde p(x) significa que x es mortal)
se traduce a
mortal(_).

Efecto colateral

El efecto colateral[2] (side effect) es una modificación que sobrevive al bloque de código que la genera. Ojo, **no es lo mismo que asignación destructiva**, que es la simple modificación del valor contenido en una variable.

Ejemplo:

```
VAR CANTIDAD_VOCALES = 0;
FUNCTION CONTAR_VOCALES(STRING PALABRA) : INTEGER
BEGIN
      RECORRER I DE 1 A LONGITUD(PALABRA)
      BEGIN
            IF (PALABRA(I) ES VOCAL)
            BEGIN
                  CANTIDAD_VOCALES = CANTIDAD_VOCALES + 1
            END
      END
```

[2] A veces también llamado “efecto de lado” en viejas cursadas

```
      ↑CANTIDAD_VOCALES
END
```

Aquí vemos que hay una asignación destructiva de la variable cantidad_vocales (se pisan sucesivamente los valores), y la misma afecta al resultado de la función, ya que el alcance al terminar la evaluación de contar_vocales el valor de esta variable permanece modificado. En este ejemplo, **existe un efecto colateral** en la función.

En cambio, si nuestra definición de cantidad_vocales es local:

```
FUNCTION CONTAR_VOCALES(STRING PALABRA) : INTEGER
BEGIN
      VAR CANTIDAD_VOCALES = 0;
      RECORRER I DE 1 A LONGITUD(PALABRA)
      ...
```

Entonces una vez terminada la evaluación de contar_vocales, la variable cantidad_vocales desaparece. Las modificaciones realizadas sobre la misma no permanecen en el programa. En este ejemplo, **no hay efecto colateral**.

El paradigma lógico no tiene efecto colateral, no permite asignar en más de un contexto el valor de una variable.

Principio de Universo cerrado

Cuando preguntamos si Maradona era mortal, la respuesta a esa consulta fue no.
¿Significa eso que Maradona no es mortal… o que Prolog no pudo determinar que fuera mortal?
Bueno, eso es lo que se llama:

Principio de Universo Cerrado o Negación por falla (Closed World assumption): todo lo que no está en la base de conocimientos no se puede inferir si es cierto o no, por lo tanto se asume falso.

Entonces recordemos: si para una consulta no tengo conocimiento suficiente, ¿qué voy a hacer?: asumir que lo que estoy preguntando no se cumple, es falso.

Un ejemplo:

```
culpable(X):-persona(X), asesino(X).
```

(toda persona que es asesina es culpable)
(si es persona **y** es asesino ➔ es culpable)

```
persona(barreda).
persona(fernando).
persona(leo).
asesino(barreda).
```

Cuando yo pregunte: culpable(leo), me va a decir que no. “Toda persona es inocente hasta que se demuestre lo contrario”. Claro, que ud. no sabrán si Leo es el asesino hasta que termine la cursada. Bueno, por ahí un poquito antes[3]…

[3] Leo puede reemplazarse por su docente de Paradigmas preferido…

Atributos y relaciones

Tanto los hechos como las reglas son predicados. Un predicado puede tener:
1 sólo argumento (monádico): representa una propiedad o una cualidad.

```
gracioso(krusty).
animal(abeja).
```

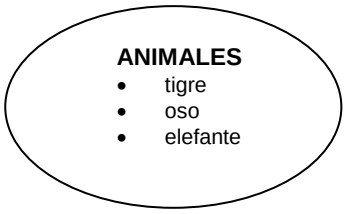
Varios argumentos (poliádicos): representan relaciones.

```
gusta(juan, morocha).
ingrediente(pollo, 1).
```

Definición por extensión y por comprensión

Un conjunto de hechos para el mismo predicado forman la definición por extensión del predicado. **Ejemplo:**

```
animal(tigre).
animal(oso).
animal(elefante).
```



Es como decir Animales = { tigre, oso, elefante }

```
planeta(venus).
planeta(tierra).
```

¿Cuál es la desventaja de este tipo de definición? Requiere la enumeración de todos los elementos que componen el conjunto. Para conjuntos con muchos elementos es cuanto menos incómodo.

Otra desventaja que presenta es que no brinda información referente al conjunto más allá de la que el observador le puede dar.

Una regla forma la definición por comprensión del predicado.

a) $A = \{a \in \mathbf{N} \mid 2 < a < 6\}$

b) $B = \{p \in \mathbf{N} \mid p < 10, \text{ siendo } p \text{ par}\}$

Si lo definimos en un lenguaje lógico como PROLOG:

```
A(X):- X > 2, X < 6.
B(X):- X < 10, par(X).
```

Disyunciones (OR)

"Un grosso es un tipo que es físico nuclear, o sale con una modelo, o ayuda a los pobres".

```
grosso(X):-fisicoNuclear (X).
grosso(X):-saleCon(X, Mujer), modelo(Mujer).
```

(recordamos predicados poliádicos que expresan relaciones)

```
grosso(X):-ayudaA(X, Y), pobre(Y).
```

Esta definición tiene de bueno que me permite definir una relación en función de otras; pero tiene de malo que no me deja saber en función de cuál fue. En principio si sólo me importa la grositud de una persona, es una buena regla. La decisión de qué tan importante es tener el detalle de grositud es la que va a determinar cómo modelo mi base de conocimientos. Para dar un ejemplo: fíjense que una persona que ayuda a los pobres puede definirse como

```
ayudaA(X, Y), pobre(Y) o
ayudaAPobres(X).
```

Nosotros definimos los individuos que componen cada predicado y en qué forma se relacionan entre sí; también nosotros le damos el sentido a los predicados: ayudaA(X, Y) implica que X es el ayudador e Y el ayudado por convención nuestra, pero podríamos haberlo definido al revés.

¿Dónde aplicamos la lógica en computación?

- Bases de datos: lenguajes de restricciones, lenguajes de consulta (SQL)
- Inteligencia artificial: representación del conocimiento, deducción.
- Ingeniería de software: especificación de sistemas (lenguajes Z)
- Criptografía: verificación de protocolos criptográficos
- Procesamiento del lenguaje natural

Pero de esto vamos a hablar sobre el final…