

Técnicas de representación y razonamiento

❑ Tema 3: Representación del conocimiento e inferencia

❑ 3.2: Lógica y Prolog – Índice de contenidos

❑ Lógica de predicados (Leng. Formal Representación)

- ❑ Sintaxis, semántica, propiedades
- ❑ ¿Cómo usar la lógica para representar conocimiento?
- ❑ Mecanismos de inferencia: deducción, resolución
- ❑ Problemas de la representación con LPO

❑ Representación de conocimiento con Prolog (Implementación)

- ❑ Conceptos básicos
- ❑ Representación de conocimiento factual con Prolog
- ❑ Consultas, backtracking, negación, reglas
- ❑ Relaciones transitivas
- ❑ Herencia de propiedades
- ❑ Relaciones simétricas
- ❑ Listas

Técnicas de representación y razonamiento

❑ Técnicas de representación del conocimiento

❑ Representaciones básicas

❑ Lógica de predicados. Representación en Prolog

- ❑ Redes semánticas
- ❑ Sistemas de producción

❑ Representaciones estructuradas

❑ Marcos (frames) y guiones (scripts)

❑ Estudio comparativo de las técnicas de representación

❑ Lenguajes de representación del conocimiento

Técnicas de representación y razonamiento

❑ Diversos formalismos para construir bases de conocimiento

❑ Representaciones basadas en relaciones

❑ Lógica

❑ Redes semánticas

❑ Representaciones basadas en objetos

❑ Marcos

❑ Objeto-Atributo-Valor

❑ Representaciones basadas en acciones

❑ Sistemas de producción

❑ Guiones

❑ Combinaciones y modificaciones de los anteriores

¿Qué es una lógica?

❑ Una lógica es un lenguaje formal

❑ Tiene una **sintaxis** que determina qué expresiones son legales (*la forma*)

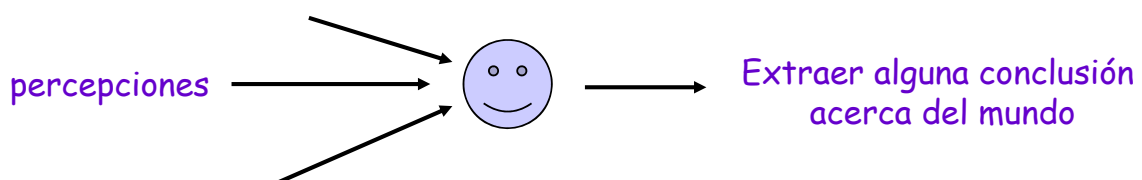
❑ También cuenta con una **semántica** que determina qué representan las expresiones legales (*el contenido*)

❑ Y suele disponer de un **sistema de inferencia** que permite manipular expresiones sintácticas para obtener otras expresiones sintácticas

❑ ¿Con qué propósito?

❑ Obtener expresiones con un significado “interesante”

❑ Que nos digan algo “nuevo” del mundo

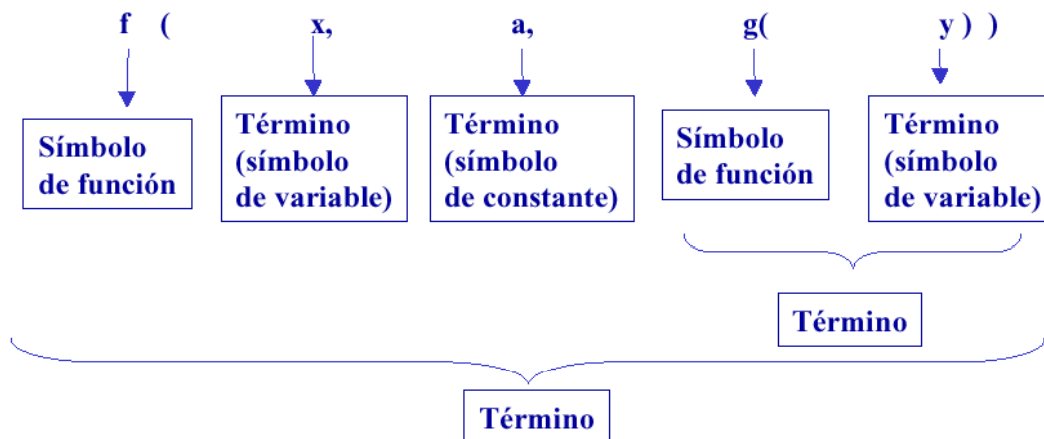


Lógica de predicados: sintaxis

□ Términos

- Un símbolo de **constante** es un término ($a, b, c...$)
- Un símbolo de **variable** es un término ($x, y, z...$)
- Si f es un símbolo de función (o functor) de aridad n , y $t_1, t_2, ..., t_n$ son términos, entonces $f(t_1, t_2, ..., t_n)$ es un término **compuesto**

□ Ejemplo: $f(x, a, g(y))$



Lógica de predicados: sintaxis

□ Fórmulas

ATÓMICAS

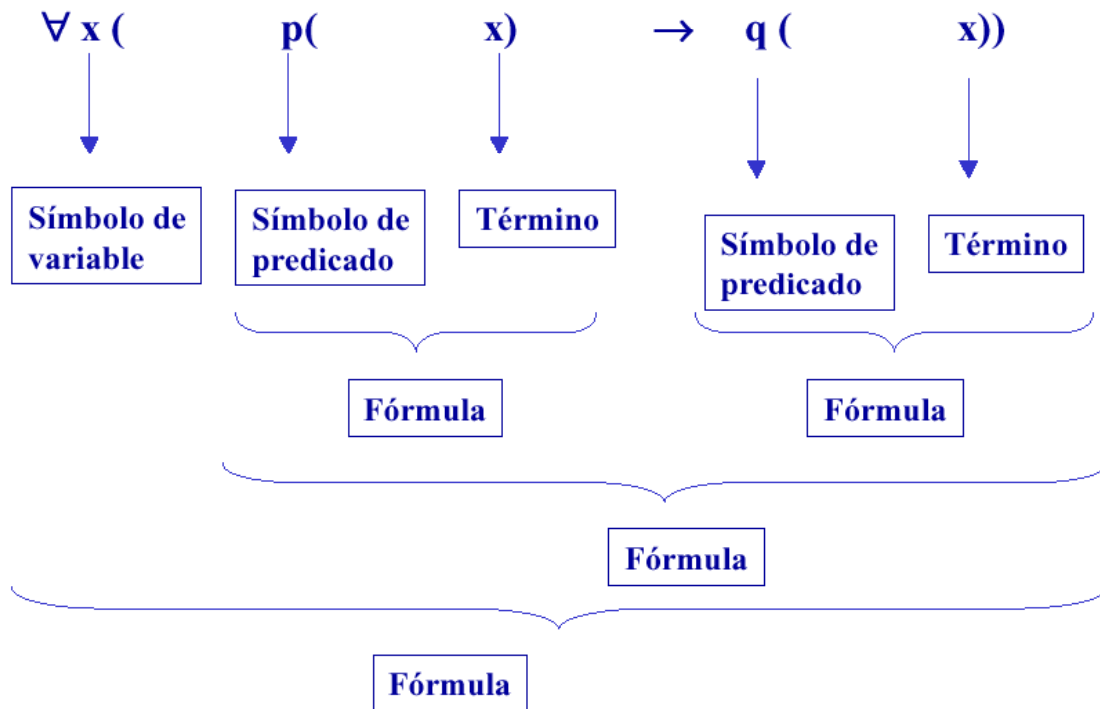
- Los símbolos de verdad T y el de falsedad o contradicción \perp son fórmulas
- Si p es un símbolo de predicado de aridad n , y $t_1, t_2, ..., t_n$ son términos, entonces $p(t_1, t_2, ..., t_n)$ es una fórmula

COMPUESTAS

- Si F es una fórmula, entonces $\neg F$ es una fórmula
- Si F y G son fórmulas, entonces:
 - $(F \wedge G)$ es una fórmula
 - $(F \vee G)$ es una fórmula
 - $(F \rightarrow G)$ es una fórmula
 - $(F \leftrightarrow G)$ es una fórmula
- Si x es un símbolo de variable, y F es una fórmula, entonces:
 - $\forall x F$ es una fórmula
 - $\exists x F$ es una fórmula

Lógica de predicados: sintaxis

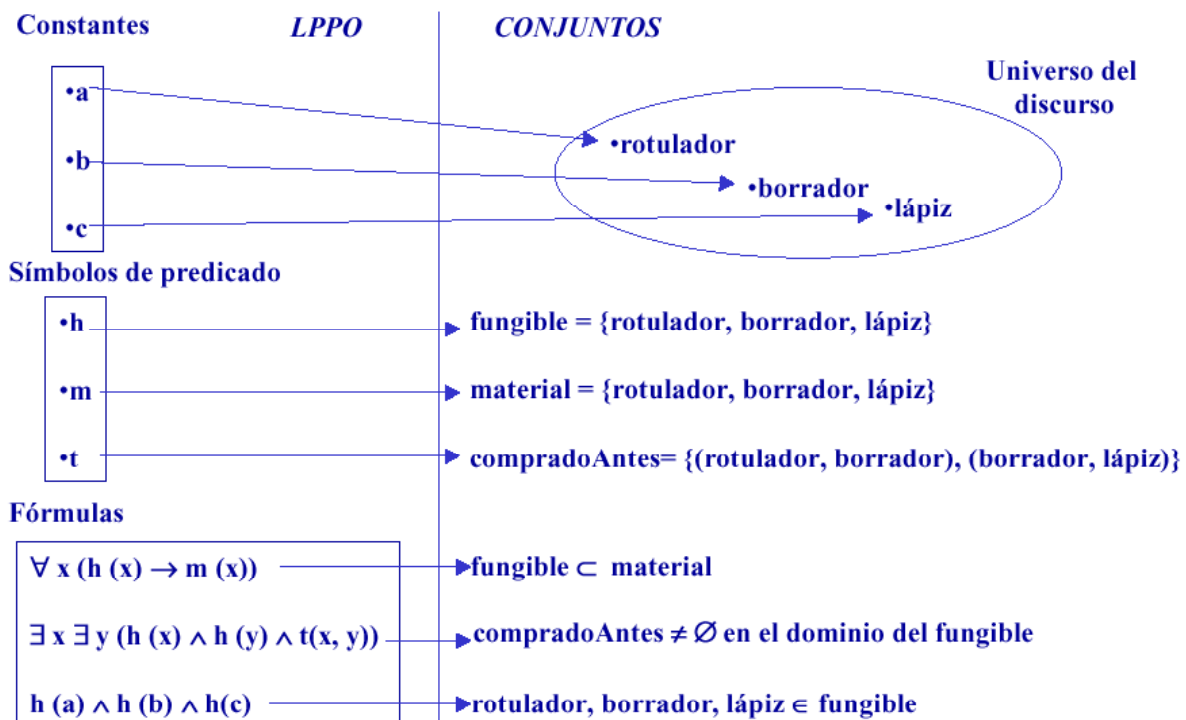
□ Ejemplo de fórmula: $\forall x (p(x) \rightarrow q(x))$



Lógica de predicados: semántica

- Los significados de términos y fórmulas se obtienen fijando una interpretación / que consta de
 - Un conjunto llamado U **universo (o dominio) de discurso**
 - Una asociación de elementos de U a los símbolos de constante
 - Una asociación de relaciones en U a los símbolos de predicado
 - Una asociación de funciones en U a los símbolos de función

Lógica de predicados: semántica



Lógica de predicados: propiedades

- ☐ Representación declarativa
 - ☐ No está fijada la forma en la que debe ser usado el conocimiento
 - ☐ Sencillez para incorporar nuevo conocimiento, o eliminarlo
 - ☐ Sencillez de integración de dos o más BCs
- ☐ Corrección y completitud
 - ☐ Cuenta con mecanismos deductivos correctos y completos
 - ☐ Permiten deducir nuevo conocimiento (conclusiones) a partir del conocimiento de partida (premisas)
 - ☐ Pueden usarse para responder a preguntas o resolver problemas
 - ☐ Demostración automática de teoremas: una de las áreas iniciales de la IA
- ☐ La deducción es semi-decidible en LPO
 - ☐ Si lo que pretendemos demostrar no se deduce del contenido de la base de conocimiento, no está garantizado que termine el proceso de demostración
 - ☐ No existe un procedimiento de decisión, ni siquiera de coste exponencial

Dificultades en la representación

- ❑ Lenguaje natural → lógica de predicados
 - ❑ Ambigüedad, sentido común, implicaciones y disyunciones...
- ❑ Incompletitud del conocimiento representado
 - ❑ Es imposible representar TODO el conocimiento
 - ❑ Representamos sólo lo más importante
 - ❑ No incluimos conocimiento de “sentido común”
 - habrá razonamientos que no podrán hacerse
- ❑ Infinidad de alternativas de representación dentro de la lógica
 - ❑ Influyen en el proceso de razonamiento. Algunas representaciones facilitan un tipo de razonamiento y dificultan otro
 - ❑ Lo importante es usar una representación consistente dentro de la misma BC
- ❑ Dificultades en la representación del conocimiento por defecto
 - ❑ Por ejemplo, las excepciones a la herencia

Alternativas de representación

- ❑ Ejemplo
 - ❑ Representación de conocimiento factual terminológico
 - ❑ Relaciones de ejemplar (\in) y subclase (\subseteq)
 - ❑ Propiedades esenciales
- ❑ Ventajas e inconvenientes de distintas alternativas
 - ❑ “Flipper es un delfín” *ejemplar de una clase*
 - ❑ “Todos los delfines son vertebrados” *relación de subclase*
 - ❑ “Los vertebrados tienen esqueleto” *propiedad esencial*
- ❑ Criterios para la elección
 - ❑ Qué pueden hacer eficientemente
 - ❑ Representar lo que tenemos
 - ❑ Obtener lo que buscamos (deducciones, inferencias)
 - ❑ Los gustos de cada cual
 - ❑ La representación sea consistente en todo el dominio

Alternativas de representación

❑ Versión 1

- ❑ Representación **implícita** de **ejemplares**: el nombre de la **clase** es un símbolo de predicado unario, el argumento es el ejemplar

Delfín(flipper)

- ❑ Representación de la **subclase** *delfin* de la clase *vertebrados*

$\forall x (Delfin(x) \rightarrow Vertebrado(x))$

- ❑ Representación de **propiedades esenciales**

$\forall x (Vertebrado(x) \rightarrow Tiene-Esqueleto(x))$

❑ Ventajas

- ❑ Sencillez de la representación

❑ Desventajas

- ❑ Para cualquier clase (Delfín, Elefante...)
 - ❑ crear predicados y reglas (subclases) específicas
- ❑ Se complica el razonamiento general
- ❑ El conocimiento implícito se deduce a través de las implicaciones.

Alternativas de representación

❑ Versión 2

- ❑ Representación **explícita** de la pertenencia de **ejemplares** a **clases**: se utiliza un símbolo de predicado binario **Es_Un(ejemplar, Clase)**

Es_Un(flipper, Delfín)

- ❑ Representación de la **subclase** *delfin* de la clase *vertebrados*

$\forall x (Es_Un(x, Delfin) \rightarrow Es_Un(x, Vertebrado))$

- ❑ Representación de **propiedades esenciales**

$\forall x (Es_Un(x, Vertebrado) \rightarrow Tiene_Esqueleto(x))$

❑ Ventajas

- ❑ Representación explícita de pertenencia de ejemplares a clases, **es_un**

❑ Desventajas

- ❑ La relación de subclase sigue siendo implícita

Alternativas de representación

❑ Versión 3

- ❑ Representación **explícita** de **ejemplares** y **subclases** utilizando pred. *Es_Un*

Es_Un(flipper, Delfín)

Es_Un(Delfín, Vertebrado)

$\forall x \text{ (} \textit{Es_Un}(x, \textit{Vertebrado}) \rightarrow \textit{Tiene_Esqueleto}(x) \text{)}$

❑ Ventajas

- ❑ Representación explícita de pertenencia de ejemplares a clases
- ❑ Representación explícita de la relación subclase

❑ Desventajas

- ❑ El sistema no puede diferenciar si *flipper* es individuo o clase
- ❑ No puede deducirse que *Flipper* tenga esqueleto
- ❑ Falta especificar la transitividad de la relación *Es_Un*. Hay que añadirla

$\forall x \forall y \forall z \text{ (} \textit{Es_Un}(x, y) \wedge \textit{Es_Un}(y, z) \rightarrow \textit{Es_Un}(x, z) \text{)}$

Alternativas de representación

❑ Versión 4

- ❑ Representación **explícita** de **ejemplares** y **subclases** utilizando dos símbolos de predicado binarios distintos

Ejemplar(flipper, Delfín)

Subclase(Delfín, Vertebrado)

$\forall x \text{ (} \textit{Ejemplar}(x, \textit{Vertebrado}) \rightarrow \textit{Tiene_Esqueleto}(x) \text{)}$

- ❑ Añadimos la **transitividad** (entre subclases y ejemplares)

$\forall x \forall y \forall z \text{ (} \textit{Ejemplar}(x, y) \wedge \textit{Subclase}(y, z) \rightarrow \textit{Ejemplar}(x, z) \text{)}$

$\forall x \forall y \forall z \text{ (} \textit{Subclase}(x, y) \wedge \textit{Subclase}(y, z) \rightarrow \textit{Subclase}(x, z) \text{)}$

Alternativas de representación

- ❑ Las cuatro versiones usan como técnica de representación a la lógica de predicados o lógica de primer orden (LPO)
 - ❑ Existen distintas posibilidades de representación dentro de la lógica (en general, esto ocurre con cualquier formalismo)
 - ❑ Algunas representaciones facilitan un tipo de razonamiento y dificultan otro

Conocimiento por omisión (by default)

- ❑ ¿Cómo añadir las excepciones a la herencia de propiedades?

$\forall x \text{ (Gorila}(x) \rightarrow \text{Pelo_Oscuro}(x))$

$\text{Gorila}(\text{Copito})$

$\neg \text{Pelo_Oscuro}(\text{Copito})$

- ❑ ¡Inconsistencia! (*inacceptable en una BC*)
- ❑ Se necesita incluir las excepciones dentro de la definición general

$\forall x \text{ (Gorila}(x) \wedge \neg \text{igual}(x, \text{Copito}) \rightarrow \text{Pelo_Oscuro}(x))$

$\text{Gorila}(\text{Copito})$

- ❑ Excepciones a la herencia son difíciles de representar en LPO
 - ❑ En Prolog se simplifica porque no hay que modificar las reglas generales sino que basta con colocar las excepciones delante

Mecanismos de inferencia

- ❑ **Deducción:** obtención de nuevo conocimiento (*implícito*)
- ❑ Se trata de saber si una fórmula Q es cierta conociendo
 - ❑ Los axiomas que son lógicamente válidos sea cual sea el significado de los símbolos (*tautologías*)
 $\neg F \vee F$
 - ❑ Los axiomas que son válidos sólo suponiendo ciertos significados de los símbolos (*conocimiento explícito*)
 - ❑ Las reglas de inferencia
 - ❑ Por ejemplo:

modus ponens

$$\begin{array}{l} P \rightarrow Q \\ P \\ \hline Q \end{array}$$

modus tollens

$$\begin{array}{l} P \rightarrow Q \\ \neg Q \\ \hline \neg P \end{array}$$

Ejemplo de deducción formal

- ❑ Dado un conjunto de hipótesis o premisas
 - $\text{perro}(\text{milú})$
 - $\forall x (\text{perro}(x) \rightarrow \text{animal}(x))$
 - $\forall y (\text{animal}(y) \rightarrow \text{mortal}(y))$
- ❑ Demostrar una conclusión
 - $\text{mortal}(\text{milú})$
- ❑ Pasos aplicados
 1. Aplicar instanciación universal con $x = \text{milú}$
 $\text{perro}(\text{milú}) \rightarrow \text{animal}(\text{milú})$
 2. Aplicar modus ponens
 $\text{animal}(\text{milú})$
 3. Aplicar instanciación universal con $y = \text{milú}$
 $\text{animal}(\text{milú}) \rightarrow \text{mortal}(\text{milú})$
 4. Aplicar modus ponens
 $\text{mortal}(\text{milú})$

Cláusulas y resolución

- ❑ En la práctica, resulta incómodo operar con un sistema deductivo formal en el que las expresiones lógicas utilizadas tienen formas muy variadas y se debe elegir entre muchas reglas de deducción aplicables en cada paso
- ❑ Esta situación se puede mejorar notablemente transformando las fórmulas lógicas a una **forma normal** más sencilla para operar
 - ❑ En particular, la forma de **cláusula** sólo utiliza las conectivas \neg y \vee (negación y disyunción), y necesita una sola regla de deducción

$$\begin{array}{ccc} \text{Resolución} & & [\text{Robinson, 1965}] \\ P \vee Q & \sim & \neg P \rightarrow Q \\ \neg Q \vee R & \sim & Q \rightarrow R \\ \hline P \vee R & & \neg P \rightarrow R \end{array}$$

- ❑ Con esta restricción no se pierde generalidad porque existe un algoritmo de conversión a forma normal conjuntiva que permite plantear cualquier problema lógico en forma de cláusulas

Forma clausal: algoritmo de conversión

- ❑ Eliminar la conectiva \rightarrow utilizando la equivalencia lógica
$$(\varphi \rightarrow \psi) \sim (\neg \varphi \vee \psi)$$
- ❑ Reducir el alcance de cada \neg , para que sólo afecten a fórmulas atómicas, utilizando las equivalencias lógicas
$$\begin{array}{lll} \neg \neg \varphi \sim \varphi & \neg(\varphi \wedge \psi) \sim (\neg \varphi \vee \neg \psi) & \neg(\varphi \vee \psi) \sim (\neg \varphi \wedge \neg \psi) \\ \neg \exists x \varphi \sim \forall x \neg \varphi & \neg \forall x \varphi \sim \exists x \neg \varphi & \end{array}$$
- ❑ Renombrar las variables ligadas por distintos cuantificadores, de manera que todas tengan nombres distintos
$$\exists x \varphi \sim \exists y \varphi[x/y] \quad \forall x \varphi \sim \forall y \varphi[x/y] \quad \text{si } y \text{ no está en } \text{lib}(\varphi)$$
- ❑ Trasladar todos los cuantificadores al principio (**forma normal prenexa**), sin cambiar su orden relativo, aplicando equivalencias lógicas
$$\begin{array}{lll} \exists x \varphi \vee \exists x \psi \sim \exists x (\varphi \vee \psi) & \forall x \varphi \wedge \forall x \psi \sim \forall x (\varphi \wedge \psi) & \\ \exists x \varphi \vee \psi \sim \exists x (\varphi \vee \psi) & \exists x \varphi \wedge \psi \sim \exists x (\varphi \wedge \psi) & \text{si } x \text{ no está en } \text{lib}(\psi) \\ \forall x \varphi \vee \psi \sim \forall x (\varphi \vee \psi) & \forall x \varphi \wedge \psi \sim \forall x (\varphi \wedge \psi) & \text{si } x \text{ no está en } \text{lib}(\psi) \end{array}$$

Forma clausal

- ❑ Eliminar los cuantificadores existenciales mediante *skolemización*, i.e., reemplazar las variables cuantificadas existencialmente por:
 - ❑ una constante nueva c si el cuantificador existencial asociado no se encuentra dentro del alcance de ningún cuantificador universal
 - ❑ un término *nuevo* $f(x)$ si el cuantificador existencial se encuentra dentro del alcance de un cuantificador universal que cuantifique a la variable x
 - ❑ un término *nuevo* $g(x, y)$ si el cuantificador existencial se encuentra dentro del alcance de cuantificadores universales para x e y , etcétera.
- ❑ Eliminar el prefijo de la fórmula: todos los cuantificadores universales que quedan al principio de la fórmula
- ❑ Distribuir \vee sobre \wedge para convertirla en una conjunción de disyunciones (*forma normal conjuntiva*) aplicando la equivalencia $(\varphi \vee (\psi \wedge \mu)) \sim ((\varphi \vee \psi) \wedge (\varphi \vee \mu))$
- ❑ Cada componente de la conjunción resultante se convierte en una cláusula independiente constituida por una disyunción de literales (fórmulas atómicas negadas o no) (*forma clausal*)

Deducción por refutación

Base de conocimientos: $Q \leftarrow T$, $T \leftarrow S$, $S \leftarrow P$, P

Consulta: Q ?

Paso 1. $Q \leftarrow T$ $\neg Q$ ----- $\neg T$	Paso 2. $T \leftarrow S$ $\neg T$ ----- $\neg S$	Paso 3. $S \leftarrow P$ $\neg S$ ----- $\neg P$	Paso 4. P $\neg P$ ----- \cdot
---	---	---	---

Paso 1. Se niega lo que se pretende demostrar

Pasos 1, 2 y 3. Aplicación reiterada de modus tolens

Paso 4. $P \wedge \neg P \sim \perp$ se llega a contradicción

Refutación: razonamiento por reducción al absurdo

$$\Phi \models \psi \Leftrightarrow \text{Insat } \Phi \cup \{\neg\psi\}$$

Deducción por resolución

Ejemplo: ¿a?

$b \wedge c \rightarrow a$
 $d \rightarrow c$
 b
 d

Forma clausal

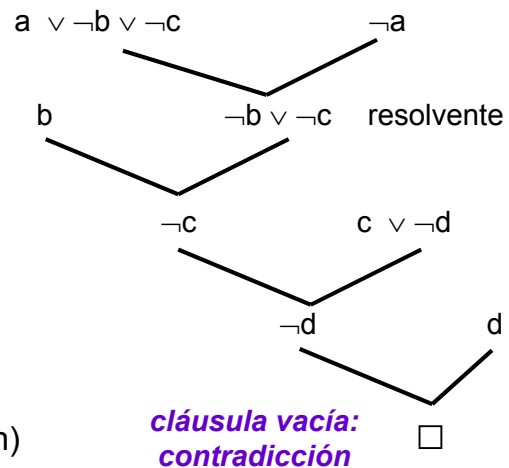
$a \vee \neg b \vee \neg c$
 $c \vee \neg d$
 b
 d

Añadir la negación de lo que queremos demostrar: $\neg a$
y probar que llegamos a una contradicción (*conjunto insatisfactible*)

Resolución

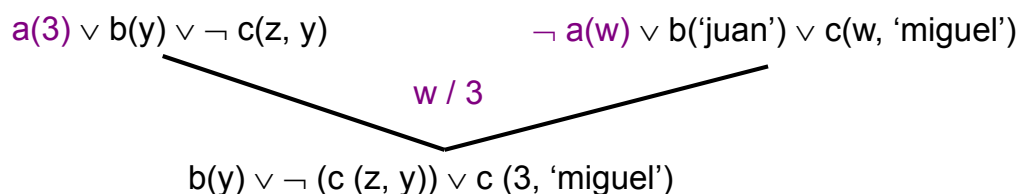
$\varphi \vee Q$ *un literal positivo en una cláusula*
 $\neg Q \vee \psi$ *y el mismo negativo en otra*

 $\varphi \vee \psi$ **resolvente**

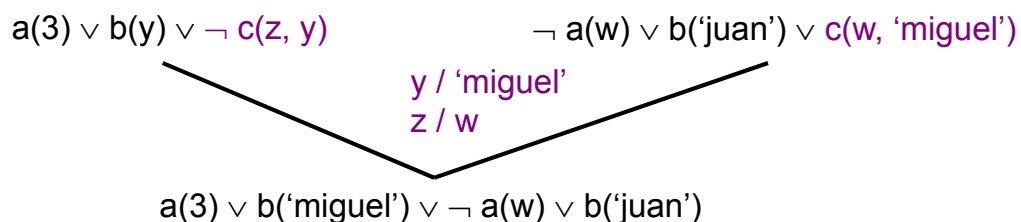


Proposiciones \rightarrow Predicados
 (Resolución con unificación)

Resolución con unificación



U.m.g.: **unificador más general posible** de los literales candidatos
 Se aplica a las cláusulas padres antes de calcular el resolvente
 Deben considerarse **variantes** de las cláusulas (variables frescas)



Se selecciona un único par de literales complementarios

Estrategias de resolución

- ❑ No es fácil determinar en cada paso qué cláusulas resolver y con qué literales (*grado de no determinismo muy alto*)
 - ❑ Si la elección se realiza de forma sistemática, la resolución llegará a contradicción si el conjunto de cláusulas es insatisfacible
 - ❑ Así es completa, pero puede requerir mucho tiempo...
- ❑ Se utilizan diversas estrategias para acelerar el proceso
 - ❑ Indexar las cláusulas por los predicados que contienen, indicando si están o no negados, para facilitar su localización
 - ❑ Eliminar tautologías ($\varphi \vee \neg \varphi$) y cláusulas que son *subsumidas* por otras (son implicadas por otras: $\varphi \vee \psi$ es subsumida por φ)
 - ❑ Intentar resolver con una de las cláusulas de la fórmula que estamos intentando refutar, o con alguna cláusula que se haya obtenido por resolución a partir de una de ellas (*intuición: la contradicción tiene que salir de ahí*)
 - ❑ Resolver con cláusulas que tengan un solo literal (*idea: disminuye el tamaño de las cláusulas generadas*)

Contestar a preguntas

- ❑ ¿Tiene esqueleto *flipper*?
 - ❑ Hay dos opciones para saber si algo se deduce o no de la BC
 - 1) Demostrar `Tiene_esqueleto(flipper)`
 - ➔ añadir `¬Tiene_esqueleto(flipper)`
 - 2) Demostrar `¬Tiene_esqueleto(flipper)`
 - ➔ añadir `Tiene_esqueleto(flipper)`
 - ❑ La opción correcta depende de cómo son los predicados de la BC

```
Ejemplar(flipper, Delfín)
Subclase(Delfín, Vertebrado)
∀x (Ejemplar(x, Vertebrado) → Tiene_Esqueleto(x))
∀x∀y∀z (Ejemplar(x,y) ∧ Subclase(y,z) →
    Ejemplar(x,z))
∀x∀y∀z (Subclase(x,y) ∧ Subclase(y,z) →
    Subclase(x,z))
```

Contestar a preguntas

❑ ¿Existe algún delfín?

❑ Demostrar $\exists x \text{ Ejemplar}(x, \text{Delfín})$

→ añadir $\neg \exists x \text{ Ejemplar}(x, \text{Delfín})$ en forma clausal

→ $\sim \forall x \neg \text{Ejemplar}(x, \text{Delfín})$ en forma clausal

→ añadir $\neg \text{Ejemplar}(t, \text{Delfín})$ variante

❑ Cómo preguntar depende de cómo son los predicados de la BC

$\text{Ejemplar}(\text{flipper}, \text{Delfín})$

$\text{Subclase}(\text{Delfín}, \text{Vertebrado})$

$\forall x (\text{Ejemplar}(x, \text{Vertebrado}) \rightarrow \text{Tiene_Esqueleto}(x))$

$\forall x \forall y \forall z (\text{Ejemplar}(x, y) \wedge \text{Subclase}(y, z) \rightarrow$
 $\text{Ejemplar}(x, z))$

$\forall x \forall y \forall z (\text{Subclase}(x, y) \wedge \text{Subclase}(y, z) \rightarrow$
 $\text{Subclase}(x, z))$

❑ Si se requiere usar unificación, la resolución devolverá los valores que hacen cierta la pregunta: $\{t = \text{flipper}\}$

Problemas de la representación con LPO

❑ La resolución no es la forma en que una persona piensa

❑ No es válida en sistemas de enseñanza o de diagnóstico médico en los que el sistema debe explicar su razonamiento

❑ Proceso de búsqueda no guiado por el razonamiento humano

❑ La resolución, con la transformación a forma clausal, no es lo más indicado para que una persona interactúe con la máquina, para ayudar

❑ Demostración por búsqueda → problemas de eficiencia

❑ Necesidad de heurísticas, indexación de la base de conocimiento

❑ BC plana: todas las cláusulas tienen la misma importancia
→ añadir meta-conocimiento para dirigir las búsquedas

❑ No organizable, imposibilidad de priorizar: explosión combinatoria

❑ Es difícil representar los distintos tipos de conocimiento

❑ Excepciones como cláusulas

❑ Conocimiento impreciso → ampliación con otros cuantificadores

❑ Conocimiento heurístico → difícil de representar

--Técnicas de representación y razonamiento--

❑ Tema 3: Representación del conocimiento e inferencia

❑ 3.2: Lógica y Prolog – Índice de contenidos

❑ Lógica de predicados (Leng. Formal Representación)

- ❑ Sintaxis, semántica, propiedades
- ❑ ¿Cómo usar la lógica para representar conocimiento?
- ❑ Mecanismos de inferencia: deducción, resolución
- ❑ Problemas de la representación con LPO

❑ Representación de conocimiento con Prolog (Implementación)

- ❑ Conceptos básicos
- ❑ Representación de conocimiento factual con Prolog
- ❑ Consultas, backtracking, negación, reglas
- ❑ Relaciones transitivas
- ❑ Herencia de propiedades
- ❑ Relaciones simétricas
- ❑ Listas

-Representa conocimiento con Prolog: Implementación-

❑ Conceptos básicos

- ❑ Término, variable, constante, átomo, estructura

❑ Representación de conocimiento factual

❑ Consultas, backtracking, negación

❑ Reglas

❑ Relaciones transitivas

❑ Herencia de propiedades

❑ Relaciones simétricas

❑ Listas

❑ Se complementa con los ejercicios propuestos en la hoja 3

Prolog

- ❑ Sistema de Programación Lógica
 - ❑ Conjunto de fórmulas = programa
- ❑ Representación del conocimiento a medio camino entre
 - ❑ Una representación declarativa
 - ❑ La lógica de predicados o de primer orden
 - ❑ Prolog no es declarativo puro porque tiene fijado el mecanismo de inferencia, aparte de por otras cuestiones
 - ❑ Una representación procedimental
 - ❑ Se clasifica más dentro de los sistemas de producción

Representación declarativa vs. procedimental

- ❑ Representación declarativa
 - ❑ Representamos el conocimiento pero no la forma de utilizarlo
 - ❑ Una representación declarativa debe acompañarse con algún programa que especifique qué hacer con el conocimiento y cómo
 - ❑ Por ejemplo, fórmulas lógicas + deducción o resolución
- ❑ Representación procedimental
 - ❑ Otro punto de vista: un conjunto de fórmulas lógicas puede verse en lugar de como un conjunto de datos que se le suministra a un programa, como un programa por sí mismas
 - ❑ Las implicaciones establecen la forma de razonar (los caminos legítimos para hacerlo) y las fórmulas atómicas nos dan los puntos de partida, razonando hacia delante, o, si razonamos hacia atrás, los puntos de llegada de esos caminos
 - ❑ La información de control para usar el conocimiento forma parte de la propia representación del conocimiento

Representación declarativa vs. procedimental

☐ No determinismo

- ☐ Si hay varias alternativas, como un programa necesita tener **determinada** una forma de proceder, el intérprete tendrá que tener fijada una **estrategia** concreta para realizar estas elecciones
- ☐ Por ejemplo, se pueden examinar las fórmulas en el orden textual en el que aparecen en el programa y la búsqueda puede hacerse primero en profundidad
- ☐ Esta estrategia forma parte del sistema y es lo que lo convierte en procedimental
- ☐ Un punto de vista declarativo consideraría todas las alternativas

☐ Mucha controversia en IA sobre qué tipo de representación es mejor

- ☐ Estudiaremos cómo distintos formalismos basados en reglas e intérpretes pueden combinarse para resolver problemas

Prolog

☐ Conocimiento expresable sólo como **cláusulas de Horn** (*Prolog puro; la parte impura de Prolog se desvía*)

- ☐ Subconjunto decidible de la LPO
- ☐ Básicamente se trata de cláusulas que tienen como mucho un literal positivo
 - ☐ Restricción que lleva a una representación uniforme del conocimiento
 - ☐ Esto posibilita la implementación de un intérprete sencillo y eficiente

☐ Razonamiento hacia atrás (*a partir de un objetivo*)

☐ Exploración de la BC en orden prefijado (*de arriba a abajo*)

☐ Búsqueda primero en profundidad con *backtracking*

- ☐ **Resolución SLD** (estrategia de resolución fija)
 - ☐ Selecting a literal, using a linear strategy, restricted to definite clauses
- ☐ En profundidad y el subobjetivo más a la izquierda

☐ Principal ventaja e inconveniente: estrategia de control fija

Prolog: conceptos básicos

- ❑ **Término**: constante, variable o estructura (*compuestos*)
 - ❑ **Término cerrado**: término que no contiene variables
- ❑ **Variable**: secuencia de caracteres (*letras, números y _*) que empieza por letra mayúscula o por `_` (*A, Algo, _algo, _*)
- ❑ **Constante**: número (*3, -3, 3.14, 2.8e+20*) o átomo
 - ❑ **Átomo**: cualquier secuencia de caracteres (*letras, números y _*) que empiece por letra minúscula o aparezca entre comillas simples
- ❑ **Estructura**: functor seguido de uno o más términos, denominados argumentos, entre paréntesis y separados por comas (*c(t1, t2, ..., tn)*)
 - ❑ Cada **functor** se caracteriza por su nombre, que es un átomo, y su aridad (*nº de argumentos*)
 - ❑ Dos funtores con el mismo nombre y distinta aridad se consideran distintos
 - ❑ Un functor *f* de aridad *n* se representa como *f/n*

Representación con Prolog: cláusulas de Horn

- ❑ Se basa en la formalización de **cláusulas de Horn**
 - ❑ Una cláusula es una disyunción de cualquier número de **literales** (fórmulas atómicas afirmadas o negadas)
 - ❑ Las cláusulas de Horn se caracterizan por tener un solo literal positivo y cualquier número de literales negativos
 - ❑ Por ejemplo: $P, Q \vee \neg P, R \vee \neg P \vee \neg Q$
 - ❑ No todas las fórmulas se pueden transformar en cláusulas de Horn
 - ❑ Constituyen un **subconjunto decidable de la LPO**
- ❑ Las cláusulas anteriores se escriben en Prolog como

Cláusulas de Horn

P
 $Q \vee \neg P$
 $R \vee \neg P \vee \neg Q$

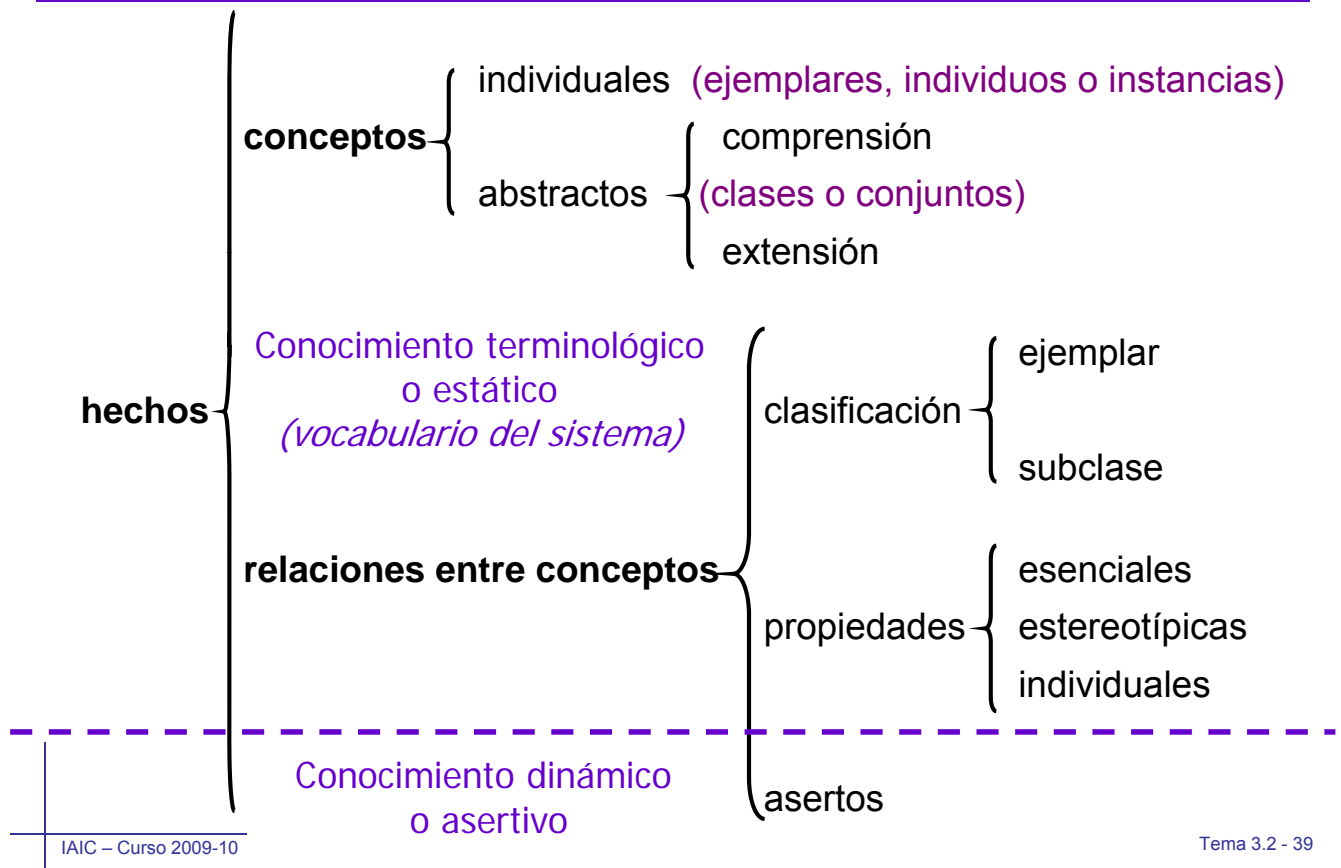
Fórmulas equivalentes

P
 $P \rightarrow Q$
 $P \wedge Q \rightarrow R$

Representación en Prolog

$p.$
 $q :- p.$
 $r :- p, q.$

Conocimiento factual: recordatorio



Representación con Prolog: conocimiento factual - I

- ❑ Fijamos convenios para la representación de los hechos
 - ❑ Conceptos (individuales o abstractos) → átomos
 - ❑ Argumentos o nombres de funtores
 - ❑ Relaciones entre conceptos → predicados → estructuras (*términos compuestos*)
- ❑ Los predicados pueden tener muchos significados. Por ello, clasificamos a algunos de ellos en categorías de predicados
 - ❑ Predicados de tipo
 - ❑ Predicados de propiedad
 - ❑ Predicados de relación
 - ❑ Predicados de aridad superior a 2
 - ❑ Predicados registro
 - ❑ Predicados función
 - ❑ Predicados con grado de certeza

Representación con Prolog: conocimiento factual - II

- ❑ **Predicados de tipo:** clase (*concepto abstracto*) a la que pertenece un ejemplar (*concepto individual*)

```
delfín(flipper).  
vertebrado(delfín).  
vehículo(automóvil).  
vehículo(motocicleta).  
objeto_físico(automóvil).
```

tipo(concepto).

- ❑ **Predicados de propiedad:** valor de una propiedad de un concepto

```
color(flipper, gris).  
color(aluminio, gris).  
fecha_nacimiento(juan, 1980).
```

propiedad(concepto, valor).

Representación con Prolog: conocimiento factual - III

- ❑ **Predicados de relación:** relación entre dos conceptos
 - ❑ Al interpretar su significado utilizamos el convenio infijo

relación(concepto1, concepto2).

```
es_parte_de(motor, coche).  
es_padre_de(juan, luis).  
es_un(flipper, delfín).  
es_un(delfín, vertebrado).  
ejemplar(flipper, delfín).  
subclase(delfín, vertebrado).
```

Relaciones de clasificación

Representación con Prolog: conocimiento factual - IV

❑ Predicados registro

```
datos_libro('El Quijote', 'Cervantes', 1605, 'Madrid').
```

❑ Predicados función

```
suma(3, 4, 7).
```

```
jefe_común(juan, maría, maite, luis).
```

2 o más argumentos

resultado: último argumento

❑ Predicados con grado de certeza:

variantes de todos los anteriores para hechos con grado de certeza

```
color(gorila, oscuro, 95).
```

grado de certeza: último argumento

Representación Prolog: objetivos, consultas – I

❑ Consultas sin variables

```
?- ejemplar(flipper, delfín).
```

yes

% éxito: se infiere de la BC

```
?- ejemplar(flipper, animal).
```

no

% hipótesis del mundo cerrado:

% ¿falso o BC incompleta?

❑ Consultas con variables

```
?- ejemplar(flipper, X).
```

X=delfín

```
?- ejemplar(X, delfín).
```

X=flipper

Base de conocimiento:

```
es_parte_de(motor, coche).
```

```
es_padre_de(juan, luis).
```

```
es_un(flipper, delfín).
```

```
es_un(delfín, vertebrado).
```

```
ejemplar(flipper, delfín).
```

```
subclase(delfín, vertebrado).
```

Representación Prolog: objetivos, consultas – II

- ❑ Hechos conocidos por el sistema (*la BC es el programa cargado*)

```
ejemplar(flipper, delfín).  
subclase(delfín, vertebrado).  
color(delfín, gris).  
ejemplar(juan, persona).  
ejemplar(maría, persona).
```

- ❑ Consultas con varias respuestas (*forzar fallo: backtracking*)

```
?- ejemplar(X, persona).  
X=juan ;  
X=maría ;  
no
```

Forzamos la vuelta a atrás,
solicitando más soluciones con ;

Representación Prolog: objetivos, consultas - III

- ❑ Consultas con varias variables

- ❑ Las respuestas siguen el orden textual de la BC

```
?-ejemplar(X, Y).  
X=flipper, Y=delfín ;  
X=juan, Y=persona ;  
X=maría, Y=persona ;  
no
```

Base de conocimiento:

```
ejemplar(flipper, delfín).  
subclase(delfín, vertebrado).  
color(delfín, gris).  
ejemplar(juan, persona).  
ejemplar(maría, persona).
```

Representación Prolog: objetivos, consultas - IV

❑ Consultas compuestas

❑ Consultas conjuntivas (operador conjunción: ',')

```
?- ejemplar(flipper, X) , color(X, C).
```

```
X=delfín, C=gris
```

❑ Consultas disyuntivas (operador disyunción: ';')

```
?- ejemplar(flipper, X) ;  
    ( ejemplar(flipper,X) , subclase(X, Y) ).
```

```
X=delfin, Y=_2 ;
```

```
X=delfin, Y=vertebrado
```

Base de conocimiento:

```
ejemplar(flipper, delfín).  
subclase(delfín, vertebrado).  
color(delfín, gris).  
ejemplar(juan, persona).  
ejemplar(maría, persona).
```

Representación Prolog: backtracking

❑ Jefazos

```
?- jefe(X,Y), jefe(Y,Z).
```

```
X=luis, Y=marcos
```

No hay ligadura para Z → *backtracking automático*

```
X=jaime, Y=luis,
```

```
Z=marcos ;
```

 solicitamos más respuestas con ; (*backtracking forzado*)

No hay más ligaduras para Z → *backtracking automático*

```
X=marta, Y=ana,
```

```
Z=marcos ;
```

```
X=ana Y=marcos
```

No hay ligadura para Z → *backtracking automático* → Fin

```
no
```

Base de conocimiento:

```
jefe(luis, marcos).  
jefe(jaime, luis).  
jefe(marta, ana).  
jefe(ana, marcos).
```


Representación Prolog: Negación - I

❑ La negación lógica no puede representarse explícitamente en Prolog puro

❑ Queda representada implícitamente por ausencia (*hipótesis del mundo cerrado*)

- ❑ Como no podemos incluir explícitamente lo que no se cumple, directamente no se incluye en el programa

```
?- ejemplar(flipper, perro).
```

no

Como Prolog no es capaz de deducirlo, contesta que no es cierto

- ❑ Con esta consulta ocurría lo mismo, pero era debido a que la BC era incompleta, y no a que fuera falso

```
?- ejemplar(flipper, animal).
```

no

❑ Esto nos lleva a la estrategia de la **negación por fallo**

Representación Prolog: Negación - II

❑ Predicado predefinido \+ (*not/1*)

- ❑ Negación por fallo finito (*no es la negación lógica*)

❑ El intérprete intenta demostrar el predicado que aparece negado

- ❑ Si tiene éxito, la negación falla
- ❑ Si falla (*falso o BC incompleta*), la negación tiene éxito
- ❑ Si no termina, la negación tampoco
- ❑ Sólo es lógicamente correcta (es decir, funciona como se pretende) si el argumento de \+ es un término cerrado

```
?- \+(color(delfin, azul)).
```

yes

- ❑ Cuando se usan variables, no se producen ligaduras de éstas

Definición de \+:

```
\+(P) :- P, !, fail.
```

```
\+(P).
```

No es Prolog puro:

- corte rojo
- lectura no declarativa

Representación Prolog: Negación - III

Base de conocimiento:

```
ejemplar(flipper, delfín).
subclase(delfín, vertebrado).
color(delfín, gris).
ejemplar(juan, persona).
ejemplar(maría, persona).
```

+

Añadimos:

```
ejemplar(clipper, delfín).
color(clipper, azul).
```

- ☐ Por ejemplo, ¿existe algún delfín que no sea gris?

```
?- \+(color(X, gris)), ejemplar(X, delfín).
```

no

ya que existe algún X (*delfín*) que satisface *color(X, gris)*, la negación falla sin que se compruebe si ese X es un ejemplar de la clase *delfín*

El orden de los objetivos es muy importante al usar \+

Representación Prolog: Negación - IV

Base de conocimiento:

```
ejemplar(flipper, delfín).
subclase(delfín, vertebrado).
color(delfín, gris).
ejemplar(juan, persona).
ejemplar(maría, persona).
```

+

Añadimos:

```
ejemplar(clipper, delfín).
color(clipper, azul).
```

- ☐ Y, sin embargo, si intercambiamos el orden de los objetivos

```
?- ejemplar(X, delfin), \+(color(X, gris)).
```



```
X=flipper ;
```

```
X=clipper ;
```

no

El orden de los objetivos es muy importante al usar \+
variables instanciadas en \+ (términos cerrados al ejecutarse \+)

Representación Prolog: Reglas - I

- ❑ Las reglas constituyen una forma de modularizar conocimiento similar a las subrutinas en otros lenguajes
- ❑ Una **regla** Prolog consta de una parte izquierda y de una parte derecha separadas por el operador **:-**

A :- B1, ..., Bn. (n > 0)

- ❑ La parte izquierda (conclusión o **cabeza** de la regla) describe lo que se está definiendo. Es un término
- ❑ La parte derecha (antecedente, premisas o **cuerpo** de la regla) describe la conjunción de objetivos que ha de satisfacerse para que la cabeza sea cierta

- ❑ Un **hecho** Prolog también puede escribirse en forma de regla

A :- true. (n = 0, conjunción vacía)

- ❑ No tiene condiciones o premisas. Suele abreviarse como

A.

Representación Prolog: (doble lectura) Reglas - II

A :- B11, ..., B1n1.

... condiciones suficientes para A

A :- Bm1, ..., B1nm.

- ❑ Las cláusulas que definen a un predicado admiten **doble lectura**

1.- el orden es indiferente: Declarativa o lógica
como conjunto de fórmulas

B11 ∧ ... ∧ B1n1 → A

...

Bm1 ∧ ... ∧ B1nm → A

2.- el orden es importante (Prolog): Procedimental (Kowalski)

Para demostrar que **A** se cumple hemos de comprobar primero si se cumple **B11**, ..., y, si es así, si se cumple **B1n1** o ... o si se cumple **Bm1**, ..., y, si es así, si se cumple **B1nm**

- ❑ Si usamos la parte impura, nos cargamos la lectura declarativa

- ❑ Se puede escribir teniendo presente el tipo 2.- para optimizar.

Representación Prolog: Reglas - III

- ❑ Al intentar demostrar un objetivo, el intérprete Prolog busca la primera cláusula cuya cabeza unifique con él y lo sustituye por el cuerpo de la regla (afectado por el unificador) e intenta demostrar de izquierda a derecha los objetivos que contiene el cuerpo
 - ❑ A su vez, estos objetivos pueden ser cabezas de otras reglas, generando toda una jerarquía de llamadas a reglas
- ❑ Si en el cuerpo de la regla aparece la propia cabeza de esa regla, la regla es **recursiva**

```
color(X, C) :- es_parte_de(X, Y), color(Y, C).
```

- ❑ Las variables que figuran como parámetros en la cabeza de la regla están cuantificadas universalmente
- ❑ Las variables que sólo figuran en el cuerpo de la regla son variables locales cuantificadas existencialmente y sus vínculos no formarán parte de la respuesta del intérprete

```
 $\forall X \forall C (\exists Y (es\_parte\_de(X, Y) \wedge color(Y, C)) \rightarrow color(X, C))$ 
```

Representación Prolog: (legibilidad) Reglas – IV

- ❑ Cuando la parte derecha de una regla es una disyunción

```
progenitor(X, Y) :- (padre(X, Y) ; madre(X, Y)).
```

se recomienda dividirla en dos reglas distintas para favorecer la legibilidad

```
progenitor(X, Y) :- padre(X, Y).
```

```
progenitor(X, Y) :- madre(X, Y).
```

Cada una de ellas expresa una condición suficiente, pero no necesaria, para que sea cierto el objetivo *progenitor(X, Y)*

Orden de las cláusulas en la BC

- ☐ El orden en el que se colocan las cláusulas que definen a un predicado y el orden en el que se colocan los subobjetivos en el cuerpo de una regla pueden afectar
 - ☐ A la solución encontrada
 - ☐ A la eficiencia del proceso de búsqueda
 - ☐ Y a la terminación de dicho proceso
- ☐ Se recomienda poner en primer lugar los hechos y, a continuación, las reglas empezando por las más simples
 - ☐ Consideramos que son más simples las que no generan llamadas a otros predicados o las que menos llamadas generan
- ☐ En el cuerpo de una regla se recomienda colocar primero los objetivos más difíciles de satisfacer
 - ☐ Como los objetivos se evalúan de izquierda a derecha, estaremos disminuyendo el factor de ramificación en los primeros niveles del árbol de búsqueda, aumentando la eficiencia de la búsqueda

Representación de conocimiento con Prolog

- ☐ Conceptos básicos
 - ☐ Término, variable, constante, átomo, estructura
 - ☐ Representación de conocimiento factual
 - ☐ Consultas, backtracking, negación
 - ☐ Reglas
 - ☐ Relaciones transitivas
 - ☐ Herencia de propiedades
 - ☐ Relaciones simétricas
 - ☐ Listas
- ☐ Se complementa con los ejercicios propuestos en la hoja 3

Representación Prolog: relaciones transitivas - I

- ❑ Un **predicado de relación** r es **transitivo** si es cierto todo lo que pueda inferirse con la regla

```
r(X, Y) :- r(X, Z), r(Z, Y).
```

- ❑ Hay relaciones que son claramente transitivas, como la relación `es_un` y la relación `es_parte_de` (y *jefe*)

- ❑ Definiendo explícitamente la transitividad nos ahorramos la representación de todo lo que se inferiría por transitividad

- ❑ Establecer como hechos las relaciones directas
(entre individuos o clases inmediatamente próximos)

```
r(a, c).           % por ejemplo
```

```
r(c, e).           % por ejemplo
```

- ❑ E inferir, a partir de la regla, las relaciones indirectas (más lejanas)

```
r(a, e)            % por ejemplo
```

Representación Prolog: relaciones transitivas - II

- ❑ Dada la siguiente base de conocimiento

```
es_un(dumbo, elefante).
```

```
es_un(elefante, vertebrado).
```

```
es_un(vertebrado, animal).
```

```
es_un(X, Y):- es_un(X, Z), es_un(Z, Y).
```

- ❑ Ante la consulta

```
?- es_un(dumbo, animal).
```

el intérprete aplica la regla dos veces, finalizando la demostración con éxito

- ❑ Pero entra en un ciclo ∞ de llamadas a la regla recursiva si la consulta es

```
?- es_un(flipper, animal).
```

```
ERROR: Out of local stack
```

Representación Prolog: relaciones transitivas - III

- ❑ El establecimiento de la transitividad es una herramienta muy útil que nos ahorra tener que introducir en la base de conocimiento grandes cantidades de hechos, pero **debemos garantizar la convergencia de las llamadas recursivas**
- ❑ Para ello, se suelen utilizar dos predicados diferentes: uno para establecer las relaciones directas y otro distinto para preguntar por relaciones directas o indirectas
- ❑ Por ejemplo, la **clausura transitiva** de *tieneParte*

```
tieneParteTrans(X, Y) :- tieneParte(X, Y).  
tieneParteTrans(X, Y) :-  
    tieneParte(X, Z),  
    tieneParteTrans(Z, Y).
```

 - ❑ *tieneParte* se utilizará para establecer las relaciones directas como hechos y *tieneParteTrans* para lanzar objetivos

Repres. Prolog: Herencia predicados de propiedad - I

- ❑ La **herencia** es un mecanismo muy útil que permite disminuir significativamente el número de hechos a representar en la BC
- ❑ Normalmente involucra a dos predicados, uno de relación y otro de propiedad
- ❑ Un predicado de propiedad *p* se hereda con respecto al predicado de relación *r* si alguna de estas dos reglas es correcta:

```
p(X, Valor) :- r(X, Y), p(Y, Valor).  
p(X, Valor) :- r(Y, X), p(Y, Valor).
```

 - % Es recomendable que *r* sea una relación directa (establecida sólo con hechos Prolog)
- ❑ Por ejemplo, el apellido se hereda de padres a hijos

```
tieneApellido(X, Apellido) :-  
    esHijoDe(X, Y),  
    tieneApellido(Y, Apellido).
```

Repres. Prolog: Herencia predicados de propiedad - II

- ❑ Normalmente, las propiedades se heredan **de arriba a abajo** (de lo más general a lo más particular) a través de la jerarquía definida por la relación *r*
 - ❑ Ejemplo claro de herencia: cuando la relación *r* es la relación *es_un*
 - ❑ Por ejemplo, la propiedad universal de *tener esqueleto* se hereda de la clase *vertebrado* hacia abajo a través de la relación *es_un*
 - ❑ Las propiedades universales suelen heredarse de arriba abajo
- ❑ En otros casos, la propiedad se hereda **de abajo a arriba** (de lo particular a lo general)
 - ❑ Las propiedades existenciales suelen heredarse así
 - ❑ Por ejemplo, la propiedad existencial de *vivir en España* se hereda de la clase *hombre* hacia arriba a la clase *persona* a través de la relación *es_un*
 - ❑ Si algunos hombres viven en España, podemos concluir que algunas personas viven en España (*un hombre es una persona*)

Repres. Prolog: Herencia predicados de propiedad - III

- ❑ Por ejemplo, dada la base de conocimiento

```
es_un(portaaviones, buque_de_guerra).
es_un(buque_de_guerra, barco).
es_un(barco, vehículo).
propósito(vehículo, transporte).
```

y la regla de herencia para la propiedad *propósito* a través de la relación *es_un*

```
propósito(X, P):- es_un(X, Y), propósito(Y, P).
```

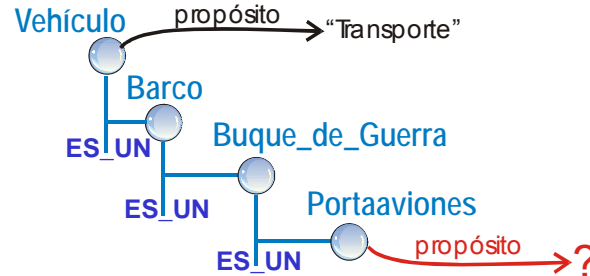
- ❑ ¿Qué respondería el sistema ante la siguiente consulta?

```
?- propósito(portaaviones, P).
```


Repres. Prolog: Herencia predicados de propiedad - IV

?- propósito(portaaviones, P).

- El intérprete ascendería por la jerarquía *es_un*, aplicando 3 veces la regla de herencia para obtener **P=transporte**

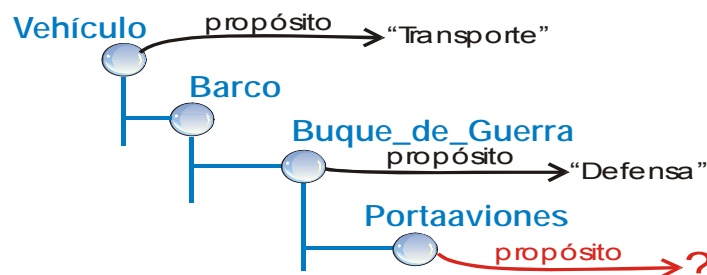


```
es_un(portaaviones, buque_de_guerra).  
es_un(buque_de_guerra, barco).  
es_un(barco, vehiculo).  
propósito(vehículo, transporte).  
propósito(X, P):- es_un(X, Y), propósito(Y, P).
```

Repres. Prolog: Herencia predicados de propiedad - V

?- propósito(portaaviones, P).

- Si hubiéramos tenido un hecho **propósito(buque_de_guerra, defensa).**
el sistema hubiera devuelto primero **P=defensa**



```
es_un(portaaviones, buque_de_guerra).  
es_un(buque_de_guerra, barco).  
es_un(barco, vehiculo).  
propósito(vehículo, transporte).  
propósito(buque_de_guerra, defensa).  
propósito(X, P):- es_un(X, Y), propósito(Y, P).
```

Repres. Prolog: Herencia predicados de propiedad - VI

- ❑ Valor de una propiedad para un determinado objeto
 - ❑ El intérprete comprueba primero si dicha propiedad ha sido establecida directamente como un hecho
 - ❑ ¡Sólo si los hechos aparecen antes que la regla de herencia!
 - ❑ Si no es así y existe una regla de herencia de esa propiedad a través de cierta relación r , se moverá por la jerarquía para encontrar el objeto más próximo (según esa relación) para el que ha sido definida la propiedad y la tomará directamente de él heredándola
- ❑ ¿Y las excepciones?

Repres. Prolog: Herencia predicados de propiedad - VII

- ❑ Excepciones a la herencia
 - ❑ Pueden solucionarse colocándolas como “hechos con corte” al principio
- ➡ `propósito(buque_de_guerra, defensa) :- !.`
`propósito(vehículo, transporte).`
`propósito(X, P):- es_un(X, Y), propósito(Y, P).`
- ❑ El sistema ya sólo devuelve **P=defensa**
 - ❑ El orden adecuado es
 1. Excepciones
 2. Propiedad estereotípica
 3. Regla de herencia

Repres Prolog: Herencia predicados de propiedad - VIII

- ❑ En el ejemplo anterior de herencia se ha supuesto que *es_un* es una relación directa establecida sólo con hechos

```
propósito(X, P):- es_un(X, Y), propósito(Y, P).
```

- ❑ Un predicado de propiedad *p* se hereda con respecto al predicado de relación *r* si alguna de estas dos reglas es correcta:

```
p(X, Valor) :- r(X, Y), p(Y, Valor).
```

```
p(X, Valor) :- r(Y, X), p(Y, Valor).
```

% Es recomendable que *r* sea una relación directa
(establecida sólo con hechos)

- ❑ Podríamos haber obtenido los mismos resultados aplicando primero

```
es_un(X, Y):- es_un(X, Z), es_un(Z, Y).
```

hasta obtener que *un portaaviones es un vehículo* y luego aplicando la regla de herencia una única vez

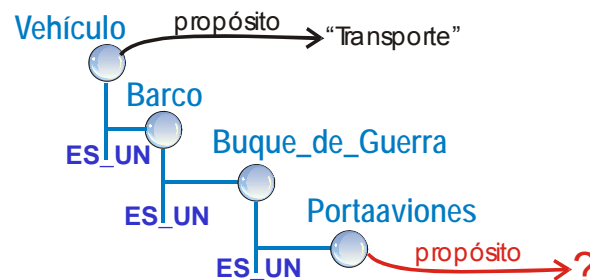
Repres. Prolog: Herencia predicados de propiedad - IX

- ❑ Aplicamos primero

```
es_un(X, Y):- es_un(X, Z), es_un(Z, Y).
```

y luego aplicamos la regla de herencia una única vez

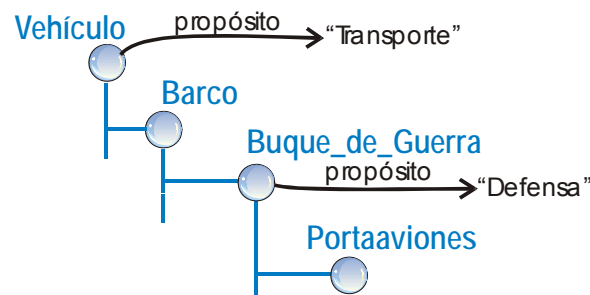
```
propósito(X, P):- es_un(X, Y), propósito(Y, P).
```



- ❑ ¿Mismo resultado siempre?
- ❑ ¿Por qué hemos recomendado el uso de relaciones directas, establecidas sólo con hechos?

Repres. Prolog: Herencia predicados de propiedad - X

```
es_un(portaaviones, buque_de_guerra).
es_un(buque_de_guerra, barco).
es_un(barco, vehículo).
es_un(X, Y):- es_un(X, Z), es_un(Z, Y).
propósito(buque_de_guerra, defensa) :- !.
propósito(vehículo, transporte).
propósito(X, P):- es_un(X, Y), propósito(Y, P).
```



?- propósito(submarino, P).



ciclo ∞

Repres. Prolog: Herencia predicados de propiedad - XI

☐ Herencia + relaciones “no directas”

☐ Puede funcionar pero hay que tener mucho cuidado...

```
es_un(portaaviones, buque_de_guerra).
es_un(buque_de_guerra, barco).
es_un(barco, vehículo).

propósito(buque_de_guerra, defensa) :- !.
propósito(vehículo, transporte).
propósito(X, P):- es_un_trans(X, Y), propósito(Y, P).

es_un_trans(X, Y):- es_un(X, Y).
es_un_trans(X, Y):- es_un(X, Z), es_un_trans(Z, Y).
```

Repres. Prolog: Herencia predicados de propiedad - XII

- ❑ En cambio, utilizando exclusivamente la herencia no correríamos el riesgo de la no terminación, ya que la regla de herencia no sería aplicable al fallar el objetivo `es_un(submarino, Y)`

```
es_un(portaaviones, buque_de_guerra).
es_un(buque_de_guerra, barco).
es_un(barco, vehículo).

propósito(buque_de_guerra, defensa) :- !.
propósito(vehículo, transporte).

propósito(X, P):- es_un(X, Y), propósito(Y, P).
```

Recomendaciones para la convergencia

- ❑ Al definir un predicado escribiremos los hechos antes que las reglas
- ❑ Evitaremos la recursividad por la izquierda

- ❑ Por ejemplo, no se debe utilizar

```
p(X, Valor) :- p(Y, Valor), r(X, Y).
```

para representar la herencia sino

```
p(X, Valor):-r(X, Y), p(Y, Valor).
```

asegurándose de que la relación *r* se establezca en forma de hechos o bien garantizando su adecuada convergencia, en caso de existir reglas para el predicado *r*

- ❑ Utilizaremos dos predicados distintos cuando se quieran representar predicados transitivos siempre que haya riesgo de generar ramas infinitas en el árbol de búsqueda

- ❑ Haremos esto en general para diferenciar predicados establecidos directamente de los que sirvan para realizar inferencias

Representación Prolog: relaciones simétricas

- ❑ Cuando se representan relaciones simétricas deben establecerse sólo en un determinado sentido

- ❑ Por ejemplo, los predicados *familiar_de*, *igual_a*

- ❑ Para todos estos predicados, lo normal es representar una única vez el hecho y establecer la simetría

- ❑ Si establecemos la simetría poniendo directamente

```
igual_a(X, Y) :- igual_a(Y, X).
```

tendríamos problemas de convergencia

- ❑ Lo más simple es definir un nuevo predicado que será el único con el que hagamos consultas: **cierre simétrico de la relación**

```
igual_a_Sim(X, Y) :- igual_a(X, Y).
```

```
igual_a_Sim(X, Y) :- igual_a(Y, X).
```

- ❑ Los hechos se establecerán exclusivamente con el predicado *igual_a* una única vez por pareja (*igual_a*: antisimétrica)

Representación Prolog: Listas - I

- ❑ Constructoras de listas en Prolog

Funcional

- ❑ Lista vacía

```
[]
```

```
[]
```

- ❑ Lista no vacía

```
[Cabeza | Resto]
```

```
(x:xs)
```

- ❑ *Cabeza* es un elemento (el primero) y *Resto* es la lista sin el 1º elemento

- ❑ Más patrones para listas

- ❑ De exactamente un elemento

```
[X]
```

- ❑ De al menos 2

```
[X, Y | Resto]
```

```
(x:y:ys)
```

- ❑ `|` separa los elementos que enumeramos de la variable que representa el resto de la lista `[X | Y | Resto]` es incorrecto

Representación Prolog: Listas - II

- ❑ Podemos representar listas por enumeración de sus elementos, escribiéndolos entre corchetes y separados por comas

`[a,b,c], [], [a, [b,c]], etc.`

- ❑ Las implementaciones de Prolog suelen incluir predicados para el manejo de listas como *member*, *append* o *length*

- ❑ SWI-Prolog los tiene predefinidos y se cargan por defecto

- ❑ SICStus Prolog también, pero no se cargan por defecto. Si se quiere disponer de estos predicados (u otros TADs habituales) es necesario cargar la biblioteca pertinente (*ver manual en ayuda*)

`:- use_module(library(lists)).`

Representación Prolog: No existe la asignación

- ❑ En lenguajes de programación declarativa no hay asignación

- ❑ `is/2` no es asignación

`VariableONúmero is ExpresiónAritmética`

- ❑ Se evalúa la expresión aritmética
 - ❑ Tiene éxito si y sólo si el lado izquierdo unifica con el resultado

- ❑ Tenéis disponible la unificación para devolver resultados

`relacionados(X, Y, C) :-`

`arista(R, X, Y),`

`C = [X, R, Y].` `% unificación`

- ❑ Lo mejor es usar unificación implícitamente: equivalente, pero más eficiente

`relacionados(X, Y, [X, R, Y]) :-`

`arista(R, X, Y).`

Representación Prolog: Especificación de predicados

- ❑ Se suelen anotar en la especificación de un predicado Prolog las posibles limitaciones de uso
 - ❑ **+**: el parámetro ha de estar instanciado, es decir, no puede ser una variable libre (sin ligar)
 - ❑ **?**: el parámetro puede estar instanciado o no
 - ❑ **-**: el parámetro debe ser una variable libre
- % predicado(+Instanciado, ?InstanciadoOVar, -Var)**
- ❑ Aunque una de las ventajas de la programación lógica son los múltiples modos de uso que puede tener un predicado, es habitual que no estén contemplados todos
 - ❑ Por eficiencia
 - ❑ Uso habitual del corte, pensando en un modo de uso concreto
 - ❑ Uso de aritmética Prolog
 - ❑ suma(+X, +Y, ?Z) si **suma(X, Y, Z) :- Z is X+Y.**
 - ❑ Características impuras en general: assert, retract...

Bibliografía

- ❑ **Rich, E. y Knight, K.**
Artificial Intelligence.
McGraw-Hill, 1991, 2ª edición.
- ❑ **Russell, S. y Norvig, P.**
Inteligencia Artificial: Un Enfoque Moderno.
Prentice Hall, 2004, 2ª edición.
- ❑ **Luger, G.F.**
Artificial Intelligence.
Addison-Wesley, 2005, 5ª edición.
- ❑ **Nilsson, J.**
Artificial Intelligence: A New Synthesis.
Prentice Hall, 2004, 2ª edición.

❑ **Jackson, P.**

Introduction to Expert Systems.

Addison-Wesley, 1999.

❑ **Gonzalez, A. J. y Dankel, D. D.**

The Engineering of Knowledge Based Systems:
Theory and Practice

Prentice Hall, 1993.

❑ **Rowe, Neil C.**

Artificial Intelligence through Prolog.

Prentice-Hall, 1988.