Facsímil 02

Inteligencia clásica

Capítulo 12

Conocimiento simbólico y recapitulación

686f6c61 · 2026

Capítulo 12: Conocimiento simbólico y recapitulación

Entrando en el tema

Este facsímil empezó con una pregunta humilde: ¿cómo resuelve problemas la IA cuando el mundo se puede modelar como estados, acciones, restricciones y decisiones?

Hemos recorrido búsqueda, heurísticas, SAT, CSP, planificación, guardrails y juegos con otros actores. Todas esas piezas tienen algo en común: no viven solo de texto plausible. Necesitan estructura.

El conocimiento simbólico es la parte de la IA que intenta decir cosas explícitas sobre el mundo: qué entidades existen, cómo se relacionan, qué reglas valen, qué se puede inferir y qué pregunta exacta queremos hacer. No sustituye a los modelos neuronales. Los complementa.

Un LLM puede redactar una respuesta magnífica. Un vector store puede encontrar el fragmento parecido. Un grafo de conocimiento puede decir: “esta factura pertenece a este cliente, este cliente tiene este contrato, este contrato exige esta condición y por eso esta acción está permitida”.

Ese “por eso” es la pieza clave.

Cuando parecerse no basta

En un buscador semántico convertimos textos y preguntas en vectores y medimos su similitud. La fórmula habitual es la **similitud coseno**, y no es nueva: viene del modelo de espacio vectorial que Salton introdujo en los años setenta para recuperación de información, mucho antes de los embeddings neuronales.[1]

$$\mathrm{sim}(q, d) = \cos(q, d) = \frac{q \cdot d}{\|q\| \, \|d\|}$$

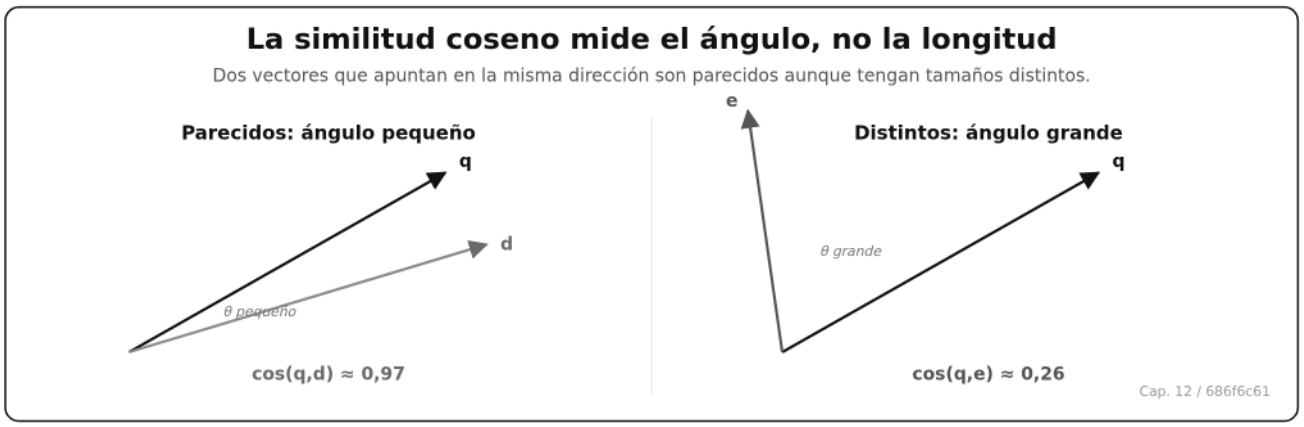
SÍMBOLO	SIGNIFICADO	LECTURA SENCILLA
q	Embedding de la pregunta.	La consulta convertida en vector.
d	Embedding del documento.	Un fragmento convertido en vector.
$q \cdot d$	Producto escalar.	Cuánto apuntan en dirección parecida.
$\|q\|$, $\|d\|$	Norma de cada vector.	Tamaño del vector.
$\cos(q, d)$	Similitud normalizada.	Cercanía semántica entre pregunta y documento.

La clave de ingeniería está en el denominador: dividir por las normas $\|q\|$ y $\|d\|$ **normaliza** los vectores, de modo que el coseno mide solo el *ángulo* entre ellos, no su longitud. Por eso un documento largo y uno corto que hablan de lo mismo salen parecidos: la magnitud no cuenta, solo la dirección. En código es directo:

```
import numpy as np

def coseno(q, d):
    return np.dot(q, d) / (np.linalg.norm(q) * np.linalg.norm(d))
```

En la práctica, un vector store no recalcula la norma en cada consulta: **normaliza los embeddings al guardarlos** (los deja de longitud 1), y entonces el coseno se reduce a un simple producto escalar $q \cdot d$. Ese producto se computa a gran velocidad sobre millones de vectores con índices aproximados como HNSW o IVF. Ese es el truco que hace viable la búsqueda semántica a escala.



Esto es potentísimo para recuperar contexto. Pero tiene un límite: la similitud no es una prueba.

PREGUNTA	UN VECTOR STORE PUEDE HACER	UN GRAFO PUEDE HACER
“Busca documentos parecidos a esta duda”	Recuperar fragmentos relacionados.	No es su punto fuerte.
“¿Qué servicios dependen de esta base de datos?”	Encontrar textos que lo mencionan.	Seguir relaciones `dependeDe`.
“¿Puede este usuario aprobar esta factura?”	Encontrar políticas similares.	Evaluar permisos, roles y umbrales.
“¿Por qué se tomó esta decisión?”	Mostrar fragmentos usados.	Devolver una cadena de hechos y reglas.

La diferencia no es estética. Es operativa. Cuando estás explorando, el parecido ayuda. Cuando estás tomando decisiones, la relación exacta importa.

RDF: hechos pequeños con identidad

RDF representa conocimiento como tripletas: sujeto, predicado y objeto.[2]

$$t = (s, p, o), \qquad \mathcal{G} = \{t_1, t_2, \ldots, t_n\}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
s	Sujeto.	`factura:f9`
p	Predicado o relación.	`perteneceA`
o	Objeto.	`cliente:c42`
t	Una tripleta.	Un hecho elemental.
$\mathcal{G}$	Grafo RDF.	Conjunto de tripletas.

Ejemplos cotidianos:

SUJETO	PREDICADO	OBJETO
`factura:f9`	`perteneceA`	`cliente:c42`
`factura:f9`	`importe`	`1280`
`cliente:c42`	`tienePlan`	`plan:empresa`
`servicio:api`	`dependeDe`	`servicio:db`
`servicio:db`	`almacena`	`tabla:facturas`
`persona:ana`	`tieneRol`	`rol:finanzas`

La parte importante no es solo el formato. Es la identidad. `cliente:c42` no es una palabra suelta. Es una referencia estable. Si otra tabla, documento, API o agente habla de `cliente:c42`, estamos hablando de la misma cosa.

Ese detalle evita un montón de niebla. “Ana”, “A. García”, “ana@empresa.com” y `persona:ana-garcia` pueden ser cuatro formas de referirse a una entidad. El conocimiento simbólico obliga a decidir cuándo son lo mismo y cuándo no.

RDFS y OWL: decir qué significa el grafo

RDF dice hechos. RDFS y OWL ayudan a decir qué significan esos hechos. Si RDF es “Ana trabaja en Finanzas”, RDFS y OWL son el vocabulario que permite entender qué es una persona, qué es un departamento, qué significa `trabajaEn` y qué consecuencias se derivan de esa relación.

No son grafos aparte. Son capas de significado encima del mismo grafo.

CAPA	QUÉ APORTA	PREGUNTA QUE RESPONDE
RDF	Hechos como tripletas.	¿Qué relaciones concretas existen?
RDFS	Clases, subclases, dominio y rango.	¿Qué tipo de cosas conectan esas relaciones?
OWL	Axiomas más expresivos para ontologías.	¿Qué reglas lógicas adicionales valen en el dominio?

RDFS es útil cuando quieres que el grafo tenga una gramática compartida. Permite declarar que `Factura` es una clase, que `Factura` es subclase de `DocumentoFiscal`, que `perteneceA` conecta documentos fiscales con clientes y que ciertos tipos se heredan.[3] OWL añade más

expresividad: clases disjuntas, equivalencias, propiedades inversas, restricciones de cardinalidad y axiomas para razonar con más precisión.[4]

Una forma rápida de verlo:

NECESIDAD	RDFS SUELE BASTAR	OWL EMPIEZA A TENER SENTIDO
Heredar tipos.	`Factura` subclase de `DocumentoFiscal`.	También, pero con más axiomas alrededor.
Decir qué conecta una relación.	`perteneceA` tiene dominio y rango.	Puedes añadir inversas o restricciones.
Evitar categorías incompatibles.	Limitado.	`Cliente` disjunto de `Servicio`.
Decir que dos clases son equivalentes.	Limitado.	`CompradorEmpresa` equivalente a cierta combinación de condiciones.
Expresar “exactamente uno”.	No es su fuerte.	Cardinalidad sobre una propiedad.

Una inferencia sencilla, la **subsunción** de clases que está en el corazón de las lógicas de descripción, la base formal de RDFS y OWL:[5]

$$\text{type}(x, C) \land C \sqsubseteq D \Rightarrow \text{type}(x, D)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\text{type}(x, C)$	x pertenece a la clase C.	`factura:f9` es `Factura`.
$C \sqsubseteq D$	C es subclase de D.	`Factura` subclase de `DocumentoFiscal`.
$\Rightarrow$	Podemos derivar.	`factura:f9` es `DocumentoFiscal`.

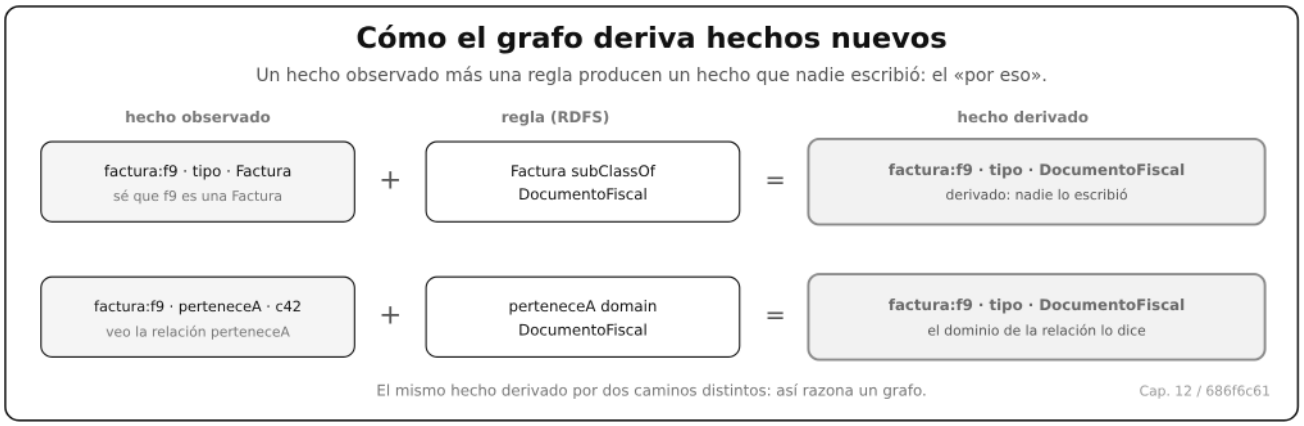
Esto parece pequeño, pero cambia cómo trabajamos. Si una regla aplica a todo `DocumentoFiscal`, también aplica a facturas, abonos o recibos que hereden de esa clase.

RDFS también permite inferir tipos desde el dominio y el rango de una propiedad. Son dos de las **reglas de implicación** (*entailment rules*) que define el propio estándar de RDF Schema:

$$(s, p, o) \land \text{domain}(p, C) \Rightarrow \text{type}(s, C)$$

$$(s, p, o) \land \text{range}(p, D) \Rightarrow \text{type}(o, D)$$

PIEZA	LECTURA SENCILLA	EJEMPLO
$\text{domain}(p, C)$	Quien usa la relación p como sujeto pertenece a C.	Si algo `perteneceA`, ese algo es `DocumentoFiscal`.
$\text{range}(p, D)$	Quien aparece como objeto de p pertenece a D.	Si algo recibe `perteneceA`, ese algo es `Cliente`.
(s, p, o)	Hecho observado.	`factura:f9 perteneceA cliente:c42`.
Tipo inferido	Conclusión derivada.	`factura:f9` es `DocumentoFiscal`; `cliente:c42` es `Cliente`.

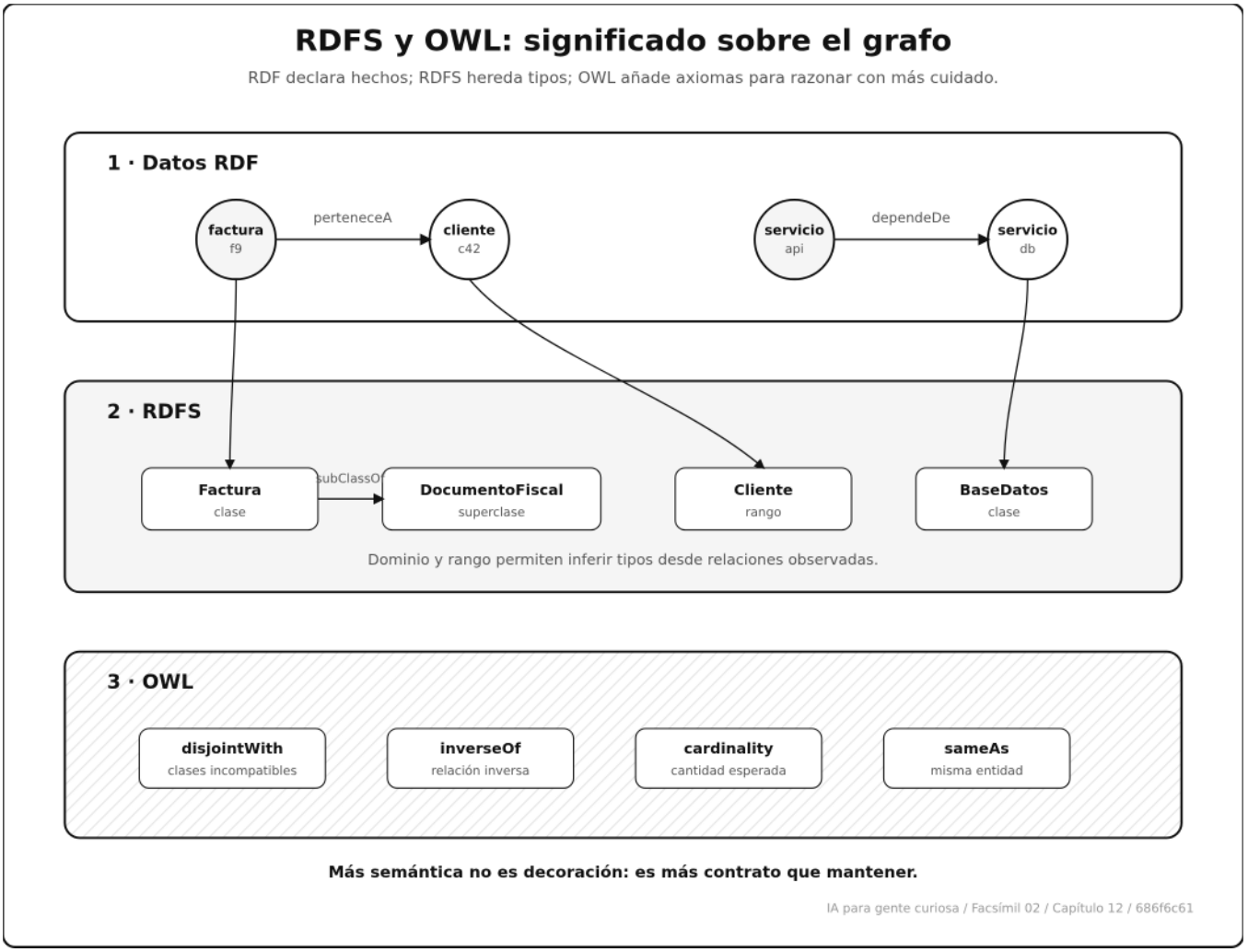


DECLARACIÓN	LECTURA HUMANA	UTILIDAD
`Factura subClassOf DocumentoFiscal`	Toda factura es un documento fiscal.	Reutilizar reglas.
`perteneceA domain DocumentoFiscal`	Si algo pertenece a un cliente, esperamos que sea documento fiscal.	Detectar modelado raro.
`perteneceA range Cliente`	El objeto de esa relación debe ser cliente.	Validar datos.
`Cliente disjointWith Servicio`	Una entidad no debería ser ambas cosas.	Evitar mezclas de dominio.

OWL conviene cuando la frase que quieres modelar ya no cabe bien en “subclase, dominio y rango”.

AXIOMA OWL	LECTURA HUMANA	EJEMPLO COMPRENSIBLE
`disjointWith`	Dos clases no deberían solaparse.	Un `Cliente` no es un `Servicio`.
`equivalentClass`	Dos descripciones nombran la misma clase.	`ClientePremium` equivale a cliente con plan empresa y contrato activo.
`inverseOf`	Una relación es la inversa de otra.	Si factura `perteneceA` cliente, cliente `tieneFactura` factura.
`cardinality`	Una relación debe aparecer cierto número de veces.	Una factura debería tener un único cliente responsable.
`sameAs`	Dos identificadores nombran la misma entidad.	`persona:ana` y `usuario:u17` son la misma persona.

La parte delicada: más expresividad también trae más responsabilidad. OWL razona con una lógica formal y, en muchos usos, con una mentalidad de mundo abierto: que no sepas un dato no significa que sea falso. Para producto, permisos o formularios, muchas veces conviene traducir las reglas críticas a validadores ejecutables además de modelarlas en la ontología.



La tentación es convertir la ontología en una catedral perfecta. Suele ser mala idea. Una ontología útil empieza pequeña: nombres claros, relaciones estables, restricciones que de verdad ayuden y ejemplos que el equipo entienda.

SHACL: cerrar el mundo cuando hace falta

Acabamos de ver que OWL razona con mundo abierto: que no conste un dato no lo hace falso. Eso es razonable para describir conocimiento, pero peligroso para validar. Si una factura debe tener exactamente un cliente, no te sirve un «quizá lo tiene»: quieres un error cuando no lo tiene.

Para eso existe **SHACL** (Shapes Constraint Language), el estándar del W3C para validar grafos RDF contra *formas* (shapes).[6] Una shape es una regla cerrada y comprobable, del estilo «toda `Factura` tiene exactamente un `perteneceA`, y su objeto es un `Cliente`». El validador recorre el grafo y devuelve un informe de conformidad: qué nodos cumplen y cuáles violan qué restricción.

OWL (MUNDO ABIERTO)	SHACL (VALIDACIÓN CERRADA)
Deriva hechos nuevos.	Comprueba que los hechos cumplen reglas.
«Puede que falte un dato.»	«Falta un dato obligatorio: error.»
Razonar y completar.	Validar y rechazar.
Ideal para inferencia.	Ideal para calidad de datos y permisos.

Así, la regla práctica del capítulo se vuelve concreta: modela el significado en la ontología, pero traduce las reglas críticas (cardinalidad, tipos obligatorios, permisos) a shapes SHACL o a validadores ejecutables. Una cosa describe el mundo; la otra protege tus decisiones.

SPARQL: preguntar por relaciones exactas

SPARQL es el lenguaje estándar para consultar grafos RDF.[7] En vez de pedir “texto parecido”, buscamos patrones de tripletas.

La forma mental más útil no es “SQL para grafos”, aunque el nombre se parezca. Es mejor leerlo así: dibujo un pequeño patrón con huecos, y el motor busca en el grafo todas las formas de rellenar esos huecos.

Un patrón mínimo:

$$q = \{(?x, \texttt{:dependeDe}, \texttt{:servicioDB})\}$$

La variable $?x$ se rellena con las entidades que encajan.

Formalmente, y esto lo fijó la semántica de SPARQL de Pérez, Arenas y Gutiérrez, una consulta de patrones devuelve asignaciones de variables (los *mappings* μ):[8]

$$\mathrm{Sol}(P, \mathcal{G}) = \{\mu \mid \forall t \in P,\ \mu(t) \in \mathcal{G}\}$$

SÍMBOLO	SIGNIFICADO	LECTURA SENCILLA
P	Patrón de tripletas.	Lo que preguntamos.
$\mathcal{G}$	Grafo RDF.	Los hechos disponibles.
μ	Asignación de variables.	Qué valor toma cada `?variable`.
$\mathrm{Sol}(P, \mathcal{G})$	Soluciones.	Filas de la tabla resultado.

```
SELECT ?factura WHERE {
  ?factura :perteneceA :clienteC42 .
  ?factura rdf:type :DocumentoFiscal .
}
```

Lectura humana: “dame las facturas que pertenecen al cliente C42 y además son documentos fiscales”.

La consulta se lee de dentro hacia fuera:

PARTE	QUÉ HACE	LECTURA HUMANA
`?factura`	Variable.	“Algo que todavía no sé”.
`:perteneceA :clienteC42`	Relación obligatoria.	Ese algo pertenece al cliente C42.
`rdf:type :DocumentoFiscal`	Tipo obligatorio.	Ese algo es documento fiscal.
`SELECT ?factura`	Proyección.	Devuelve solo la variable `?factura`.

Un ejemplo algo más realista:

```
PREFIX : <https://empresa.ejemplo/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?factura ?importe WHERE {
  ?factura rdf:type :DocumentoFiscal ;
           :perteneceA :clienteC42 ;
           :importe ?importe .

  FILTER(?importe > 1000)
}
ORDER BY DESC(?importe)
```

Lectura humana: “dame documentos fiscales del cliente C42 cuyo importe supere 1000, ordenados de mayor a menor”.

Hay tres detalles prácticos aquí:

DETALLE	QUÉ SIGNIFICA	POR QUÉ IMPORTA
`PREFIX`	Abrevia URIs largas.	Hace legible la consulta.
`;`	Reutiliza el mismo sujeto.	Evita repetir `?factura` tres veces.
`FILTER`	Restringe soluciones.	No basta con encajar: debe cumplir una condición.

`OPTIONAL` sirve cuando un dato ayuda, pero no debería eliminar la fila si falta:

```
SELECT ?factura ?importe ?fechaPago WHERE {
  ?factura rdf:type :DocumentoFiscal ;
           :perteneceA :clienteC42 ;
           :importe ?importe .

  OPTIONAL {
    ?factura :fechaPago ?fechaPago .
  }
}
```

Lectura humana: “dame las facturas y su importe; si existe fecha de pago, añádela, pero no descartes facturas sin fecha”.

Esto es muy importante en producto. Si usas una relación obligatoria cuando el dato es opcional, desaparecen resultados válidos. Si usas `OPTIONAL` para algo que de verdad debería existir, puedes ocultar un problema de calidad de datos.

SPARQL también tiene distintas formas de preguntar:

FORMA	DEVUELVE	CUÁNDO USARLA
`SELECT`	Tabla de variables.	Listar facturas, servicios, permisos.
`ASK`	`true` o `false`.	Comprobar si existe una relación.
`CONSTRUCT`	Nuevas tripletas RDF.	Crear una vista o grafo derivado.
`DESCRIBE`	Descripción de un recurso.	Explorar una entidad concreta.

Ejemplo con `ASK`:

```
ASK WHERE {
  :usuarioU17 :tienePermiso :permisoExportar .
  :permisoExportar :autorizaFuncion :exportarDatos .
}
```

Lectura humana: “¿tiene este usuario un permiso que autoriza exportar datos?”.

Ejemplo con `CONSTRUCT`:

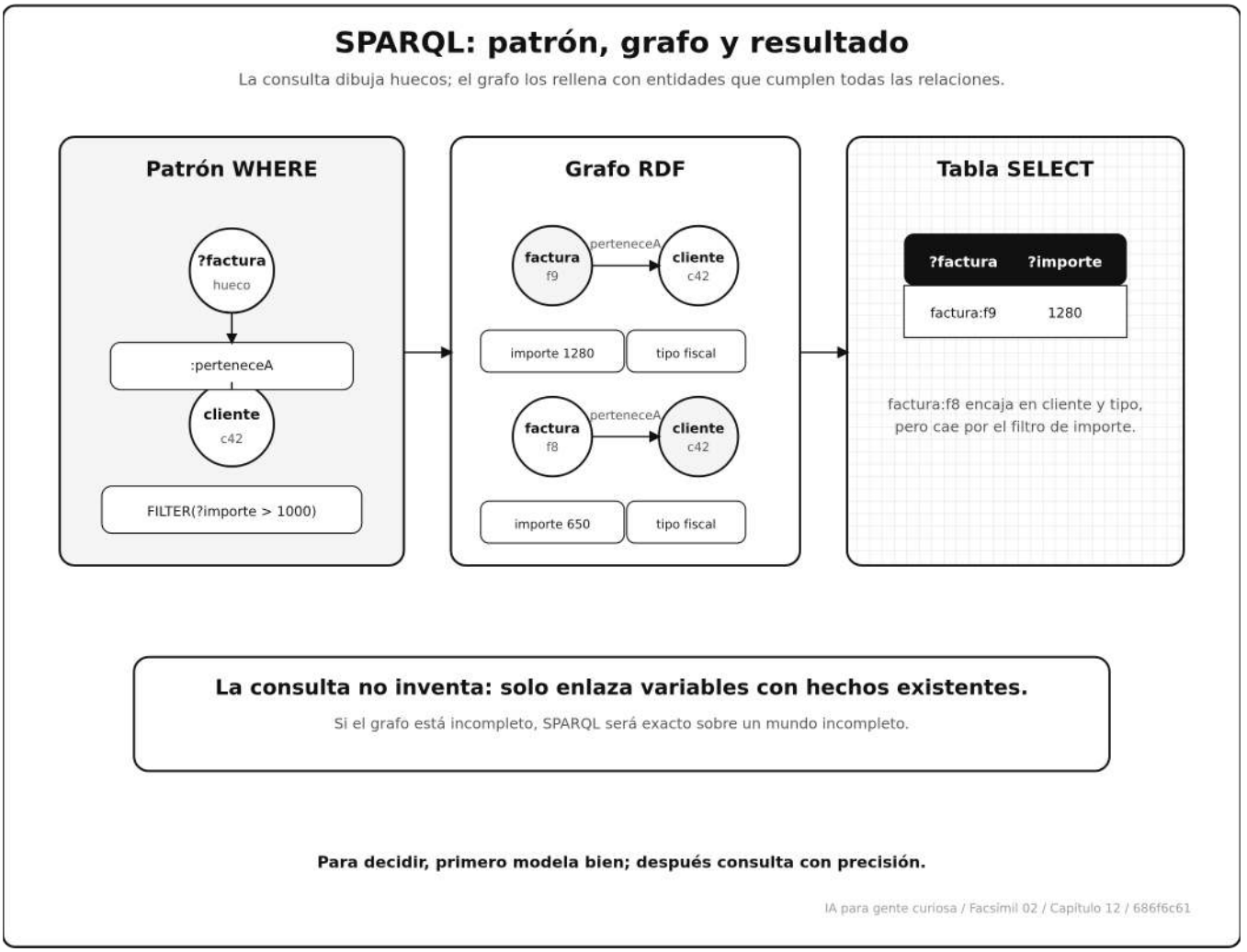
```
CONSTRUCT {
  ?servicio :afectadoPor :servicioDB .
}
WHERE {
  ?servicio :dependeDe+ :servicioDB .
}
```

Lectura humana: “construye nuevas tripletas diciendo qué servicios quedan afectados por `servicioDB`, siguiendo una o más dependencias”.

El `+` de `:dependeDe+` es un camino de propiedad. Permite seguir relaciones encadenadas:

EXPRESIÓN	LECTURA
`:dependeDe`	Una dependencia directa.
`:dependeDe+`	Una o más dependencias encadenadas.
`:dependeDe*`	Cero o más dependencias encadenadas.
`^:dependeDe`	La relación en sentido inverso.

En operaciones, esto es oro. Si `api` depende de `db`, y `web` depende de `api`, preguntar por `:dependeDe+ :db` permite encontrar tanto `api` como `web`.



Otro ejemplo:

```
SELECT ?servicio WHERE {
  ?servicio :dependeDe+ :servicioDB .
}
```

Lectura humana: “dime qué servicios dependen directa o indirectamente de esta base de datos”.

En sistemas con LLM, SPARQL suele entrar después de resolver entidades. El modelo puede entender que “la base de datos de facturación” se refiere a `:servicioDB`. Pero la consulta que decide qué servicios dependen de ella debería ser formal, trazable y repetible.

PASO	QUÉ HACE EL LLM	QUÉ HACE SPARQL
Entender la pregunta	Detecta intención y entidades candidatas.	No interpreta lenguaje natural.
Resolver entidad	Propone `:servicioDB`.	Usa la URI exacta.
Consultar relaciones	Puede explicar la consulta.	Devuelve coincidencias del grafo.
Responder	Redacta con contexto.	Aporta hechos y trazabilidad.

SPARQL no arregla un grafo pobre. Si falta una relación, no aparecerá. Si dos entidades están duplicadas, puede devolver resultados partidos. Si una propiedad se usa con significados distintos, la consulta será técnicamente correcta y semánticamente frágil.

Por eso SPARQL y ontología van juntas: una buena consulta depende de un buen vocabulario.

La diferencia con una búsqueda textual es clara:

BÚSQUEDA TEXTUAL	CONSULTA SIMBÓLICA
“Devuélveme textos donde parezca hablarse de dependencias.”	“Devuélveme entidades con relación `dependeDe` hacia `servicioDB`.”
Puede recuperar fragmentos útiles aunque usen otras palabras.	Devuelve coincidencias exactas del modelo.
Tolera ambigüedad.	Exige datos bien modelados.
Ideal para explorar.	Ideal para decidir, auditar y explicar.

Modo ingeniero: el motor en pocas líneas

Lo bonito del conocimiento simbólico es que el mecanismo cabe en poquísimo código. Un grafo es un conjunto de tripletas; inferir tipos por subclase es aplicar una regla hasta que no derive nada nuevo (un punto fijo); y consultar es buscar las tripletas que encajan con un patrón.

```
# El grafo es un conjunto de tripletas (sujeto, predicado, objeto).
grafo = {
    ("factura:f9", "tipo", "Factura"),
    ("factura:f9", "perteneceA", "cliente:c42"),
    ("Factura", "subClassOf", "DocumentoFiscal"),
}

def inferir_tipos(grafo):
    """Cierra los tipos por subclase: si x es C y C subClassOf D, x es D."""
    hechos = set(grafo)
    cambio = True
    while cambio:                       # repetir hasta que no se derive nada nuevo
        cambio = False
        for (x, p, c) in list(hechos):
            if p != "tipo":
                continue
            for (c2, p2, d) in list(hechos):
                if p2 == "subClassOf" and c2 == c and (x, "tipo", d) not in hechos:
                    hechos.add((x, "tipo", d))
                    cambio = True
    return hechos

def consultar(grafo, patron):
    """Resuelve el patron (?x, p, o): devuelve los sujetos que encajan."""
    _, p, o = patron
    return {s for (s, pred, obj) in grafo if pred == p and obj == o}

g = inferir_tipos(grafo)
consultar(g, ("?x", "tipo", "DocumentoFiscal"))   # -> {"factura:f9"}
```

Esto es, en miniatura, lo que hace el lab de este capítulo. Un motor real (RDFLib, un *triple store*, un razonador OWL) añade muchísimo más: caminos de propiedad, OPTIONAL, escala, optimización. Pero la idea de fondo es esta: hechos explícitos, reglas que derivan hechos nuevos y consultas que enlazan variables. No hay magia; por eso el resultado es trazable y explicable, justo lo que un vector store no te da.

Ontologías y sistemas expertos

Thomas Gruber definió una ontología como una especificación explícita de una conceptualización.[9] Dicho sin solemnidad: una ontología es el acuerdo sobre cómo vamos a nombrar y relacionar las cosas importantes.

Ese acuerdo es más profundo que un glosario. Un glosario define palabras. Una ontología define qué tipos de cosas existen, qué relaciones son válidas, qué restricciones importan y qué se puede inferir. Noy y McGuinness popularizaron una guía práctica para crear ontologías empezando por alcance, reutilización, clases, propiedades, restricciones e instancias.[10]

Una forma compacta de escribir esa definición, recogiendo las cuatro piezas que la literatura de ontologías considera básicas:

$$\mathcal{O} = (\mathcal{C}, \mathcal{P}, \mathcal{R}, \mathcal{A})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\mathcal{C}$	Clases del dominio.	`Cliente`, `Factura`, `Servicio`.
$\mathcal{P}$	Propiedades o relaciones.	`perteneceA`, `dependeDe`, `autoriza`.
$\mathcal{R}$	Restricciones.	“Una factura tiene un cliente responsable”.
$\mathcal{A}$	Axiomas.	Subclases, equivalencias, incompatibilidades.

Y la base de conocimiento es la unión del modelo y los datos. Es la división clásica de las lógicas de descripción entre **TBox** (el esquema, es decir, la ontología) y **ABox** (los hechos, es decir, las instancias):

$$\mathcal{KB} = \mathcal{O} \cup \mathcal{I}$$

PIEZA	QUÉ ES	EJEMPLO
$\mathcal{O}$	Ontología: el modelo del dominio.	Qué es una factura y cómo se relaciona con clientes.
$\mathcal{I}$	Instancias: datos concretos.	`factura:f9`, `cliente:c42`, `servicio:api`.
$\mathcal{KB}$	Base de conocimiento.	Modelo + datos + inferencias posibles.

La distinción importa porque una ontología sin instancias es un mapa vacío, y muchas instancias sin ontología son una habitación llena de etiquetas sueltas.

NO CONFUNDIR	QUÉ HACE	QUÉ LE FALTA
Glosario	Define términos.	Relaciones formales e inferencia.
Taxonomía	Ordena categorías.	Propiedades, restricciones y reglas.
Esquema de base de datos	Define tablas y columnas.	Semántica compartida entre sistemas.
JSON Schema	Valida forma de datos.	Significado del dominio y herencia.
Ontología	Modela conceptos, relaciones y restricciones.	Datos concretos si no se instancia.

Una ontología nace bien cuando empieza por preguntas de competencia: preguntas concretas que el sistema debería poder responder. No son preguntas decorativas; son el test de alcance.

PREGUNTA DE COMPETENCIA	QUÉ OBLIGA A MODELAR
¿Quién puede aprobar esta factura y por qué?	Persona, rol, factura, importe, política, autorización.
¿Qué servicios se ven afectados si cae esta base de datos?	Servicio, dependencia, equipo responsable, criticidad.
¿Qué documentos fiscales tiene este cliente?	Cliente, documento fiscal, relación de pertenencia.
¿Qué funciones incluye este plan de producto?	Plan, función, contrato, disponibilidad.
¿Qué regla explica que esta acción se escale a una persona?	Umbral, riesgo, aprobación, traza.

Si no puedes escribir cinco preguntas así, todavía no necesitas una ontología completa. Necesitas entender mejor el dominio.

El proceso práctico suele parecerse a esto:

1. Fijar alcance: qué preguntas debe responder y cuáles no.
2. Reutilizar vocabularios existentes cuando encajen.
3. Nombrar clases: cosas del dominio, no pantallas ni tablas.
4. Nombrar propiedades: verbos o relaciones estables.
5. Añadir restricciones: lo mínimo que evita ambigüedad peligrosa.
6. Crear instancias de ejemplo: casos reales, no juguetes perfectos.

7. Probar consultas: SPARQL, reglas o validadores.
8. Revisar con personas del dominio: contabilidad, legal, soporte, producto, operaciones.
9. Versionar cambios: una ontología viva necesita dueño y criterio de evolución.

DOMINIO	ENTIDADES	RELACIONES	REGLA ÚTIL
Universidad	Alumno, asignatura, matrícula.	`cursa`, `aprueba`, `requiere`.	No matricular si falta prerrequisito.
Finanzas	Factura, cliente, contrato, rol.	`perteneceA`, `autoriza`, `superaUmbral`.	Escalar si importe supera límite.
Producto	Plan, función, usuario, permiso.	`incluye`, `puedeUsar`, `requiere`.	Mostrar solo funciones disponibles.
Operación	Servicio, base de datos, equipo.	`dependeDe`, `mantiene`, `expone`.	Avisar a equipos afectados.

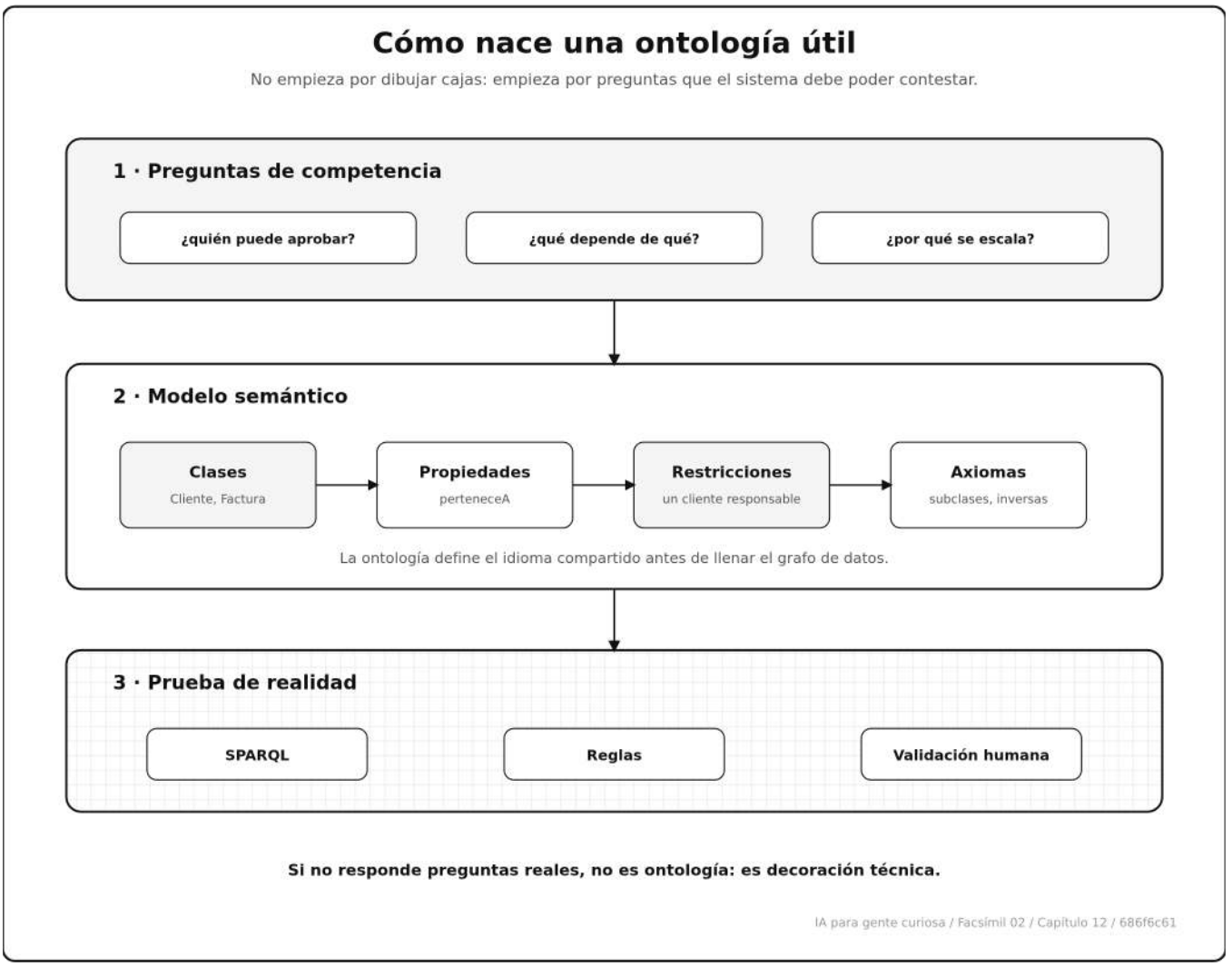
Ejemplo: en un producto SaaS, una ontología mínima podría decir:

CLASE	INSTANCIAS	PROPIEDADES
`Cliente`	`cliente:c42`	`tieneContrato`, `tienePlan`.
`Plan`	`plan:empresa`	`incluyeFuncion`.
`Funcion`	`funcion:exportarDatos`	`requierePermiso`.
`Usuario`	`usuario:u17`	`perteneceACliente`, `tieneRol`.
`Permiso`	`permiso:exportar`	`autorizaFuncion`.

Con eso puedes contestar algo muy concreto: “¿puede este usuario exportar datos?”. El LLM puede explicar la respuesta en lenguaje humano, pero la decisión debería apoyarse en relaciones verificables:

$$\text{puedeUsar}(u, f) \Leftarrow \text{perteneceACliente}(u, c) \land \text{tienePlan}(c, p) \land \text{incluyeFuncion}(p, f) \land \text{tienePermiso}(u, \pi) \land \text{a}$$

PARTE DE LA REGLA	LECTURA HUMANA
`perteneceACliente`	El usuario pertenece a ese cliente.
`tienePlan`	El cliente tiene un plan contratado.
`incluyeFuncion`	El plan incluye la función pedida.
`tienePermiso`	El usuario tiene un permiso concreto.
`autorizaFuncion`	Ese permiso autoriza esa función.



La ontología también necesita gobierno. Alguien debe poder responder: quién puede añadir una clase, cuándo una relación queda obsoleta, cómo se migran instancias, qué cambios rompen consultas y cómo se documenta una decisión de modelado. Sin eso, el grafo envejece rápido.

DECISIÓN DE GOBIERNO	PREGUNTA PRÁCTICA
Dueño del vocabulario	¿Quién aprueba cambios de clases y propiedades?
Versionado	¿Qué consultas o agentes dependen de esta versión?
Calidad de datos	¿Qué instancias están incompletas o duplicadas?
Deprecación	¿Qué relación antigua sigue existiendo solo por compatibilidad?
Trazabilidad	¿Qué fuente justifica este hecho o regla?

Un sistema experto combina cinco piezas:

PIEZA	QUÉ CONTIENE	PREGUNTA QUE RESPONDE
Base de conocimiento	Hechos del dominio.	¿Qué sabemos?
Ontología	Vocabulario y estructura.	¿Qué significa lo que sabemos?
Reglas	Condiciones explícitas.	¿Qué se deriva?
Motor de inferencia	Mecanismo que aplica reglas.	¿Qué conclusiones siguen?
Explicación	Trazas de hechos y reglas.	¿Por qué?

Esto suena antiguo hasta que lo conectas con agentes modernos. Un LLM puede interpretar una solicitud. Un RAG puede traer contexto. Pero una ontología y una regla simbólica pueden decidir si una tool está permitida, si falta una aprobación o si una respuesta debe incluir una fuente.

No es nostalgia. Es ingeniería.

Grafo de conocimiento, vector store y GraphRAG

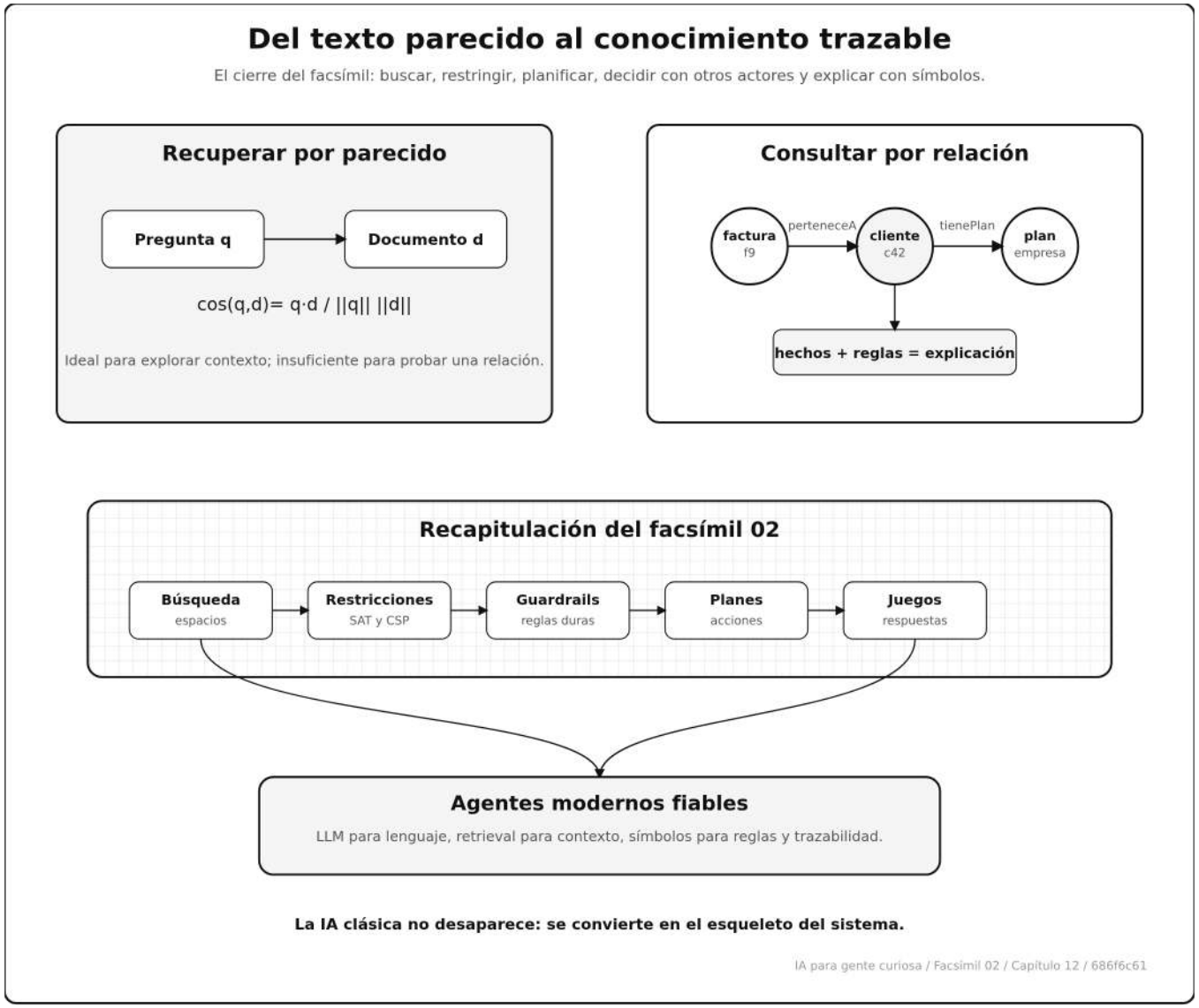
Un vector store recupera por parecido. Un grafo de conocimiento consulta por relación. GraphRAG intenta usar estructura de grafo para mejorar recuperación, agregación y explicación en sistemas RAG.[11] La idea es atractiva, pero conviene no confundir extracción automática con conocimiento fiable.

NECESIDAD	VECTOR STORE	GRAFO DE CONOCIMIENTO	HÍBRIDO
Encontrar texto relevante.	Muy fuerte.	Menos directo.	Recupera documentos y entidades.
Responder “quién depende de quién”.	Débil si no hay frases explícitas.	Muy fuerte.	Consulta grafo y cita documentos.
Explicar una decisión.	Muestra fragmentos.	Muestra hechos y reglas.	Une evidencia textual y trazabilidad.
Actualizar permisos.	Reindexar texto no basta.	Cambiar regla o relación.	Validar acciones con reglas.

La combinación madura suele verse así:

1. El LLM entiende la pregunta y propone una intención.
2. El vector store recupera fragmentos útiles.
3. El grafo resuelve entidades y relaciones.
4. Las reglas validan permisos, límites y coherencia.
5. La respuesta cita evidencia y explica el camino.

Cuando alguien dice “hagamos GraphRAG”, la pregunta buena es: ¿qué relaciones sabemos de verdad, quién las mantiene y cómo sabremos que siguen siendo correctas?



En el día a día

El conocimiento simbólico aparece cuando necesitas que el sistema recuerde hechos con nombre propio.

SITUACIÓN	SIN SÍMBOLOS	CON SÍMBOLOS
Soporte interno	Buscar tickets parecidos.	Saber qué cliente, contrato y SLA aplican.
Agente con tools	El modelo decide desde texto.	Las tools consultan permisos y estado.
Compliance	Resumen libre de políticas.	Reglas ejecutables y trazas revisables.
Operaciones	Documentos de arquitectura.	Grafo de dependencias vivo.
Producto	Preguntas frecuentes.	Planes, funciones y permisos consultables.

La señal práctica es sencilla: si la frase contiene “siempre”, “solo si”, “depende de”, “pertenece a”, “autoriza”, “requiere” o “explica por qué”, probablemente hay conocimiento simbólico esperando salir.

Por qué debería importarte

Porque muchos sistemas de IA fallan no por falta de modelo, sino por falta de estructura alrededor del modelo.

Si todo vive como texto, cada decisión vuelve a interpretarse desde cero. Si una regla está en un prompt, no es una regla operativa. Si una entidad no tiene identidad estable, dos sistemas pueden hablar de lo mismo sin saberlo. Si una relación no está modelada, el sistema solo podrá adivinarla por parecido.

El conocimiento simbólico no hace que un sistema sea perfecto. Hace algo igual de importante: permite preguntar, validar y explicar.

Recapitulación activa del facsímil

Este cierre funciona como el capítulo 12 del facsímil 1: no es un resumen para leer rápido, sino una revisión activa. Cada sección recupera un concepto nuclear, lo reformula desde otro ángulo, lo conecta con el resto y te confronta con una pregunta. Si algo no te sale, el número de capítulo te dice exactamente dónde volver.

No es un examen. Es un espejo. Si te reconoces en estas páginas, tienes el vocabulario clásico que permite mirar los agentes modernos como sistemas de decisión, no como cajas negras.

---

1. Búsqueda: resolver problemas como espacio de estados

El concepto. Un problema de búsqueda se define por estado inicial, acciones, transición, objetivo y coste. Resolverlo es recorrer un espacio hasta encontrar una ruta aceptable.

Para recordar. Si no sabes nombrar estados y acciones, todavía no tienes un problema de IA: tienes una idea sin contrato operativo.

Ejemplo fresco. Un agente que reserva una cita médica tiene estados: especialidad elegida, fecha filtrada, seguro validado, cita confirmada. Cada click cambia el estado.

Vuelve al capítulo 1 si: no puedes escribir $P = (S, A, T, s_0, G, c)$ y explicar cada pieza.

---

2. BFS, DFS y coste uniforme

El concepto. Los algoritmos ciegos comparten el mismo bucle: extraer de una frontera, expandir vecinos y decidir qué entra después. Cambia la estructura de la frontera.

Para recordar. BFS prioriza poca profundidad, DFS poca memoria, UCS menor coste acumulado.

Ejemplo fresco. Si todas las acciones cuestan igual, BFS puede encontrar la solución más corta. Si llamar a una API cuesta más que leer caché, necesitas coste uniforme.

Vuelve al capítulo 2 si: confundes profundidad, anchura y coste acumulado.

---

3. Greedy, A* y heurísticas

El concepto. Una heurística estima cuánto falta. Greedy mira solo $h(n)$. A* combina lo recorrido y lo estimado:

$$f(n) = g(n) + h(n)$$

Para recordar. Una buena estimación no reemplaza el coste real. A* funciona tan bien porque equilibra ambos.

Ejemplo fresco. En un asistente que depura código, $g(n)$ puede ser coste ya gastado y $h(n)$ la cercanía estimada al fallo. Si solo sigues la pista más prometedora, puedes ignorar una solución barata.

Vuelve al capítulo 3 si: no puedes explicar por qué Greedy puede ser rápido y equivocarse.

---

4. Búsqueda en agentes modernos

El concepto. Un agente moderno también busca: propone pasos, ejecuta tools, observa resultados y replanifica.

Para recordar. El LLM no es todo el sistema. Es una pieza dentro de un bucle con estado, herramientas, validaciones y memoria.

Ejemplo fresco. Un agente de datos prueba una consulta SQL, recibe error de columna inexistente, revisa el esquema y genera otra consulta. Eso es búsqueda con observación.

Vuelve al capítulo 4 si: ves un agente como una respuesta larga en vez de como un proceso iterativo.

---

5. SAT y CSP

El concepto. SAT pregunta si existe una asignación booleana que hace verdadera una fórmula. CSP generaliza a variables, dominios y restricciones.

Para recordar. A veces la IA no “predice”: satisface condiciones.

Ejemplo fresco. Crear horarios de un curso no es escribir texto bonito. Es asignar aulas, docentes y franjas sin romper restricciones.

Vuelve al capítulo 5 si: no puedes distinguir validez, satisfacibilidad y optimización.

---

6. Variables, dominios y restricciones

El concepto. Un CSP se modela como:

$$\mathcal{P} = (X, D, C)$$

Para recordar. Modelar bien decide más que elegir solver. Variables malas producen problemas raros.

Ejemplo fresco. Si una variable representa “turno completo” quizá el dominio explota. Si separas día, hora y persona, aparecen restricciones más claras.

Vuelve al capítulo 6 si: te cuesta convertir un problema cotidiano en variables, dominios y restricciones.

---

7. Propagación, backtracking y heurísticas en CSP

El concepto. Propagar reduce dominios antes de buscar. Backtracking prueba asignaciones parciales y vuelve atrás cuando una restricción falla. Heurísticas como MRV eligen primero lo más limitado.

Para recordar. El gran ahorro no está en probar más rápido, sino en probar menos.

Ejemplo fresco. Si Ana solo puede lunes o martes, decidir su turno antes puede revelar pronto si el horario es viable.

Vuelve al capítulo 7 si: no puedes explicar por qué “fallar pronto” es una virtud.

---

8. Restricciones como guardrails

El concepto. Un guardrail convierte una regla de negocio o seguridad en validación ejecutable.

Para recordar. Un prompt orienta. Un control decide.

Ejemplo fresco. “No borres datos sin aprobación” no debería vivir solo como frase. Debe ser permiso, schema, umbral y traza.

Vuelve al capítulo 8 si: todavía pones reglas duras únicamente en texto libre.

---

9. Planificación automática

El concepto. Planificar es encontrar una secuencia de acciones que transforma un estado inicial en un estado objetivo respetando precondiciones y efectos.

Para recordar. Una lista de tareas no es un plan si no dice qué debe ser cierto antes y después de cada acción.

Ejemplo fresco. Para enviar una factura: validar cliente, calcular importe, generar PDF, aprobar, enviar y registrar. Si falta aprobación, el plan no es ejecutable.

Vuelve al capítulo 9 si: no puedes distinguir acción, precondición, efecto y objetivo.

---

10. Planificación heurística, SAT y agentes LLM

El concepto. La planificación avanzada usa heurísticas para ordenar búsqueda, codificación SAT para probar horizontes y bucles agente para observar y replanificar.

Para recordar. Proponer una acción no equivale a tener permiso para ejecutarla.

Ejemplo fresco. Un agente puede proponer actualizar una base de datos. El sistema debe comprobar estado, permisos, coste, reversibilidad y trazas antes de actuar.

Vuelve al capítulo 10 si: no puedes explicar qué significa horizonte k en planificación con SAT.

---

11. Juegos: decidir con otros actores

El concepto. Los juegos añaden interdependencia: otras personas, sistemas, reglas o instrucciones también pueden elegir.

Para recordar. La calidad de una acción depende de las respuestas que habilita.

Ejemplo fresco. Si un documento recuperado contiene otra orden, tu agente debe distinguir datos de instrucciones antes de llamar una tool.

Vuelve al capítulo 11 si: evalúas decisiones solo por el primer movimiento.

---

12. Conocimiento simbólico

El concepto. Entidades, relaciones, reglas y consultas explícitas permiten representar conocimiento trazable.

Para recordar. Los embeddings recuperan parecido. Los símbolos expresan relación.

Ejemplo fresco. “Cliente C42 puede usar la función X porque su plan la incluye y su contrato está activo” es una explicación simbólica.

Vuelve a este capítulo si: no puedes distinguir un vector store de un grafo de conocimiento.

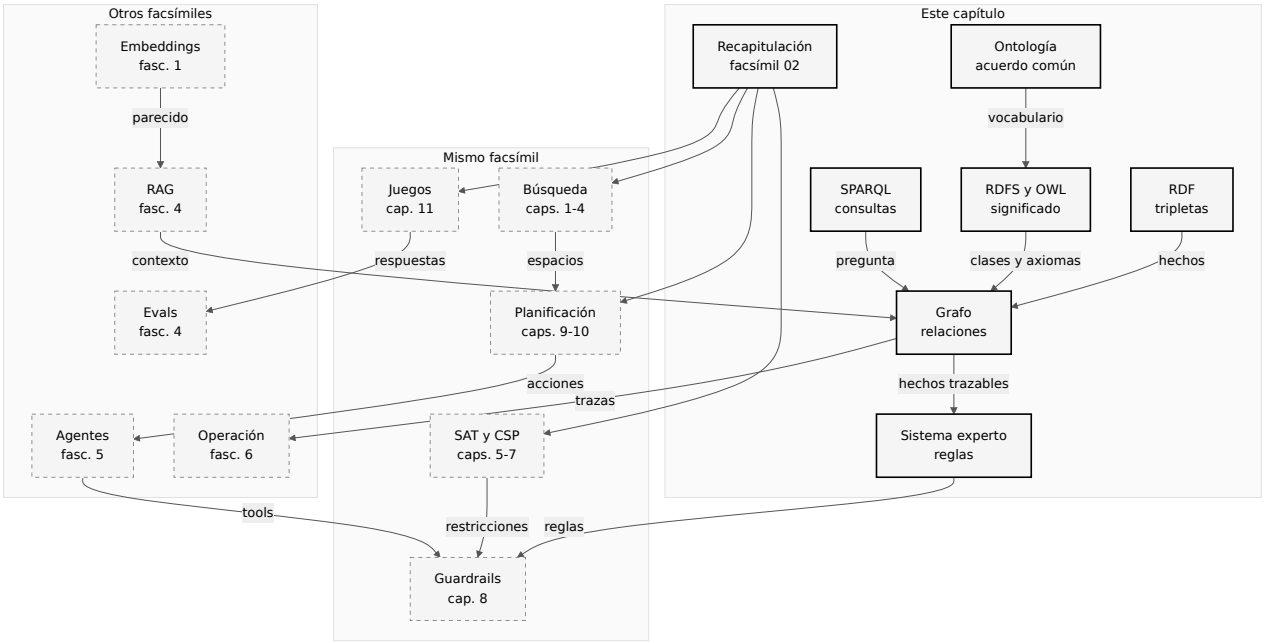
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Pensar que simbólico significa viejo	Muchas arquitecturas modernas necesitan reglas, permisos, grafos y consultas.	Preguntar qué parte del sistema debe ser trazable.
Confundir similitud con verdad	Un texto cercano puede no probar una relación.	Separar recuperación, verificación e inferencia.
Sobremodelar una ontología	Una ontología enorme se vuelve inmantenible.	Empezar con pocas clases y relaciones críticas.
Creer que GraphRAG aparece solo	Extraer entidades no garantiza conocimiento correcto.	Definir dueños, validación y actualización del grafo.
Meter reglas duras en prompts	El prompt no es una base de conocimiento ejecutable.	Convertir reglas en políticas, schemas o consultas.
Recapitular como quien pasa lista	Recordar nombres no asegura comprensión.	Volver a explicar cada pieza con un ejemplo nuevo.

Cómo encaja todo

Este mapa es el cierre del facsímil. La búsqueda, las restricciones, la planificación y los juegos nos han dado mecanismos para decidir. El conocimiento simbólico añade identidad, relaciones y explicación: qué entidad es cuál, qué regla aplica y qué consulta demuestra una decisión.

La decisión aprendida es cuándo no basta con parecido semántico. Si necesitas trazabilidad, permisos, dependencias o una explicación reproducible, los hechos y relaciones explícitas se vuelven parte de la arquitectura.


Otros facsímiles
Embeddings fasc. 1
parecido
RAG fasc. 4
contexto
Evals fasc. 4
Agentes fasc. 5
Operación fasc. 6
tools
Mismo facsímil
Juegos cap. 11
Búsqueda caps. 1-4
respuestas
espacios
Planificación caps. 9-10
acciones
trazas
SAT y CSP caps. 5-7
restricciones
reglas
Guardrails cap. 8
Este capítulo
Recapitulación facsímil 02
Ontología acuerdo común
vocabulario
SPARQL consultas
RDFS y OWL significado
RDF tripletas
pregunta
clases y axiomas
hechos
Grafo relaciones
hechos trazables
Sistema experto reglas

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Entidad	Objeto identificable del dominio.
Tripleta RDF	Hecho con forma sujeto-predicado-objeto.
Grafo de conocimiento	Conjunto de entidades y relaciones explícitas.
Ontología	Acuerdo formal sobre clases, relaciones y restricciones.
Clase	Categoría de entidades dentro de una ontología.
Instancia	Entidad concreta que pertenece a una clase.
Propiedad	Relación o atributo que conecta entidades o asigna valores.
Pregunta de competencia	Pregunta que la ontología debería poder responder.
RDFS	Vocabulario para clases, subclases, dominio y rango.
OWL	Lenguaje de ontologías con axiomas más expresivos.
SHACL	Lenguaje del W3C para validar grafos RDF contra formas (shapes) con reglas cerradas.
SPARQL	Lenguaje para consultar patrones de tripletas.
Consulta SPARQL	Pregunta formal que enlaza variables con hechos del grafo.
FILTER	Condición que restringe las soluciones de una consulta.
OPTIONAL	Bloque que añade datos si existen sin eliminar la solución principal.
Camino de propiedad	Expresión para recorrer relaciones encadenadas.
Linked Data	Datos publicados con identificadores estables y enlaces.
Sistema experto	Ontología, hechos, reglas, inferencia y explicación.
Vector store	Almacén de embeddings para recuperar por similitud.

Antes de pasar página

- [ ] ¿Puedo explicar la diferencia entre parecido semántico y relación explícita? (Si no, vuelve a «Cuando parecerse no basta».)
- [ ] ¿Sé leer una tripleta RDF como sujeto, predicado y objeto? (Si no, vuelve a «RDF: hechos pequeños con identidad».)
- [ ] ¿Entiendo qué añade una ontología frente a una lista de hechos? (Si no, vuelve a «Ontologías y sistemas expertos».)
- [ ] ¿Puedo distinguir clase, instancia, propiedad, restricción y axioma? (Si no, vuelve a «RDFS y OWL: decir qué significa el grafo».)
- [ ] ¿Sé escribir preguntas de competencia para acotar una ontología? (Si no, vuelve a «Ontologías y sistemas expertos».)
- [ ] ¿Puedo explicar para qué sirve SPARQL? (Si no, vuelve a «SPARQL: preguntar por relaciones exactas».)
- [ ] ¿Sé leer `SELECT`, `WHERE`, `FILTER`, `OPTIONAL`, `ASK` y `CONSTRUCT`? (Si no, vuelve a «SPARQL: preguntar por relaciones exactas».)
- [ ] ¿Entiendo por qué `:dependeDe+` encuentra dependencias encadenadas? (Si no, vuelve a «SPARQL: preguntar por relaciones exactas».)
- [ ] ¿Sabría escribir el motor mínimo que infiere tipos por subclase y resuelve un patrón? (Si no, vuelve a «Modo ingeniero: el motor en pocas líneas».)
- [ ] ¿Distingo cuándo inferir con OWL (mundo abierto) y cuándo validar con SHACL (mundo cerrado)? (Si no, vuelve a «SHACL: cerrar el mundo cuando hace falta».)
- [ ] ¿Sé cuándo usar vector store, grafo o una combinación? (Si no, vuelve a «Grafo de conocimiento, vector store y GraphRAG».)
- [ ] ¿Puedo resumir los doce capítulos del facsímil con un ejemplo de cada uno? (Si no, vuelve a «Recapitulación activa del facsímil».)
- [ ] ¿Tengo claro qué aporta la IA clásica a los agentes modernos? (Si no, vuelve a «Por qué debería importarte».)

En resumen

IDEA FUERZA	DETALLE
Los vectores recuperan parecido.	Muy útiles para contexto, menos para probar relaciones.
Los símbolos dan identidad.	Entidades y relaciones permiten consultar y explicar.
RDF modela hechos mínimos.	Una tripleta basta para declarar una relación.
Una ontología es contrato semántico.	Define clases, propiedades, restricciones, axiomas y alcance.
RDFS y OWL añaden significado.	Clases, subclases, dominio, rango y axiomas.
SPARQL pregunta al grafo.	Enlaza variables con patrones exactos, filtros, opcionales y caminos de propiedad.
GraphRAG necesita cuidado.	Un grafo útil requiere validación, mantenimiento y dueños.
La IA clásica sigue viva.	Búsqueda, restricciones, planificación, juegos y símbolos son infraestructura para sistemas modernos.

Recursos para seguir: leer, construir y experimentar

La IA clásica (búsqueda, restricciones, planificación, juegos) sigue muy viva debajo de los sistemas modernos. Aquí tienes recursos reales para seguir tocándola.

Para experimentar sin código. Es de lo más visual del facsímil, y lo has probado en las cajas «Pruébalo en 5 minutos»: el visualizador de PathFinding.js (`qiao.github.io/PathFinding.js/visual`) para comparar BFS, Dijkstra y A* sobre un laberinto; el editor de PDDL online (`editor.planning.domains`) para describir un dominio y dejar que un planificador encuentre el plan; y Lichess (`lichess.org`) para jugar contra un motor por niveles y sentir cómo la profundidad de búsqueda lo hace más fuerte.

Para construir. Cuando quieras programarlo: para búsqueda y grafos, `networkx` en Python; para restricciones y optimización, OR-Tools de Google, una de las mejores librerías de CSP y programación con restricciones; para planificación, planificadores como Fast Downward conectados a tus dominios PDDL. Casi todo es código abierto y se ejecuta en tu máquina.

Para leer. La referencia canónica es *Artificial Intelligence: A Modern Approach*, de Russell y Norvig: el libro que define este campo y que cubre con detalle todo lo de este facsículo. Cada capítulo cierra además con su «Para saber más». Los cuadernos del facsímil son el puente entre el algoritmo dibujado y el código que lo implementa.

Para saber más

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D. y Patel-Schneider, P. F. (eds.) (2003). *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press.

Berners-Lee, T. (2006). *Linked Data*. https://www.w3.org/DesignIssues/LinkedData.html

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2), 199-220. https://doi.org/10.1006/knac.1993.1008

Microsoft. (2024). *GraphRAG*. https://microsoft.github.io/graphrag/

Noy, N. F. y McGuinness, D. L. (2001). *Ontology Development 101: A Guide to Creating Your First Ontology*. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05. https://protege.stanford.edu/publications/ontology_development/ontology101.pdf

Pérez, J., Arenas, M. y Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3), 1-45. https://doi.org/10.1145/1567274.1567278

Russell, S. y Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4.ª ed.). Pearson.

Salton, G., Wong, A. y Yang, C. S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18(11), 613-620. https://doi.org/10.1145/361219.361220

W3C. (2014). *RDF 1.1 Concepts and Abstract Syntax*. https://www.w3.org/TR/rdf11-concepts/

W3C. (2014). *RDF Schema 1.1*. https://www.w3.org/TR/rdf-schema/

W3C. (2012). *OWL 2 Web Ontology Language Document Overview*. https://www.w3.org/TR/owl2-overview/

W3C. (2013). *SPARQL 1.1 Query Language*. https://www.w3.org/TR/sparql11-query/

W3C. (2017). *Shapes Constraint Language (SHACL)*. https://www.w3.org/TR/shacl/

Cuadernos para practicar

Has modelado problemas de búsqueda, restricciones y planificación a lo largo del facsímil; estos cuadernos te dejan ejecutar esas ideas. Son *notebooks* que se abren en Google Colab —gratis, en el navegador— o te puedes descargar. Cada uno está explicado paso a paso, con salidas reales, y dice de qué capítulo sale.

Buscar en un mapa: BFS, DFS, coste uniforme y A*

Qué practicas: ver cómo cuatro algoritmos de búsqueda recorren un mapa y, sobre todo, cuánto miran para encontrar la ruta. **Dónde encaja:** capítulos 1 a 3 (espacio de estados, algoritmos ciegos y A* con heurísticas). **Qué necesitas:** un navegador. Corre en CPU; sin nada más.

Sueltas un repartidor en una explanada con un edificio que hay que rodear y lanzas los cuatro buscadores hacia el destino. Los tres óptimos (BFS, coste uniforme y A*) encuentran el mismo camino de 45 pasos, pero mira cuánto exploran: BFS mira **532** casillas —casi todo el mapa, en todas direcciones— y A*, gracias a su heurística (la distancia a la meta) y a un buen desempate, mira solo **45**: justo las del camino. DFS, en cambio, llega rápido a *algo*... pero su ruta es de 151 pasos. Lo ves pintado: la nube gris de BFS llena el mapa; la de A* es una línea dirigida a la meta.

Esto es lo que late bajo un GPS o un agente que decide pasos: no basta con encontrar el camino, importa cuánto cuesta encontrarlo.

▶ Abrir en Google Colab **↓ Descargar el cuaderno (.ipynb)**

Sudoku como CSP: fuerza bruta frente a propagación

Qué practicas: modelar un sudoku como problema de restricciones y ver cuánto ahorra propagar antes de probar. **Dónde encaja:** capítulos 5 a 7 (CSP; variables, dominios y restricciones; propagación y heurísticas). **Qué necesitas:** un navegador. Corre en CPU; sin nada más.

Tratas el sudoku como lo que es: 81 casillas con valores posibles y reglas que las atan. Lo resuelves de dos maneras y cuentas el esfuerzo. A lo bruto —probar 1, 2, 3... y retroceder al atascarte— necesita **4208** colocaciones. Con cabeza —tachar de los vecinos cada valor que colocas (*forward checking*) y atacar siempre la casilla con menos opciones (heurística MRV)— bastan **51**: un 99% menos de trabajo para la misma solución. No es que el ordenador corra más: es que deduce en vez de adivinar.

Esa diferencia —razonar sobre las restricciones antes de actuar— es exactamente lo que separa un sistema que tantea de uno que decide con criterio.

► **Abrir en Google Colab** ↓ **Descargar el cuaderno (.ipynb)**

Decidir contra alguien: minimax y poda alfa-beta

Qué practicas: construir un jugador que no pierde nunca y ver cuánto ahorra la poda alfa-beta. **Dónde encaja:** capítulo 11 (juegos: decidir con otros actores). **Qué necesitas:** un navegador. Corre en CPU; sin nada más.

Construyes un jugador de tres en raya que piensa todas las jugadas —suyas y del rival — hasta el final, asumiendo que el otro juega lo mejor posible: eso es minimax. Calcula que el valor del juego es **0** (con juego perfecto, siempre empate: no hay victoria que forzar) tras visitar **549.946** estados. Luego le añades la **poda alfa-beta**, que descarta ramas que ya no pueden cambiar la decisión, y baja a **18.297** —un 97% menos— sin cambiar ni una jugada. Para rematar, lo pones a jugar 2000 partidas contra un rival al azar: gana 1993, empata 7 y **pierde 0**.

Es cómo razona una IA cuando hay otro que responde —un rival, un mercado, otro agente—: asumir que el otro juega óptimo y decidir en consecuencia.

► **Abrir en Google Colab** ↓ **Descargar el cuaderno (.ipynb)**

Planificar de verdad: el mundo de bloques

Qué practicas: describir acciones con precondiciones y efectos y dejar que un planificador deduzca el plan. **Dónde encaja:** capítulos 9 y 10 (planificación automática, PDDL y modelado de dominios). **Qué necesitas:** un navegador. Corre en CPU; sin nada más.

Le das a un planificador unos bloques desordenados y una foto de cómo los quieres —sin decirle los pasos—. Solo describes cuatro acciones (coger, dejar, apilar, desapilar) con lo que cada una exige y lo que provoca, y él deduce la secuencia: aquí, **6 pasos** tras explorar **18 estados**, empezando por quitar el bloque que estorba antes de poder construir la torre. En ningún momento le dijiste «primero libera A»: lo razonó a partir del modelo. Es la misma búsqueda del primer cuaderno, ahora sobre estados del mundo.

Es lo que late bajo un agente que «planifica pasos»: un modelo de acciones y un buscador que arma la secuencia.

► **Abrir en Google Colab** ↓ **Descargar el cuaderno (.ipynb)**

Notas

1. Salton, G., Wong, A. y Yang, C. S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18(11), 613-620. https://doi.org/10.1145/361219.361220 ↩

2. W3C. (2014). *RDF 1.1 Concepts and Abstract Syntax*. https://www.w3.org/TR/rdf11-concepts/ ↩

3. W3C. (2014). *RDF Schema 1.1*. https://www.w3.org/TR/rdf-schema/ ↩

4. W3C. (2012). *OWL 2 Web Ontology Language Document Overview*. https://www.w3.org/TR/owl2-overview/ ↩

5. Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D. y Patel-Schneider, P. F. (eds.) (2003). *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press. ↩

6. W3C. (2017). *Shapes Constraint Language (SHACL)*. https://www.w3.org/TR/shacl/ ↩

7. W3C. (2013). *SPARQL 1.1 Query Language*. https://www.w3.org/TR/sparql11-query/ ↩

8. Pérez, J., Arenas, M. y Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3), 1-45. https://doi.org/10.1145/1567274.1567278 ↩

9. Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2), 199-220. https://doi.org/10.1006/knac.1993.1008 ↩

10. Noy, N. F. y McGuinness, D. L. (2001). *Ontology Development 101: A Guide to Creating Your First Ontology*. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05. https://protege.stanford.edu/publications/ontology_development/ontology101.pdf ↩

11. Microsoft. (2024). *GraphRAG*. https://microsoft.github.io/graphrag/ ↩