Facsímil 12

Lógica e incertidumbre

Capítulo 01

Lógica para máquinas: proposicional, primer orden y resolución

686f6c61 · 2026

Capítulo 01: Lógica para máquinas: proposicional, primer orden y resolución

Entrando en el tema

Discutes con alguien sobre el horario de un vuelo. Tú dices: «Si el vuelo sale tarde, perdemos la conexión. Y el vuelo sale tarde». La otra persona, sin mirar nada más, concluye: «Entonces perdemos la conexión». Nadie ha consultado la pantalla de salidas. La conclusión no viene de los datos, viene de la forma del razonamiento. Da igual que hablemos de vuelos, facturas o reactores nucleares: si aceptas las dos primeras frases, la tercera es inevitable.

Esa inevitabilidad es lo que estudia la lógica. No le importa de qué hablas, le importa cómo encajan las afirmaciones. Y esa indiferencia al contenido es justo lo que la hace mecanizable: si una conclusión se sigue por pura forma, una máquina puede comprobarlo sin entender de qué va el asunto.

Este capítulo trata de cómo se le enseña a razonar a una máquina con garantías. No con plausibilidad, no con «suena bien», sino con la clase de certeza que tiene «2 + 2 = 4». Es la base de la IA simbólica, la rama que durante décadas fue *la* inteligencia artificial, y que hoy sigue sosteniendo cosas muy concretas: los solvers SAT del Facsímil 2, los validadores que aprueban o rechazan la salida de un modelo, las ontologías OWL que dicen qué se puede inferir de un grafo. Cuando un agente del Facsímil 5 decide que una acción está permitida, por debajo hay reglas que se evalúan con esta maquinaria.

Vamos a recorrer tres niveles. Primero la lógica proposicional: enunciados enteros que son verdaderos o falsos. Después la regla de resolución, el único mecanismo de inferencia que necesita una máquina para demostrar cualquier cosa demostrable. Y por último la lógica de primer orden, que añade objetos, relaciones y cuantificadores, y nos lleva directos a Prolog y a la programación lógica.

Lógica proposicional: enunciados que valen verdadero o falso

La pieza más pequeña de la lógica proposicional es la **proposición**: una afirmación que es verdadera o falsa, sin medias tintas. «Llueve», «el usuario es administrador», «el importe supera el límite». No nos interesa su contenido interno, solo su valor de verdad. Por eso usamos letras: p, q, r, o nombres más legibles como $admin$, $borrar$.

A partir de ahí, combinamos proposiciones con **conectivas**. Son cinco, y conviene conocerlas con precisión porque toda la maquinaria descansa en ellas.[1]

CONECTIVA	SÍMBOLO	LECTURA	EJEMPLO COTIDIANO
Negación	$\neg$	«no»	$\neg admin$: el usuario no es administrador.
Conjunción	$\wedge$	«y»	$admin \wedge activo$: es administrador y está activo.
Disyunción	$\vee$	«o» (inclusivo)	$email \vee banner$: sale por email, por banner o por ambos.
Implicación	$\rightarrow$	«si... entonces»	$admin \rightarrow borrar$: si es administrador, puede borrar.
Bicondicional	$\leftrightarrow$	«si y solo si»	$activo \leftrightarrow pagado$: activo exactamente cuando ha pagado.

La **sintaxis** dice qué cadenas están bien formadas: una variable es una fórmula; si φ es fórmula, $\neg\varphi$ también lo es; si φ y ψ lo son, también lo son $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ y $(\varphi \leftrightarrow \psi)$. Nada más es una fórmula. Esta definición recursiva es la misma idea que un árbol de sintaxis en un compilador: las fórmulas se construyen unas dentro de otras.

La **semántica** dice qué significan. Una **interpretación** (o asignación) es una función que da verdadero o falso a cada variable:

$$v : \{p_1, p_2, \ldots, p_n\} \rightarrow \{V, F\}$$

En palabras: fijar una interpretación es rellenar, una por una, las casillas de «verdadero» o «falso» de cada variable; es elegir uno de los mundos posibles entre los que la fórmula podría vivir.

SÍMBOLO	SIGNIFICADO	EJEMPLO
v	Interpretación: una forma concreta de fijar los valores.	$v(admin) = V$, $v(activo) = F$.
p_i	Variable proposicional.	$admin$, $borrar$.
$\{V, F\}$	Los dos valores de verdad posibles.	Verdadero o falso.

Dada una interpretación, el valor de cualquier fórmula se calcula de dentro hacia fuera siguiendo las tablas de cada conectiva. La conectiva que más sorprende al principio es la implicación: $admin \rightarrow borrar$ solo es **falsa** cuando $admin$ es verdadero y $borrar$ es falso. Si el usuario no es administrador, la promesa «si eres administrador, puedes borrar» no se ha roto: simplemente no aplica, y por convención la consideramos verdadera. Esa convención (la implicación material) es la que permite razonar con reglas sin tener que comprobar antes si se dan sus condiciones.

Tablas de verdad, tautologías y la diferencia entre validez y satisfacibilidad

Una **tabla de verdad** enumera todas las combinaciones posibles de valores de las variables y, para cada una, da el valor de la fórmula. Con n variables hay 2^n filas. Esta es la tabla con las cinco conectivas a la vez:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \to B$	$A \leftrightarrow B$
V	V	F	V	V	V	V
V	F	F	F	V	F	F
F	V	V	F	V	V	F
F	F	V	F	F	V	V

La tabla de verdad es el método más bruto y más seguro que existe: si quieres saber qué hace una fórmula, la calculas en sus cuatro, ocho o dieciséis filas. El problema es que crece de forma explosiva. Con 30 variables ya tienes más de mil millones de filas. Esa explosión es exactamente la frontera de complejidad de SAT que vimos en el Facsímil 2: decidir si una fórmula proposicional tiene algún modelo es el primer problema que se demostró NP-completo.[2]

Con la tabla en la mano podemos clasificar cualquier fórmula:

CLASE	DEFINICIÓN	EJEMPLO
Tautología (válida)	Verdadera en **todas** las interpretaciones.	$A \vee \neg A$: o llueve o no llueve.
Contradicción (insatisfacible)	Falsa en todas las interpretaciones.	$A \wedge \neg A$: llueve y no llueve.
Satisfacible	Verdadera en **al menos una** interpretación.	$A \wedge B$: basta con $A = V, B = V$.
Contingente	Verdadera en unas y falsa en otras.	$A \to B$: depende de A y B.

Aquí aparece una de las distinciones más importantes del capítulo, y la que más confusión genera:

- **Validez** pregunta: ¿es verdadera *siempre*? Es una propiedad fuerte. Una fórmula válida no aporta información sobre el mundo, porque se cumple pase lo que pase.

- **Satisfacibilidad** pregunta: ¿es verdadera *alguna vez*? Es una propiedad débil. Que algo sea satisfacible solo dice que no es contradictorio.

Las dos están conectadas por la negación, y esta equivalencia es el puente que usan los demostradores automáticos:

$$\varphi \text{ es válida} \iff \neg\varphi \text{ es insatisfacible}$$

En palabras: demostrar que algo se cumple pase lo que pase es exactamente lo mismo que demostrar que su contrario no puede cumplirse en ningún escenario.

Léelo despacio, porque es la idea sobre la que se construye todo lo que viene después. Si quiero demostrar que algo es verdadero siempre, me basta con demostrar que su negación no puede ser verdadera nunca. En lugar de comprobar infinitas (o exponencialmente muchas) interpretaciones buenas, busco una sola contradicción. Un solver SAT, que solo sabe responder «satisfacible» o «insatisfacible», se convierte así en un demostrador de teoremas: para probar φ, le pides que intente satisfacer $\neg\varphi$; si fracasa, φ queda demostrada.

Pruébalo aquí mismo: calculadora de tablas de verdad

No hace falta esperar al cuaderno para ver esto funcionar. Escribe una fórmula y la tabla aparece al instante, con su clasificación: **tautología** (verdadera en todas las filas), **contradicción** (falsa en todas) o **contingente** (depende de los valores). Prueba con `p ∨ ¬p` (tautología), `p ∧ ¬p` (contradicción) o el clásico `(p -> q) & (q -> r)`.

Cambia la fórmula y observa: cuando algo es una tautología, ninguna fila de la última columna es falsa; cuando lo niegas, ninguna es verdadera. Esa es, en miniatura, la maquinaria que un *solver* SAT hace a gran escala sin enumerar todas las filas.

Equivalencias: reescribir sin cambiar el significado

Dos fórmulas son **equivalentes** ($\equiv$) si tienen la misma tabla de verdad. Equivalente no significa idéntico: significa intercambiable, como $\frac{2}{4}$ y $\frac{1}{2}$. Estas son las equivalencias que más se usan al preparar fórmulas para una máquina:

NOMBRE	EQUIVALENCIA	PARA QUÉ SIRVE
Eliminar implicación	$A \to B \;\equiv\; \neg A \lor B$	Quitar el $\to$, que un solver no consume directamente.
Contrapositiva	$A \to B \;\equiv\; \neg B \to \neg A$	Razonar hacia atrás (es la base de modus tollens).
Doble negación	$\neg\neg A \;\equiv\; A$	Simplificar.
De Morgan	$\neg(A \land B) \;\equiv\; \neg A \lor \neg B$	Empujar la negación hacia dentro.
De Morgan	$\neg(A \lor B) \;\equiv\; \neg A \land \neg B$	Lo mismo, dual.
Distributiva	$A \lor (B \land C) \;\equiv\; (A \lor B) \land (A \lor C)$	Convertir a forma normal conjuntiva.

Forma normal conjuntiva: el formato que las máquinas entienden

Una máquina no quiere fórmulas con paréntesis anidados y conectivas mezcladas. Quiere un formato plano y uniforme. Ese formato es la **forma normal conjuntiva** (CNF, por sus siglas en inglés): una conjunción de cláusulas, donde cada cláusula es una disyunción de literales, y un literal es una variable o su negación.

$$\varphi = \bigwedge_{j=1}^{m} C_j, \qquad C_j = \bigvee_{\ell \in L_j} \ell, \qquad \ell \in \{p_i, \neg p_i\}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\bigwedge_{j=1}^{m}$	Conjunción de las m cláusulas: todas deben cumplirse a la vez.	Las dos cláusulas de la regla de permisos juntas.
C_j	La cláusula número j, una disyunción de literales.	$(\neg admin \lor borrar)$.
$\bigvee_{\ell \in L_j}$	Disyunción dentro de la cláusula: basta con un literal verdadero.	$borrar$ o $\neg admin$.
ℓ	Un literal: una variable o su negación.	$admin$, $\neg admin$.

En palabras: una fórmula en forma normal conjuntiva es una lista de exigencias unidas por «y», donde cada exigencia es un menú de opciones unidas por «o»; se cumple solo si en cada cláusula al menos un literal sale verdadero.

Es exactamente la misma CNF del capítulo de SAT del Facsímil 2: una lista de cláusulas que deben cumplirse todas, donde cada cláusula se satisface si al menos uno de sus literales es verdadero. Toda fórmula proposicional se puede llevar a CNF con tres pasos mecánicos:

1. Eliminar implicaciones y bicondicionales con las equivalencias de arriba.
2. Empujar las negaciones hasta pegarlas a las variables, usando De Morgan y doble negación.
3. Distribuir $\vee$ sobre $\wedge$ hasta que la fórmula sea una conjunción de disyunciones.

Veámoslo con una regla de permisos. Queremos pasar a CNF la frase «si Ana es administradora, entonces puede borrar y queda registrada en auditoría»:

$$admin \to (borrar \wedge auditar)$$

PASO	RESULTADO	QUÉ HEMOS HECHO
Partida	$admin \to (borrar \wedge auditar)$	La regla original.
Quitar $\to$	$\neg admin \vee (borrar \wedge auditar)$	Aplicamos $A \to B \equiv \neg A \vee B$.
Distribuir	$(\neg admin \vee borrar) \wedge (\neg admin \vee auditar)$	Repartimos el $\vee$ sobre el $\wedge$.

El resultado son dos cláusulas: $(\neg admin \vee borrar)$ y $(\neg admin \vee auditar)$. Una sola regla de negocio se ha convertido en dos cláusulas limpias, listas para un solver o para la regla de resolución que viene ahora.

Inferencia: de las reglas a la resolución

Hasta aquí hemos descrito fórmulas. Pero el objetivo de la lógica es **inferir**: a partir de unas premisas que aceptamos, obtener conclusiones que estábamos obligados a aceptar. La relación clave se llama **consecuencia lógica** y se escribe con la doble barra:

$$\Gamma \models \psi$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
Γ	El conjunto de premisas que aceptamos como verdaderas.	$\{admin,\ admin \to borrar\}$.
$\models$	«tiene como consecuencia lógica»: la doble barra.	Lo que separa premisas de conclusión.
ψ	La conclusión que queremos justificar.	$borrar$.

En palabras: siempre que todas las premisas sean verdaderas, la conclusión también lo es; no existe ningún escenario que cumpla las premisas y falle la conclusión.

Significa: en toda interpretación que hace verdaderas todas las fórmulas del conjunto Γ, también ψ es verdadera. No dice «ψ es verdadera», dice «ψ es verdadera siempre que lo sean las premisas». Es el «entonces perdemos la conexión» del principio del capítulo: no afirmamos nada sobre el mundo, afirmamos que la conclusión va incluida en lo que ya hemos aceptado.

Reglas clásicas: modus ponens y modus tollens

Durante siglos, razonar fue aplicar reglas con nombre. Dos de ellas siguen siendo el pan de cada día:

REGLA	FORMA	LECTURA
Modus ponens	De A y $A \to B$, deriva B.	«Es administrador; si es administrador puede borrar; luego puede borrar.»
Modus tollens	De $\neg B$ y $A \to B$, deriva $\neg A$.	«No quedó registrado en auditoría; si fuera administrador habría quedado; luego no es administrador.»

Modus ponens razona hacia delante; modus tollens, hacia atrás por la contrapositiva. Son válidas y útiles, pero tienen un problema para una máquina: hay decenas de reglas de este estilo, y elegir cuál aplicar en cada momento es un arte. Sería mucho mejor disponer de **una sola regla** que, aplicada mecánicamente, baste para demostrar cualquier consecuencia. Esa regla existe.

La regla de resolución

En 1965, John Alan Robinson publicó una regla de inferencia pensada para máquinas, no para humanos.[3] La idea es de una simplicidad desconcertante. Si en una cláusula aparece un literal ℓ y en otra aparece su complementario $\neg\ell$, esos dos no pueden ser verdaderos a la vez, así que se cancelan, y el resto de ambas cláusulas se une en una cláusula nueva:

$$\frac{(\ell \vee A) \qquad (\neg\ell \vee B)}{(A \vee B)}$$

SÍMBOLO	SIGNIFICADO
ℓ	Literal que aparece afirmado en una cláusula.
$\neg\ell$	Su complementario, que aparece en la otra.
A, B	El resto de literales de cada cláusula (pueden estar vacíos).
$(A \vee B)$	El **resolvente**: la cláusula nueva que se deriva.

En palabras: si una cláusula afirma algo y otra lo niega, ese algo no puede inclinar la balanza, así que se tacha y los literales que quedan en ambas se juntan en una cláusula nueva.

Un caso particular merece nombre propio. Cuando una de las cláusulas es un solo literal, la regla se llama **resolución unitaria**, y es justo la propagación unitaria de DPLL del Facsímil 2: de (ℓ) y $(\neg\ell \vee B)$ se obtiene (B). Modus ponens, de hecho, no es más que un caso de resolución: A es la cláusula (A), $A \to B$ es la cláusula $(\neg A \vee B)$, y al resolverlas sale (B). Una sola regla absorbe a las clásicas.

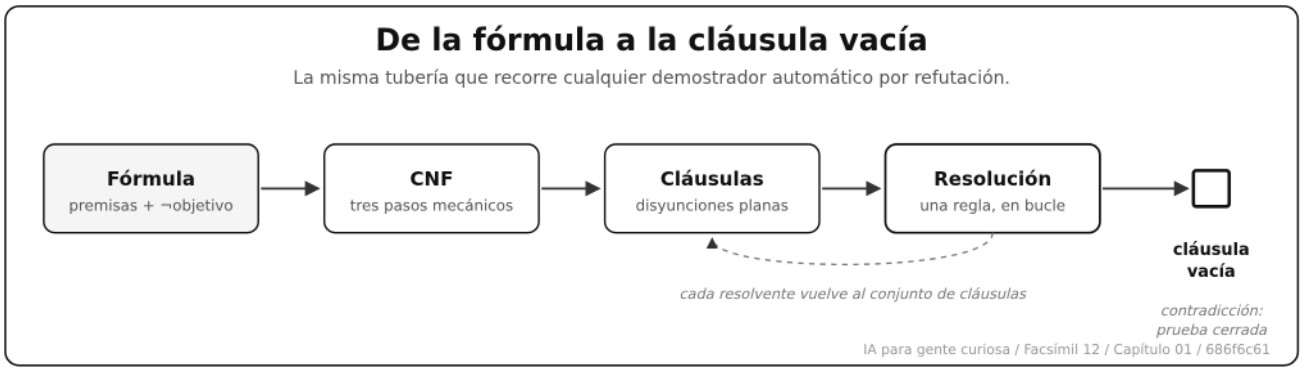
¿Y qué pasa cuando resolvemos (ℓ) con $(\neg\ell)$, dos literales solos y opuestos? No queda nada. El resolvente es la **cláusula vacía**, que se escribe como un cuadro vacío. Una cláusula es verdadera si alguno de sus literales lo es; la cláusula vacía no tiene literales, así que no puede ser verdadera nunca. Es la contradicción hecha símbolo. Encontrarla es la señal de que algo en nuestro conjunto de cláusulas es imposible.

Demostrar por refutación

Esto enlaza con el puente que vimos antes (validez es insatisfacibilidad de la negación) y da el método estrella de la lógica computacional: la **prueba por refutación**. Para demostrar que ψ se sigue de unas premisas Γ, no intentamos derivar ψ directamente. Hacemos lo contrario:[4]

1. Suponemos lo contrario de lo que queremos probar: añadimos $\neg\psi$ a las premisas.
2. Pasamos todo a CNF: $\Gamma \wedge \neg\psi$ se convierte en un conjunto de cláusulas.
3. Aplicamos resolución una y otra vez, derivando resolventes.
4. Si llegamos a la cláusula vacía, hemos encontrado una contradicción. Eso significa que $\Gamma \wedge \neg\psi$ es insatisfacible, y por tanto $\Gamma \models \psi$.

Visto de lejos, todo el método es una sola tubería: una fórmula entra por un extremo y, si la conclusión se seguía de verdad, sale por el otro convertida en la cláusula vacía.



Suena a truco, pero es la forma de razonar de todo demostrador automático serio. Tomemos un ejemplo concreto, del mundo de los permisos que tanto aparece en el facsímil. Las premisas:

- «Si Ana es administradora, puede borrar»: $admin \to borrar$, es decir la cláusula $(\neg admin \lor borrar)$.
- «Si puede borrar, la acción queda registrada en auditoría»: $borrar \to auditar$, es decir $(\neg borrar \lor auditar)$.
- «Ana es administradora»: $(admin)$.

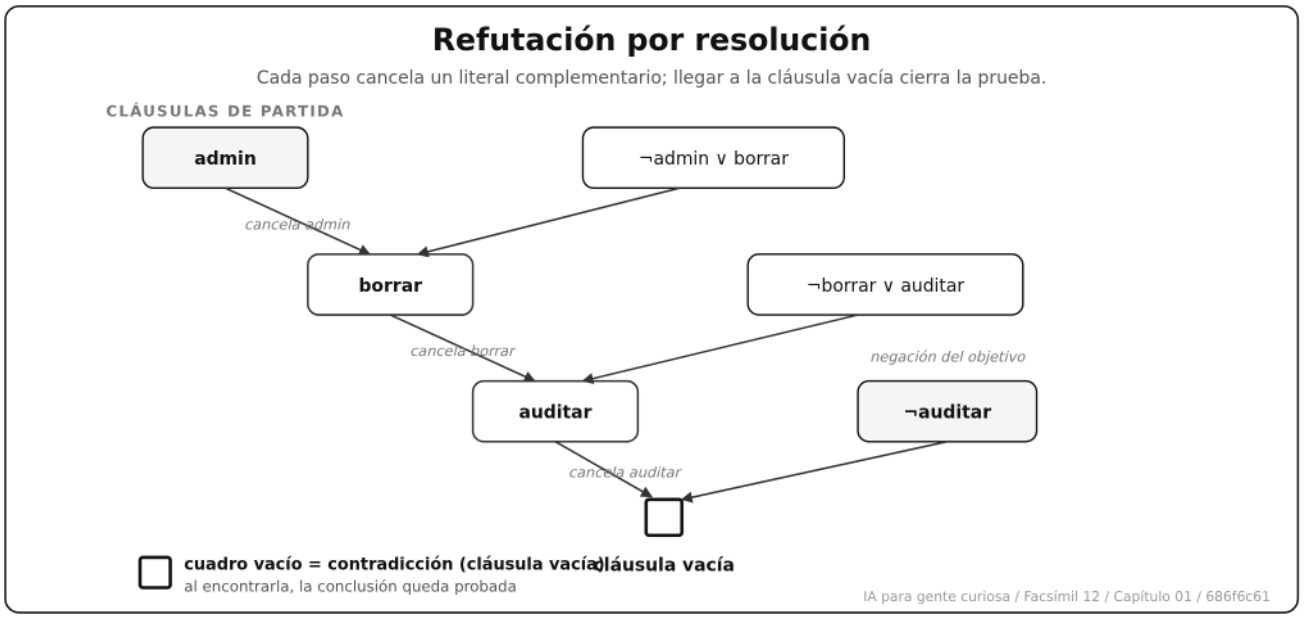
Queremos demostrar: $auditar$, o sea, que la acción quedará registrada. Negamos la conclusión y añadimos $(\neg auditar)$. El conjunto de cláusulas es:

$$C_1 = (\neg admin \lor borrar), \quad C_2 = (\neg borrar \lor auditar), \quad C_3 = (admin), \quad C_4 = (\neg auditar)$$

Y la refutación va así, paso a paso:

PASO	RESOLVEMOS	SOBRE EL LITERAL	RESOLVENTE
1	C_3 y C_1	$admin$	$(borrar)$
2	paso 1 y C_2	$borrar$	$(auditar)$
3	paso 2 y C_4	$auditar$	(vacía)

Hemos llegado a la cláusula vacía. La suposición «la acción no queda registrada» era incompatible con las premisas, así que la conclusión queda demostrada: la acción **quedará** registrada en auditoría. El árbol de esta refutación es el siguiente.



Corrección y completitud

¿Por qué fiarse de este procedimiento? Por dos propiedades que cualquier mecanismo de inferencia debe ganarse:

- **Corrección** (en inglés *soundness*): la regla nunca miente. Todo lo que la resolución deriva es de verdad una consecuencia lógica de las premisas. Si llega a la cláusula vacía, el conjunto era de verdad insatisfacible. Una máquina correcta jamás demuestra algo falso.
- **Completitud** (*completeness*): la regla no se deja nada por demostrar. Si un conjunto de cláusulas es insatisfacible, la resolución es capaz de derivar la cláusula vacía a partir de él. Esto se conoce como **completitud refutacional**, y es el resultado profundo del trabajo de Robinson.

Juntas dicen algo extraordinario: con esta única regla, aplicada de forma mecánica, una máquina puede demostrar exactamente las consecuencias lógicas verdaderas, ni una más ni una menos. No necesita ingenio, no necesita elegir entre decenas de reglas. Necesita combinar cláusulas hasta encontrar la vacía. Esa es la promesa de la demostración automática de teoremas, y su raíz histórica está en el algoritmo de Davis, Putnam, Logemann y Loveland, anterior a la resolución y todavía vivo en el corazón de los solvers SAT.[5]

Lógica de primer orden: objetos, relaciones y cuantificadores

La lógica proposicional tiene un techo bajo. Imagina la frase «todos los administradores pueden borrar». En proposicional, «Ana es administradora», «Luis es administrador» y «Marta es administradora» son tres variables sin ninguna relación entre sí. No hay forma de

decir «todos» de golpe: tendrías que escribir una regla por persona, y si mañana entra alguien nuevo, tu lógica no sabe nada de ella. La estructura interna de los enunciados (que hablan de *objetos* que tienen *propiedades* y mantienen *relaciones*) se pierde por completo.

La **lógica de primer orden** (FOL) rompe ese techo. Distingue piezas dentro de cada enunciado:

PIEZA	QUÉ ES	EJEMPLO
Constantes	Objetos concretos del dominio.	ana, $factura9$.
Variables	Objetos cualesquiera, sin fijar.	x, y.
Predicados	Propiedades o relaciones, verdaderas o falsas.	$Admin(x)$, $PuedeBorrar(x)$, $Aprueba(x, y)$.
Funciones	Aplicaciones que devuelven un objeto a partir de otros.	$jefe(x)$, $madre(x)$.
Cuantificadores	Alcance sobre los objetos.	$\forall$ («para todo»), $\exists$ («existe»).

Con esto, «todos los administradores pueden borrar» se escribe de una vez y vale para Ana, para Luis y para quien entre mañana:

$$\forall x \left(Admin(x) \rightarrow PuedeBorrar(x) \right)$$

En palabras: sea quien sea ese objeto x, si resulta ser administrador entonces puede borrar; la regla vale para todos a la vez, también para quien entre mañana.

Y «existe al menos un administrador» se escribe:

$$\exists x \, Admin(x)$$

En palabras: hay como mínimo un objeto del que es cierto que es administrador, aunque no digamos cuál.

Los dos cuantificadores son duales, conectados por la negación de la misma forma que las leyes de De Morgan conectaban $\wedge$ y $\vee$: «no todos cumplen P» es lo mismo que «existe alguno que no cumple P».

$$\neg \forall x \, P(x) \;\equiv\; \exists x \, \neg P(x) \qquad\qquad \neg \exists x \, P(x) \;\equiv\; \forall x \, \neg P(x)$$

En palabras: negar un «para todo» equivale a afirmar que existe al menos una excepción; negar un «existe» equivale a afirmar que la propiedad falla para todos sin excepción.

Esta capacidad expresiva es justo lo que hay debajo de las ontologías OWL y las lógicas de descripción del Facsímil 2: cuando declaras que «toda factura pertenece a exactamente un cliente», estás escribiendo una fórmula cuantificada de primer orden (restringida, para que el razonamiento siga siendo tratable). FOL es el lenguaje madre de toda esa familia.[6]

Skolemización: deshacerse de «existe»

La resolución trabaja con cláusulas, y las cláusulas no tienen cuantificadores explícitos. Para aplicar resolución en primer orden hay que preparar las fórmulas, y el paso más curioso es la **skolemización**: eliminar los cuantificadores existenciales sustituyendo la variable que afirman que existe por un objeto concreto.

La intuición es esta. Si digo $\exists x\, Admin(x)$, «existe un administrador», puedo darle un nombre a ese administrador que sé que existe: lo llamo a_0 (una **constante de Skolem**) y reescribo $Admin(a_0)$. Si el «existe» está dentro de un «para todo» (cada empleado tiene un jefe: $\forall x\, \exists y\, Jefe(y, x)$), el objeto que existe depende de x, así que en lugar de una constante uso una **función** de Skolem: $\forall x\, Jefe(jefe(x), x)$. No estamos inventando información: solo le ponemos nombre a algo cuya existencia ya habíamos afirmado. Tras skolemizar, todos los cuantificadores son universales y se pueden dar por supuestos, y la fórmula se lleva a cláusulas como en proposicional.

Unificación: el motor del primer orden

Aquí está la pieza que de verdad distingue la resolución de primer orden. En proposicional, dos literales se cancelaban si uno era exactamente el complementario del otro. En primer orden, los literales tienen variables, y casi nunca coinciden palabra por palabra. $Admin(x)$ y $Admin(ana)$ no son idénticos, pero está claro que el primero «encaja» con el segundo si hacemos que x sea ana. Encontrar ese encaje es la **unificación**: buscar una sustitución de variables que iguale dos expresiones.

Una **sustitución** se escribe como un conjunto de reemplazos, por ejemplo $\sigma = \{x/ana,\ y/jefe(ana)\}$, que se lee «pon ana donde haya x y $jefe(ana)$ donde haya y». El algoritmo de unificación recorre las dos expresiones en paralelo:

CASO	QUÉ HACE
Dos constantes iguales	Siguen adelante; no hace falta sustituir nada.
Dos constantes distintas	Fallo: ana y $luis$ no se pueden igualar.
Una variable y un término	Sustituye la variable por el término (si la variable no aparece dentro del término, lo que se llama *comprobación de ocurrencia*).
Dos predicados o funciones	Deben tener el mismo nombre y aridad; entonces unifica argumento a argumento.

Un ejemplo trabajado. Queremos unificar:

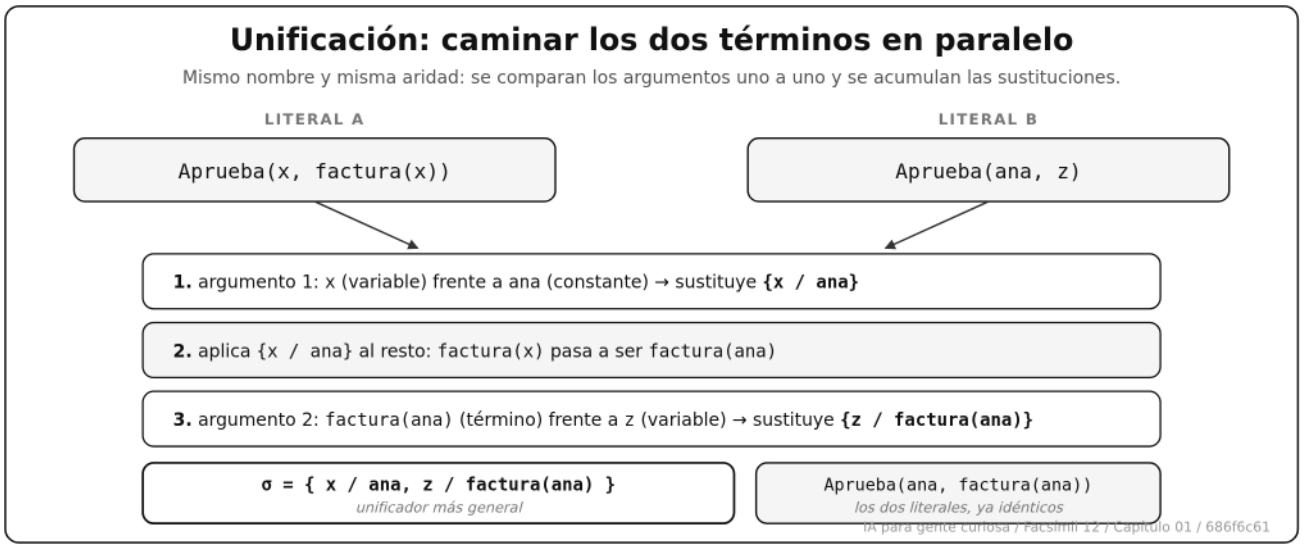
$$Aprueba(x,\ factura(x)) \qquad \text{y} \qquad Aprueba(ana,\ z)$$

Recorremos los argumentos. El primero: x frente a ana, variable contra constante, sustituimos $\{x/ana\}$. Aplicamos esa sustitución al resto, con lo que $factura(x)$ se vuelve $factura(ana)$. El segundo argumento: $factura(ana)$ frente a z, sustituimos $\{z/factura(ana)\}$. El resultado es el **unificador**:

$$\sigma = \{\, x/ana,\ \ z/factura(ana) \,\}$$

Aplicado a ambos literales, los dos se vuelven $Aprueba(ana,\ factura(ana))$: idénticos. El algoritmo siempre encuentra, si existe, el **unificador más general**, el que sustituye lo mínimo imprescindible y deja todo lo demás libre. Esa generalidad es lo que hace que una sola derivación valga para infinitos casos concretos.

El recorrido completo de ese ejemplo, argumento a argumento, es el siguiente.



Resolución de primer orden

Con la unificación, la regla de resolución se generaliza casi sola: para cancelar dos literales complementarios, primero los unificamos y luego aplicamos la sustitución al resolvente. El ejemplo canónico, con más de dos mil años de antigüedad, es la mortalidad de Sócrates:

- «Todos los humanos son mortales»: $\forall x\,(Humano(x) \to Mortal(x))$, que en cláusula es $(\neg Humano(x) \lor Mortal(x))$.
- «Sócrates es humano»: $(Humano(socrates))$.
- Queremos probar: $Mortal(socrates)$. Negamos: $(\neg Mortal(socrates))$.

Resolvemos. La cláusula $(\neg Humano(x) \lor Mortal(x))$ y $(Humano(socrates))$ se unifican con $\sigma = \{x/socrates\}$, cancelando $Humano$, y dan $(Mortal(socrates))$. Esta y $(\neg Mortal(socrates))$ se cancelan y dan la cláusula vacía. Sócrates es mortal. La estructura es idéntica a la refutación proposicional de antes; lo único nuevo es que la unificación ha hecho el trabajo de emparejar la regla general con el caso concreto.

Cláusulas de Horn y la conexión con Prolog

La resolución de primer orden es completa, pero buscar a ciegas entre todas las combinaciones de cláusulas es caro. Resulta que si restringimos un poco la forma de las cláusulas, el razonamiento se vuelve mucho más manejable y, además, se parece muchísimo a programar.

Una **cláusula de Horn** es una cláusula con **como mucho un literal positivo**. Suena técnico, pero tiene una lectura cotidiana preciosa. Una cláusula de Horn con exactamente un positivo, como $(\neg A \lor \neg B \lor C)$, es equivalente a la implicación $(A \land B) \to C$: «si se cumplen A y B, entonces C». Es decir, una regla. Y eso es exactamente lo que escribes cuando programas en Prolog, solo que del revés:

```
% C :- A, B.   se lee "C es cierto si A y B lo son"
puede_borrar(X) :- administrador(X), cuenta_activa(X).

administrador(ana).
cuenta_activa(ana).

% Pregunta:
?- puede_borrar(ana).
% Prolog responde: true
```

Prolog es, en el fondo, un motor de resolución sobre cláusulas de Horn.[7] Cuando le preguntas `?- puede_borrar(ana)`, niega el objetivo (como en la refutación) e intenta derivar la cláusula vacía encadenando reglas hacia atrás: para probar `puede_borrar(ana)` necesita `administrador(ana)` y `cuenta_activa(ana)`, que están como hechos, así que cierra la prueba. Esta estrategia dirigida por el objetivo se llama **resolución SLD**, y es lo que hace que Prolog sea a la vez una lógica y un lenguaje de programación: el mismo texto se lee como especificación declarativa («qué es verdad») y como programa («cómo calcularlo»).

La teoría que garantiza que esto funciona (que el significado lógico de un programa Prolog y su comportamiento al ejecutarse coinciden) es la base de la programación lógica.[8] Y hay un bono de eficiencia: decidir la satisfacibilidad de un conjunto de cláusulas de Horn (el problema HORN-SAT) se resuelve en tiempo lineal, no exponencial. Restringirse a Horn nos saca de la zona difícil de SAT que vimos en el Facsímil 2 y nos pone en terreno rápido y predecible. Por eso los motores de reglas, muchos sistemas expertos (que veremos en el capítulo 4 de este facsímil) y buena parte de la programación lógica viven cómodamente en este fragmento.

Los límites: qué puede y qué no puede una máquina lógica

Toda esta potencia tiene fronteras, y conocerlas es parte de usar la lógica con honestidad.

Decidibilidad. La lógica proposicional es **decidible**: existe un procedimiento que, para cualquier fórmula, termina siempre y responde sí o no (la tabla de verdad, o un solver SAT). Puede ser lento (NP-completo), pero acaba. La lógica de primer orden, en cambio, es solo **semidecidible**. Si una fórmula es de verdad una consecuencia lógica, la resolución terminará encontrando la prueba; pero si **no** lo es, el procedimiento puede no detenerse nunca, siguiendo generando resolventes sin llegar a conclusión. No es un defecto de implementación: se demostró que no puede existir ningún algoritmo que decida siempre la validez en primer orden. Hay un límite teórico, no de ingeniería.

Explosión combinatoria. Incluso dentro de lo decidible, el número de cláusulas que se pueden generar crece de forma brutal. La unificación produce resolventes nuevos constantemente, y sin estrategias de control un demostrador se ahoga en su propia

producción. La investigación en demostración automática es, en buena medida, la búsqueda de heurísticas para no explorar de más.

El frame problem. Cuando se intentó usar la lógica para que un agente razonara sobre acciones en el mundo (si muevo esta caja, ¿qué cambia y qué sigue igual?), apareció un problema sutil: hace falta declarar explícitamente todo lo que **no** cambia con cada acción, y eso es inabarcable.[9] Este *frame problem* fue uno de los grandes quebraderos de cabeza de la IA simbólica clásica y una de las razones por las que, para ciertos dominios, se buscaron enfoques distintos. Lo mencionamos porque enmarca por qué la lógica pura no se quedó como la única herramienta: convive con la probabilidad (el próximo capítulo) y con el aprendizaje.

En el día a día

Aunque no escribas ∀ ni la cláusula vacía en tu trabajo, esta maquinaria está funcionando por debajo de cosas que usas a diario.

Cada vez que un sistema de permisos decide si puedes hacer una acción, evalúa reglas: «si el rol es administrador y la cuenta está activa, permitir». Eso son cláusulas de Horn evaluándose. Cada vez que un validador comprueba que una salida cumple un esquema (un JSON con los campos obligatorios, los tipos correctos, las restricciones de cardinalidad), está aplicando lógica determinista, no plausibilidad. Cada vez que una ontología responde «esta entidad es un documento fiscal porque es una factura, y toda factura lo es», ha hecho una inferencia por subsunción, que es resolución disfrazada.

En el desarrollo de software, los verificadores formales que comprueban que un programa cumple su especificación usan demostradores SMT como Z3, que combinan resolución con teorías (aritmética, arrays, cadenas). Los compiladores optimizan usando análisis lógico. Y en sistemas con LLMs, la lógica es la red de seguridad: el modelo propone, pero una capa lógica verificable decide si la propuesta se ejecuta.

Por qué debería importarte

Porque la diferencia entre «suena correcto» y «es correcto» es exactamente la diferencia entre un modelo generativo y un demostrador lógico. Un LLM te puede dar una respuesta convincente y equivocada con total seguridad. Un razonador lógico correcto nunca afirma algo falso a partir de premisas verdaderas: si dice que algo se sigue, se sigue.

Esa garantía es escasa y valiosa. En todo el resto del facsímil aparecen sistemas que mezclan generación con verificación, y la parte de verificación, la que de verdad protege tus decisiones, casi siempre tiene una lógica determinista debajo. Entender resolución, refutación y la diferencia entre validez y satisfacibilidad te da el vocabulario para saber dónde poner una

garantía dura y dónde basta con una sugerencia blanda. Es la misma lección del Facsímil 2: cuando hay reglas que no se pueden violar, la aceptación no debe depender de si la respuesta suena bien.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Confundir validez con satisfacibilidad	Una fórmula satisfacible solo es «no imposible». Una válida es «siempre verdadera». Tratar la primera como la segunda lleva a aceptar cosas que solo se cumplen a veces.	Pregúntate: ¿quiero que se cumpla *siempre* (validez) o que sea *posible* (satisfacibilidad)? Recuerda el puente: válida equivale a que su negación sea insatisfacible.
Leer mal la implicación material	$A \to B$ es verdadera cuando A es falsa. Esperar que «si llueve, llevo paraguas» diga algo cuando no llueve es esperar de más.	Quédate con la única fila falsa: A verdadero y B falso. Solo ahí se rompe la promesa.
Intentar demostrar ψ directamente	Derivar la conclusión a pelo es difícil y poco sistemático.	Usa refutación: niega la conclusión, pásalo todo a CNF y busca la cláusula vacía. Es lo que hace una máquina.
Olvidar la comprobación de ocurrencia al unificar	Unificar x con $\mathit{factura}(x)$ crea un término infinito y rompe el algoritmo.	Antes de sustituir x por un término, comprueba que x no aparece dentro de ese término.
Creer que FOL siempre termina	Es solo semidecidible: si la conclusión no se sigue, el demostrador puede no parar nunca.	Pon límites de tiempo o de profundidad, y desconfía de un «todavía pensando» eterno.

Cómo encaja todo

Este capítulo parece un rincón teórico, pero en realidad es el suelo sobre el que se apoyan media docena de cosas que ya viste o que vas a ver en el resto del facsímil. Vale la pena seguir los hilos uno a uno, porque cada uno explica por qué la lógica no es una curiosidad histórica, sino una pieza que sigue trabajando por debajo.

El hilo más directo va al **Facsímil 2, capítulo 5**, donde aparecieron los solvers SAT. Allí la pregunta era «¿hay alguna forma de asignar valores que cumpla todas estas restricciones a la vez?»: cuadrar un horario sin solapes, colocar las piezas de un sudoku, repartir turnos respetando las reglas. Esa pregunta es, palabra por palabra, la **satisfacibilidad** de este

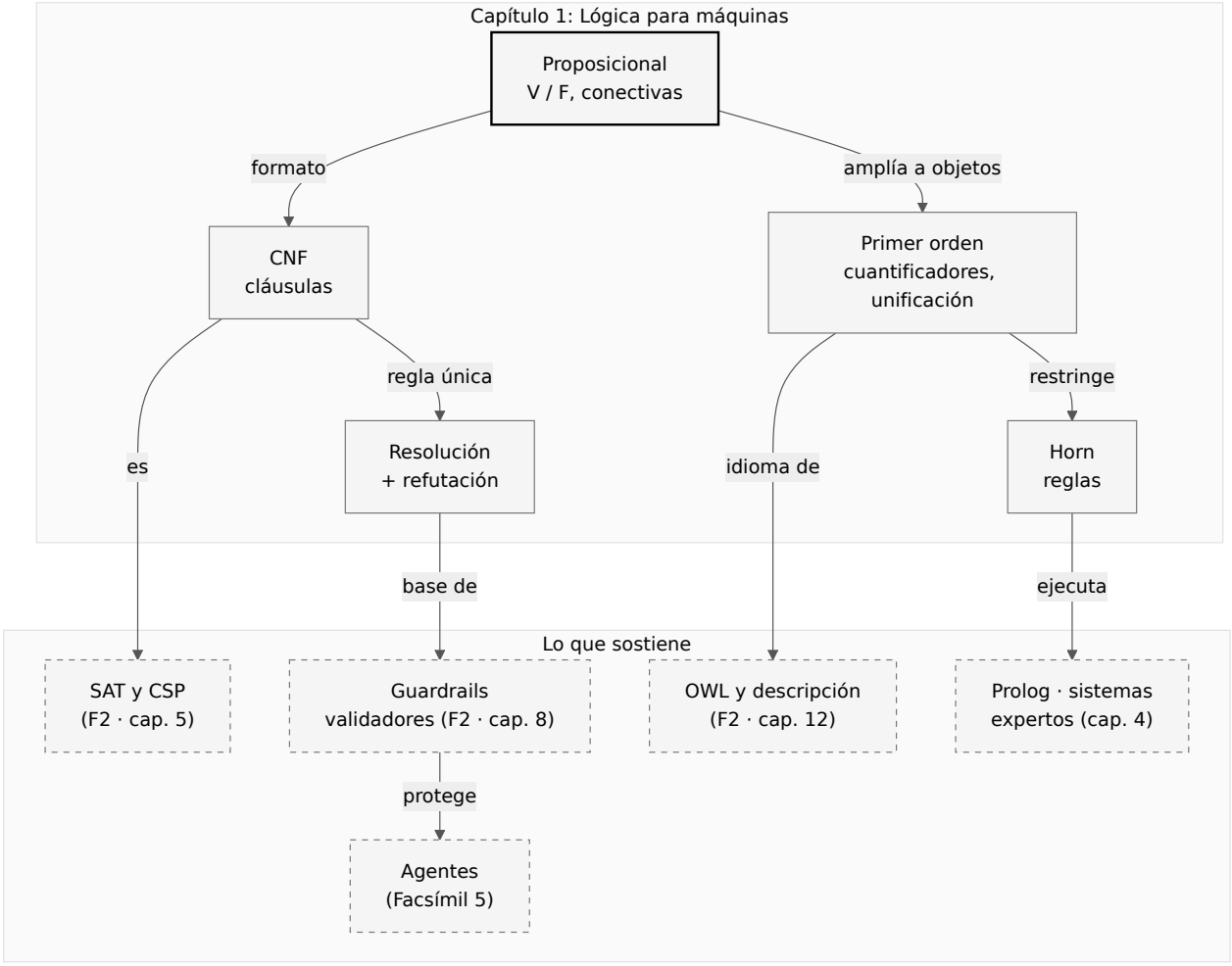
capítulo. Lo que cambia es el envoltorio: lo que el Facsímil 2 contaba como un problema de búsqueda, aquí se ve como lo que es por dentro, una fórmula en CNF a la que le buscamos un modelo. Y el puente «validez equivale a insatisfacibilidad de la negación» es justo lo que permite usar ese mismo solver no solo para encajar restricciones, sino para demostrar teoremas: le das la vuelta a la pregunta y la máquina te dice si algo se cumple siempre.

El segundo hilo lleva al **Facsímil 2, capítulo 8**, el de las restricciones como guardrails. Allí la idea era poner barreras duras alrededor de un sistema que genera texto: reglas que no son sugerencias, sino verjas que aceptan o rechazan una salida sin negociar. Eso es exactamente la resolución y la refutación funcionando como portero. Piensa en un asistente que redacta un correo a un cliente: el modelo propone el texto, pero antes de enviarlo una capa de reglas comprueba cosas que no pueden fallar, como que no haya un importe sin formato válido o que no se prometa un descuento no autorizado. Esa comprobación no «suena bien o mal», se cumple o no se cumple, y debajo hay cláusulas evaluándose con la maquinaria de aquí.

El tercer hilo, también en el **Facsímil 2, capítulo 12**, son las ontologías OWL y las lógicas de descripción. Cuando declaras que «toda factura pertenece a exactamente un cliente» o que «todo documento fiscal es un documento», estás escribiendo fórmulas de primer orden con cuantificadores, solo que con una sintaxis amable y recortada para que el razonamiento siga siendo rápido. Por eso, cuando una ontología deduce sola que una entidad concreta es un documento fiscal porque es una factura, no está adivinando: está haciendo la misma inferencia que vimos con Sócrates, una regla general que se aplica a un caso concreto mediante unificación. La lógica de descripción es primer orden domesticado, y este capítulo es su lengua materna.

El cuarto hilo salta al **Facsímil 5**, el de los agentes. Un agente que actúa en el mundo necesita saber qué tiene permitido hacer antes de hacerlo, y esos permisos no son intuiciones: son reglas. «Si el rol es administrador y la cuenta está activa, puede borrar» es una cláusula de Horn, idéntica a las que escribíamos en Prolog. Cada vez que el agente se plantea una acción, evalúa esas reglas y la respuesta es un sí o un no comprobable. La diferencia entre un agente en el que confías y uno que da miedo soltar suele estar justo ahí: en si sus permisos son lógica determinista o una corazonada del modelo.

Y el último hilo se queda en casa, en el **capítulo 4 de este mismo facsímil**, el del motor de inferencia. Lo que aquí hemos descrito como reglas y como una regla de resolución que se aplica en bucle, allí se convierte en la máquina que de verdad las ejecuta: un motor que toma hechos y reglas y va derivando conclusiones, encadenando hacia delante o hacia atrás como hace Prolog. Este capítulo te da el vocabulario (cláusula, resolvente, unificación, cláusula vacía); el capítulo 4 te enseña el motor que los pone a girar. Conviene leerlos seguidos, porque uno es la gramática y el otro, la conversación.


Capítulo 1: Lógica para máquinas
Proposicional
V / F, conectivas
formato
amplía a objetos
CNF
cláusulas
Primer orden
cuantificadores,
unificación
regla única
restringe
es
Resolución
+ refutación
idioma de
Horn
reglas
base de
ejecuta
Lo que sostiene
SAT y CSP
(F2 · cap. 5)
Guardrails
validadores (F2 · cap. 8)
OWL y descripción
(F2 · cap. 12)
Prolog · sistemas
expertos (cap. 4)
protege
Agentes
(Facsímil 5)

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Lógica proposicional	Lógica de enunciados que son verdaderos o falsos, combinados con conectivas.
Conectiva lógica	Operador que combina enunciados: negación, conjunción, disyunción, implicación y bicondicional.
Tabla de verdad	Tabla que da el valor de una fórmula para cada combinación de valores de sus variables.
Validez	Una fórmula es válida si es verdadera en toda interpretación (tautología).
Satisfacibilidad	Una fórmula es satisfacible si existe al menos una interpretación que la hace verdadera.
Forma normal conjuntiva	Conjunción de cláusulas, donde cada cláusula es una disyunción de literales.
Modus ponens	Regla de inferencia: de A y $A \to B$ se deriva B.
Resolución	Regla que, de dos cláusulas con un literal complementario, deriva una nueva cláusula.
Refutación	Probar que algo se deduce demostrando que su negación lleva a contradicción (cláusula vacía).
Lógica de primer orden	Lógica con predicados, funciones, variables y cuantificadores sobre objetos.
Cuantificador	Símbolo que expresa «para todo» (∀) o «existe» (∃).
Unificación	Proceso de encontrar una sustitución que iguala dos expresiones lógicas.
Cláusula de Horn	Cláusula con como mucho un literal positivo; base de Prolog y la programación lógica.
Completitud	Una regla de inferencia es completa si puede derivar toda consecuencia lógica verdadera.

Antes de pasar página

- [ ] ¿Sé construir la tabla de verdad de una fórmula con dos o tres variables y clasificarla? (Si no, vuelve a «Tablas de verdad, tautologías y la diferencia entre validez y satisfacibilidad».)

- [ ] ¿Distingo validez de satisfacibilidad y entiendo el puente entre ambas por la negación? (Si no, vuelve a la misma sección.)
- [ ] ¿Puedo pasar una implicación a CNF en tres pasos? (Si no, vuelve a «Forma normal conjuntiva».)
- [ ] ¿Sé aplicar la regla de resolución y explicar qué es la cláusula vacía? (Si no, vuelve a «La regla de resolución».)
- [ ] ¿Entiendo cómo una refutación demuestra una conclusión llegando a la cláusula vacía? (Si no, vuelve a «Demostrar por refutación».)
- [ ] ¿Veo por qué FOL expresa «todos» y «existe» y la proposicional no? (Si no, vuelve a «Lógica de primer orden».)
- [ ] ¿Puedo unificar dos literales y dar el unificador más general? (Si no, vuelve a «Unificación: el motor del primer orden».)
- [ ] ¿Reconozco una cláusula de Horn como una regla y su relación con Prolog? (Si no, vuelve a «Cláusulas de Horn y la conexión con Prolog».)

En resumen

IDEA FUERZA	DETALLE
La lógica razona por la forma, no por el contenido.	Por eso una conclusión válida es mecanizable: una máquina la comprueba sin entender el tema.
Validez e insatisfacibilidad son dos caras.	Demostrar que algo es siempre verdadero equivale a demostrar que su negación es imposible.
La resolución es una sola regla, correcta y completa.	Aplicada por refutación hasta la cláusula vacía, demuestra cualquier consecuencia lógica.
El primer orden añade objetos y cuantificadores.	Con unificación, una regla general se aplica a infinitos casos; es el idioma de OWL y Prolog.
Toda esta lógica sostiene la IA verificable del facsímil.	SAT, guardrails, ontologías y sistemas expertos descansan en proposicional, resolución y Horn.

Para saber más

Cook, S. A. (1971). The complexity of theorem-proving procedures. En *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (pp. 151-158). ACM. https://doi.org/10.1145/800157.805047

Davis, M., Logemann, G. y Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7), 394-397. https://doi.org/10.1145/368273.368557

Genesereth, M. R. y Nilsson, N. J. (1987). *Logical foundations of artificial intelligence*. Morgan Kaufmann.

Huth, M. y Ryan, M. (2004). *Logic in computer science: modelling and reasoning about systems* (2.ª ed.). Cambridge University Press.

Kowalski, R. (1979). *Logic for problem solving*. North-Holland.

Lloyd, J. W. (1987). *Foundations of logic programming* (2.ª ed.). Springer-Verlag.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1), 23-41. https://doi.org/10.1145/321250.321253

Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson.

Notas

1. Huth, M. y Ryan, M. (2004). *Logic in computer science: modelling and reasoning about systems* (2.ª ed.). Cambridge University Press. El capítulo inicial introduce con rigor la sintaxis y la semántica de la lógica proposicional, y es una de las exposiciones más limpias para informáticos. ↩

2. Cook, S. A. (1971). The complexity of theorem-proving procedures. En *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (pp. 151-158). ACM. https://doi.org/10.1145/800157.805047 El resultado de Cook es el que conecta la lógica proposicional con la teoría de la complejidad y justifica por qué la fuerza bruta de la tabla de verdad no escala. ↩

3. Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1), 23-41. https://doi.org/10.1145/321250.321253 Este artículo es el origen de la resolución; el propio título («una lógica orientada a máquinas») deja claro que el objetivo era la demostración automática, no la elegancia para el lector humano. ↩

4. Genesereth, M. R. y Nilsson, N. J. (1987). *Logical foundations of artificial intelligence*. Morgan Kaufmann. Su tratamiento de la resolución como procedimiento de refutación es el estándar en los textos de IA simbólica y conecta la regla con la representación del conocimiento. ↩

5. Davis, M., Logemann, G. y Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7), 394-397. https://doi.org/10.1145/368273.368557 El procedimiento DPLL, base de los solvers SAT modernos, comparte con la resolución la idea de razonar sobre cláusulas en CNF para decidir la insatisfacibilidad. ↩

6. Genesereth, M. R. y Nilsson, N. J. (1987). *Logical foundations of artificial intelligence*. Morgan Kaufmann. El libro desarrolla la lógica de primer orden precisamente como lenguaje de representación del conocimiento para la inteligencia artificial. ↩

7. Kowalski, R. (1979). *Logic for problem solving*. North-Holland. Kowalski formuló la idea de que la lógica de Horn puede leerse a la vez como especificación declarativa y como programa ejecutable, el principio que dio nacimiento a la programación lógica. ↩

8. Lloyd, J. W. (1987). *Foundations of logic programming* (2.ª ed.). Springer-Verlag. Es el tratado de referencia sobre la semántica de los programas lógicos y la corrección de la resolución SLD sobre cláusulas de Horn. ↩

9. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. Los capítulos dedicados a la lógica proposicional, de primer orden y a la inferencia tratan la decidibilidad, la resolución y el frame problem como temas centrales de la representación del conocimiento. ↩

Cuadernos para practicar

Escanea el código con el móvil para abrir el cuaderno en Google Colab y ejecutarlo sin instalar nada.

Lógica y resolución: razonar con reglas, no con datos

https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%C3%ADmil%2012/01-logica-y-resolucion.ipynb