1. La lógica en la informática

Durante muchos años se ha estado confundiendo la esencia intelectual de la Informática con las herramientas que se han utilizado. Por ejemplo, se dice: "no puede ser que el alumno de Informática no sepa Java, ni el lenguaje máquina del Pentium!". Pero hace 25 años no teníamos ni Windows, ni Pentiums, ni Java, y dentro de 25 años tampoco los tendremos (una muy acertada observación de Jordi Cortadella; véanse los libros del 25 aniversario de la FIB). Por lo tanto, debemos preguntarnos: ¿qué *fundamentos* deberían aprender los estudiantes para que en un futuro dispongan de madurez y agilidad al tener que asimilar los nuevos conceptos, lenguajes, técnicas y herramientas que surjan?

Gran parte de la respuesta está en el estudio de la Lógica. Ya lo dijo McCarthy[1] en los años 60, cuando aún era difícil apreciar su justeza, al considerar que "la lógica va a tener para la informática una importancia comparable a la que tuvo el análisis matemático para la física". Antes incluso que eso, el propio Alan Turing[2] ya había afirmado:

> "I expect that digital computing machines will eventually stimulate a considerable interest in symbolic logic (...). The language in which one communicates with these machines (...) forms a sort of symbolic logic."

Asimismo, informáticos actuales de la talla de Moshe Vardi han llamado a la lógica "el cálculo de la informática", debido a que su papel en la informática tanto teórica como práctica es similar al de las matemáticas en las ciencias físicas y, en particular, al del cálculo en ingeniería.

Al igual que los arquitectos e ingenieros, que analizan sus construcciones matemáticamente, los informáticos necesitan analizar las propiedades lógicas de sus sistemas mientras los diseñan, desarrollan, verifican y mantienen, especialmente cuando se trata de sistemas críticos económicamente (un fallo tendría un alto coste económico), o críticos en seguridad o privacidad. Pero también en sistemas cuya *eficiencia* resulta crítica, los análisis lógicos pueden ser reveladores y, en *todos* los tipos de sistemas, los métodos y herramientas basados en la lógica pueden mejorar la calidad y reducir costes.

Esta visión está ampliamente documentada en el artículo de Vardi y otros: "On the Unusual Effectiveness of Logic in Computer Science" (disponible en `www.lsi.upc.edu/~roberto`; ver allí también "The calculus of Computer Science"). En él se detalla el papel crucial de la lógica en la informática en general y, en particular, en áreas como la complejidad computacional, bases de datos, lenguajes de programación, inteligencia artificial, o la verificación de sistemas.

[1] Importante investigador y pionero de la Inteligencia Artificial, Premio Turing en 1971.

[2] 1912–1954, uno de los padres de la Informática. Formalizó los conceptos de algoritmo y computación mediante la Máquina de Turing. La hoy ampliamente aceptada Tesis de Church-Turing postula que ningún modelo computacional existente supera (en un sentido definido de manera precisa) las capacidades computacionales de la Máquina de Turing.

1.1. Ejemplos de usos de la lógica en la informática

Los métodos y herramientas basados en la lógica se utilizan cada vez más en todo tipo de sistemas de hardware y software. A menudo ha ocurrido que una técnica pensada para un uso muy concreto, *ad-hoc*, ha resultado ser un método lógico mucho más general, con muchas más aplicaciones. También ha pasado que técnicas desarrolladas independientemente, para aplicaciones muy distintas, han resultado ser "casualmente" el mismo principio lógico! Por ejemplo:

- *La lógica proposicional*, que veremos en IL, se usa para la especificación, síntesis y verificación de numerosas clases de circuitos y sistemas (reactivos, asíncronos, concurrentes, distribuidos). La deducción en lógica proposicional (resolver el problema de SAT, como veremos en IL) también sirve para resolver muy diversos problemas de la vida real: horarios, rutas de transporte, planificación de obras,...

- *La resolución para cláusulas de Horn* (una manera de hacer deducción en lógica de primer orden, que veremos en IL) es el principio lógico común resultante de, por un lado, la evolución de sistemas de bases de datos (que progresaron desde los jerárquicos y en red hacia los relacionales y deductivos), y, por otro lado, de los lenguajes de programación lógica, como Prolog, y sus extensiones a la programación lógica con restricciones y la programación lógico-funcional.

- *Las técnicas de reescritura*, la aplicación de *reglas* lógicas, son la base común de la programación funcional (LISP, ML, Miranda,...), de diversas aplicaciones en inteligencia artificial (sistemas expertos, bases de conocimiento), de ciertos (sub)sistemas de álgebra por ordenador (Maple, Mathematica), etc.

- La lógica también está jugando un papel central desde el primer momento en la llamada *web semántica*, que permite buscar con criterios semánticos; un ejemplo muy sencillo es poder buscar un herrero *de profesión*, sin tener que filtrar miles de respuestas sobre personas cuyo apellido es Herrero. En la web semántica se usan las *description logics* para dotar a las páginas de internet y de intranets de criterios de búsqueda semánticos, mecanismos deductivos, restricciones de consistencia o integridad, etc.

- Otro buen ejemplo es la necesidad absoluta de lógicas para la verificación de protocolos criptográficos, usados en aplicaciones que requieren privacidad, autentificación (firma electrónica), dinero electrónico anónimo, etc.

- Existen estrechas relaciones entre la lógica y la *complejidad computacional*, que estudia los recursos en cuanto a tiempo de cálculo o memoria necesarios para resolver clases de problemas. A veces es posible determinar la complejidad de un problema simplemente expresando el problema en una determinada lógica.

1.2. Lógica matemática vs. lógica informática

Para los matemáticos, el interés en la lógica surgió en parte del programa de Hilbert de dotar a las matemáticas de fundamentos sólidos[3]. Pero para la investigación actual en matemáticas la lógica ya no juega ese papel central, y hace énfasis en aspectos diferentes a los que interesan en la informática, donde, en cambio, la lógica adquiere un interés cada vez mayor. Hoy día, son de informática la mayoría de los trabajos de investigación en lógica que se publican.

No es lo mismo estudiar la lógica desde el punto de vista de un matemático que desde el de un informático. Por ejemplo, para un informático no sólo importa el poder expresivo de un lenguaje lógico, sino también la comodidad o la naturalidad al utilizarlo, porque su tarea consiste precisamente en crear formalizaciones (programas, especificaciones,...) y necesita hacerlo con la máxima facilidad; un informático sabe perfectamente que es preferible programar en un lenguaje de alto nivel que en un lenguaje máquina, aunque ambas tengan el mismo "poder expresivo".

En los aspectos de implementabilidad y eficiencia son también evidentes las diferencias: los informáticos desean ver sus lógicas funcionando, en forma de, por ejemplo, lenguajes de programación, bases de datos, herramientas de síntesis y verificación, o lenguajes de consulta de la *web semántica*. Los matemáticos tradicionales no se han interesado tanto por aspectos como los procedimientos de decisión viables en la práctica, o las clases de complejidad asociadas a las distintas lógicas, o lo que hoy día se denomina "Lógicas Computacionales", o la combinación modular de lógicas mediante *constraints*.

2. Objetivos para la asignatura de IL

El objetivo de esta asignatura es proporcionar al estudiante una base sólida de conocimientos teóricos y prácticos de lógica para informáticos, y, al mismo tiempo, contribuir a su formación general en razonamiento formal (sobre conjuntos, relaciones, funciones, estructuras de orden, recursión, conteo) y en técnicas de demostración (contrarrecíproco, contraejemplos, inducción, reducción al absurdo). En concreto, la asignatura ha de proporcionar los siguientes conocimientos específicos:

1. El concepto de qué es una lógica, solamente en términos de sintaxis: *¿qué es una fórmula F?*, y semántica: *¿qué es una interpretación I?*, y *¿cuándo una interpretación I satisface a una fórmula F?*.

2. La definición de dos lógicas: la proposicional y la de primer orden.

3. Las propiedades de tautología/validez, contradicción, consecuencia y equivalencia lógica (de forma independiente de la lógica concreta), cómo se utilizan para formalizar problemas prácticos en informática, y cómo se reducen al problema de satisfactibilidad.

[3] Programa parcialmente frustrado con los resultados negativos de Gödel, Church y Turing.

4. Los métodos de deducción para determinar estas propiedades de mayor relevancia para la informática: Davis-Putnam, y resolución; su corrección y completitud con respecto de la definición de la lógica.

5. El poder expresivo de las dos lógicas, y entender en qué sentido la lógica proposicional es una restricción de la otra; compromiso entre poder expresivo y buenas propiedades computacionales: primeras nociones intuitivas de complejidad y decidibilidad.

6. Algunas de las -cada vez más abundantes- aplicaciones a la informática de los métodos deductivos: fundamentos de la programación lógica (Prolog) bases de datos deductivas, circuitos, problemas de planificación, etc.

También ha de proporcionar las siguientes habilidades:

1. Entender, escribir y manipular ágilmente fórmulas en ambas lógicas, con especial énfasis en aplicaciones a la informática.

2. Saber demostrar formalmente propiedades sencillas sobre fórmulas, conjuntos, relaciones, funciones, etc, mediante técnicas de demostración como el contrarrecíproco, contraejemplos, inducción, o reducción al absurdo.

3. Ser capaz de aplicar a mano resolución y Davis-Putnam sobre ejemplos prácticos abordables y saber utilizar la resolución como mecanismo de cómputo (cálculo de respuestas).

4. Saber expresar algunos problemas prácticos NP-completos (sudokus, horarios, problemas de grafos, circuitos) como problemas de satisfacción de lógica proposicional y resolverlos con un SAT solver.

5. Saber expresar problemas sencillos utilizando el lenguaje Prolog y entender las extensiones “extra-lógicas” de Prolog (la aritmética predefinida, el operador de corte y la negación, la entrada/salida).

1. ¿Qué es una Lógica?

Consideraremos en esta asignatura que *una lógica* es la unión de:

1. **Sintaxis:** ¿De qué símbolos dispone el lenguaje y de qué maneras los puedo combinar para obtener una fórmula F?
2. **Semántica:**
   - Una definición de *interpretación* I:
     ¿Qué posibles significados existen para los símbolos?
   - Una definición del concepto de *satisfacción*:
     ¿Cuándo una interpretación I *satisface* una fórmula F?

En una lógica también suele haber (pero no necesariamente) métodos de *deducción*, que describen cómo a partir de unas fórmulas dadas se pueden inferir otras nuevas, dando lugar a gran diversidad de aplicaciones prácticas de las lógicas. Aquí, para comenzar, veremos un ejemplo de lógica muy simple, la *lógica proposicional*, que, a pesar de su sencillez, permite expresar conceptos muy importantes en el mundo de la informática y tiene muchas aplicaciones.

2. Definición de la Lógica Proposicional

Sintaxis: En esta lógica, el *vocabulario* es un conjunto de *símbolos proposicionales* o *de predicado* $\mathcal{P}$ (cuyos elementos escribiremos normalmente como $p, q, r, \ldots$). Una *fórmula* de lógica proposicional sobre $\mathcal{P}$ se define como:

- Todo símbolo de predicado de $\mathcal{P}$ es una fórmula.
- Si F y G son fórmulas, entonces $(F \wedge G)$ y $(F \vee G)$ son fórmulas.
- Si F es una fórmula, entonces $\neg F$ es una fórmula.
- Nada más es una fórmula.

Interpretación: Una *interpretación* I sobre el vocabulario $\mathcal{P}$ es una función $I : \mathcal{P} \to \{0, 1\}$, es decir, I es una función que, para cada símbolo de predicado, nos dice si es 1 (cierto, true) o 0 (falso, false).

Satisfacción: Sean I una interpretación y F una fórmula, ambas sobre el vocabulario $\mathcal{P}$. La *evaluación* en I de F, denotada $eval_I(F)$, es una función que por cada fórmula da un valor de $\{0, 1\}$. Definimos $eval_I(F)$ para todos los casos posibles de F, usando *min*, *max* y $-$ que denotan el mínimo, el máximo, y la resta (sobre números del conjunto $\{0, 1\}$), donde, como sabemos:

$$\begin{array}{llllllll} min(0,0) & = & min(0,1) & = & min(1,0) & = & 0 \quad \text{y} & min(1,1) = 1 \\ max(1,1) & = & max(0,1) & = & max(1,0) & = & 1 \quad \text{y} & max(0,0) = 0: \end{array}$$

- si F es un símbolo p de $\mathcal{P}$ entonces $eval_I(F) = I(p)$
- $$\begin{array}{lll} eval_I(\quad (F \wedge G) \quad) & = & min(\ eval_I(F),\ eval_I(G)\ ) \\ eval_I(\quad (F \vee G) \quad) & = & max(\ eval_I(F),\ eval_I(G)\ ) \\ eval_I(\quad \neg F \quad) & = & 1 - eval_I(F) \end{array}$$

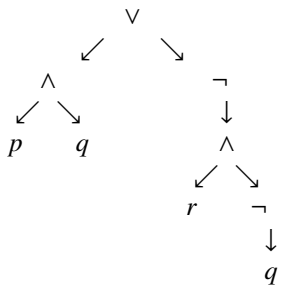
Definimos: si $eval_I(F) = 1$ entonces I *satisface* F, denotado $I \models F$. En este caso también se dice que F *es cierto en* I, o que I *es un modelo de* F.

3. Explicaciones sobre la definición de la lógica proposicional

Otras conectivas: En lógica proposicional, a $\wedge$, $\vee$ y $\neg$ se les llama *conectivas lógicas*. A menudo también se consideran otras conectivas, como $\rightarrow$ o $\leftrightarrow$. Aquí no las hemos introducido porque son expresables mediante las tres conectivas dadas. Por ejemplo, las fórmulas $F \rightarrow G$ y $F \leftrightarrow G$ pueden verse como abreviaturas de $(\neg F \vee G)$ y de $((\neg F \vee G) \wedge (\neg G \vee F))$ respectivamente.

No ambigüedad, representación en árbol: Las fórmulas, vistas como una cadena de símbolos (conectivas, símbolos de predicado y paréntesis), no son *ambiguas*: existe una única forma de entenderlas. Formalmente: para cada fórmula F existe un único árbol que representa la construcción de F según la definición de la sintaxis de la lógica proposicional.

Por ejemplo, la fórmula $((p \wedge q) \vee \neg(r \wedge \neg q))$ sólo puede leerse como un $\vee$ de la fórmula $(p \wedge q)$ y de la fórmula $\neg(r \wedge \neg q)$. Éstas a su vez sólo son legibles de una única manera, y así sucesivamente. El árbol correspondiente es:



Prioridades de conectivas y paréntesis: A veces podemos omitir paréntesis innecesarios: quitarlos no introduce ambigüedad. Por ejemplo, podemos escribir $p \wedge q$ en vez de $(p \wedge q)$. Además se asumen las siguientes *prioridades* entre las conectivas; de más prioritario a menos, tenemos: $\neg \ \wedge \ \vee \ \rightarrow \ \leftrightarrow$. Así, podemos escribir $p \wedge q \vee r$ para referirnos a $(p \wedge q) \vee r$, pero no podemos omitir paréntesis en $\neg(p \wedge (q \vee r))$. Nótese la similitud con la suma, el producto, y el "$-$" unario en aritmética: con $-x * y + z$ nos referimos a $((-x) * y) + z$ y no a, por ejemplo, $-(x * (y + z))$, ni a $(-x) * (y + z)$.

¿Qué cosas se modelan con esta lógica?: La idea intuitiva de esta lógica es que sirve para modelar (y razonar formalmente sobre) situaciones de la vida real en las que tratemos con enunciados o *proposiciones* que sólo pueden ser o bien ciertas o bien falsas.

Por ejemplo, si $\mathcal{P}$ es { *llueve*, *hace_sol*, *esta_nublado* }, cada interpretación I para esta $\mathcal{P}$ modela una situación real del tiempo del estilo de "sí llueve, no hace sol y no está nublado", es decir, $I(llueve) = 1$, $I(hace\_sol) = 0$, $I(esta\_nublado) = 0$. Con esta interpretación I, si F es la fórmula

$$llueve \wedge \neg(hace\_sol \vee esta\_nublado)$$

entonces, por la definición de $eval_I$, tenemos $eval_I(F) =$

$$\begin{array}{lllllll} min(& I(llueve), & 1 - & max(& I(hace\_sol), & I(esta\_nublado) &)) = \\ min(& 1, & 1 - & max(& 0, & 0 &)) = 1, \end{array}$$

luego F es cierta en I, es decir, tenemos $I \models F$. Similarmente, si G es la fórmula $\neg llueve \vee (hace\_sol \vee esta\_nublado)$, entonces I no satisface G, es decir, $I \not\models G$.

Podemos listar todas las posibles interpretaciones y la evaluación en cada una de ellas de la fórmula $llueve \wedge \neg(hace\_sol \vee esta\_nublado)$ mediante una *tabla de verdad* (escribiendo p, q, r en vez de *llueve*, *hace_sol*, y *esta_nublado*):

p	q	r	$p \wedge \neg(q \vee r)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

y vemos que en este ejemplo sólo hay una interpretación que satisface la fórmula, que es cuando llueve, no hace sol y no está nublado.

4. Satisfactibilidad, tautología, consecuencia, y equivalencia

Ahora podemos introducir la siguiente nomenclatura básica. Estas nociones no dependen de la lógica concreta. Se definen de manera idéntica en otras lógicas distintas a la lógica proposicional, como la lógica de primer orden. Por eso no se repetirán cuando tratemos a la lógica de primer orden.

1. Una fórmula F es *satisfactible* si tiene algún modelo, es decir, si existe alguna interpretación I tal que $I \models F$. Un fórmula F es *insatisfactible* (o es una *contradicción*) si no es satisfactible.

2. Una fórmula F es una *tautología* (o es *válida*), si *toda* interpretación es modelo de F, es decir, si para toda interpretación I tenemos $I \models F$.

3. Sean F y G fórmulas construidas sobre un vocabulario $\mathcal{P}$.
   Definimos: *F es consecuencia lógica de G* si todo modelo de G también lo es de F, es decir, si para toda interpretación I sobre $\mathcal{P}$ tal que $I \models G$ tenemos $I \models F$.
   Sobrecargando el operador "$\models$", esto se denotará por $G \models F$.

4. Sean F y G fórmulas construidas sobre un vocabulario $\mathcal{P}$.
   Definimos: *F y G son lógicamente equivalentes* si tienen los mismos modelos, es decir, si para toda interpretación I sobre $\mathcal{P}$ tenemos $I \models G$ si y sólo si $I \models F$.
   Esto se denotará por $G \equiv F$.

Nótese que el símbolo "$\models$" denota satisfacción cuando a su izquierda hay una interpretación I; en ese caso, $I \models F$ denota que I satisface F. En cambio, cuando a la izquierda hay una fórmula G, el símbolo "$\models$" denota consecuencia lógica; así, $G \models F$ denota que *todo modelo de G* satisface F.

Este tipo de propiedades (satisfactibilidad o tautología de una fórmula, consecuencia lógica o equivalencia entre fórmulas, etc.) se plantean en diversas aplicaciones prácticas de una lógica. Por ejemplo, en Intel pueden estar interesados en saber si dos circuitos (dos fórmulas) son equivalentes, o saber si un circuito F cumple cierta propiedad G (es decir, si $F \models G$).

La lógica proposicional es especialmente interesante porque todas estas propiedades son *decidibles*: para cada una de ellas hay algún programa de ordenador que siempre termina y nos da una respuesta correcta sí/no. Por ejemplo, para saber si una fórmula F dada es satisfactible, podemos construir su tabla de verdad con la lista de todas las posibles interpretaciones I para el vocabulario $\mathcal{P}$ sobre el que F está construido, y averiguar si hay algún 1 en la columna de F, esto es, si $eval_I(F) = 1$ para alguna I.

Similarmente, tenemos $F \models G$ para dos fórmulas F y G dadas si y sólo si en la tabla de verdad (para el vocabulario $\mathcal{P}$ de F y G), por cada fila con un 1 en la columna de F también hay un 1 en la columna de G.

Sin embargo, estos métodos basados en la tabla de verdad no son los más eficientes. En los ejercicios veremos que cualquiera de estas propiedades se puede expresar en términos del problema de la satisfactibilidad (abreviado, *SAT*), esto es, el de determinar si una fórmula proposicional dada es satisfactible o no. Por ejemplo, para dos fórmulas F y G tenemos $F \models G$ si, y sólo si, $F \wedge \neg G$ es insatisfactible. Por eso, para las aplicaciones prácticas se usa un *SAT solver*, esto es, un programa de ordenador que, dada una fórmula F, decide si F es satisfactible o no.

5. Ejercicios

1. (dificultad 1) ¿Cuántas interpretaciones posibles hay en función de $|\mathcal{P}|$?
   (nota: si S es un conjunto, $|S|$ denota su *cardinalidad*, es decir, el número de elementos de S).

2. (dificultad 1) Demuestra que $p \wedge \neg p$ es insatisfactible.

3. (dificultad 1) Demuestra que $p \vee \neg p$ es una tautología.

4. (dificultad 1) Escribe la tabla de verdad para las fórmulas $(p \wedge \neg(q \vee r))$ y $(\neg p \vee (q \vee r))$. Di, por cada una de las fórmulas si es tautología o insatisfactible y averigua las posibles relaciones de consecuencia o equivalencia lógica entre ellas.

5. (dificultad 1) Sean F y G dos fórmulas. ¿Es cierto que $F \vee G$ es tautología si y sólo si alguna de las dos fórmulas F o G lo es? Demuéstralo.

6. (dificultad 2) Sea F una fórmula. Demuestra que F es tautología si y sólo si $\neg F$ es insatisfactible.

7. (dificultad 2) Sean F y G dos fórmulas. Demuestra que F es consecuencia lógica de G si y sólo si $G \wedge \neg F$ es insatisfactible.

8. (dificultad 2) Sean F y G dos fórmulas. Demuestra que F es lógicamente equivalente a G si y sólo si $G \leftrightarrow F$ es tautología si y sólo si $(G \wedge \neg F) \vee (F \wedge \neg G)$ es insatisfactible.

9. (dificultad 2) Da un ejemplo de tres fórmulas F_1, F_2, y F_3 tales que $F_1 \wedge F_2 \wedge F_3$ sea insatisfactible y donde cualquier conjunción de todas ellas menos una sea satisfactible. Generalízalo a n fórmulas.

10. (dificultad 1) Demuestra que son tautologías:

    a) $(p \wedge (p \to q)) \to q$

    b) $((p \to q) \wedge \neg q) \to \neg q$

11. (dificultad 2) Dada una fórmula F, si $c(F)$ denota el número de sus conectivas binarias (contando repeticiones) y $p(F)$ el número de sus símbolos de predicado (contando repeticiones), demuestra que $p(F) = c(F) + 1$.

12. (dificultad 1) Demuestra de la forma más sencilla que encuentres:
    $(\neg q \vee \neg r \vee p) \wedge (p \vee q) \wedge (r \vee p) \quad \not\equiv \quad p \vee q$

13. (dificultad 1) Demuestra de la forma más sencilla que encuentres:
    $(p \vee q) \wedge (r \vee p) \wedge (\neg q \vee \neg r \vee p) \quad \equiv \quad p$

14. (dificultad 3) Sea F una fórmula que no contiene el símbolo $\neg$. Demuestra que si el número de símbolos de predicado es $k + 1$ entonces la longitud de la fórmula es $4k + 1$.

15. (dificultad 2) Si $F \to G$ es tautología y F es tautología, entonces ¿G es tautología? Demuéstralo.

16. (dificultad 2) Si $F \to G$ es satisfactible y F es satisfactible, entonces ¿G es satisfactible? Demuéstralo.

17. (dificultad 2) Si $F \to G$ es tautología y F satisfactible, entonces ¿G es satisfactible? Demuéstralo.

18. (dificultad 2) Demuestra las siguientes equivalencias entre fórmulas:

$F \wedge F$	$\equiv F$	idempotencia de $\wedge$
$F \vee F$	$\equiv F$	idempotencia de $\vee$
$F \wedge G$	$\equiv G \wedge F$	conmutatividad de $\wedge$
$F \vee G$	$\equiv G \vee F$	conmutatividad de $\vee$
$F \wedge (F \vee G)$	$\equiv F$	absorción 1
$F \vee (F \wedge G)$	$\equiv F$	absorción 2
$(F \wedge G) \wedge H$	$\equiv F \wedge (G \wedge H)$	asociatividad de $\wedge$
$(F \vee G) \vee H$	$\equiv F \vee (G \vee H)$	asociatividad de $\vee$
$(F \wedge G) \vee H$	$\equiv (F \vee H) \wedge (G \vee H)$	distributividad 1
$(F \vee G) \wedge H$	$\equiv (F \wedge H) \vee (G \wedge H)$	distributividad 2
$\neg\neg F$	$\equiv F$	doble negación
$\neg(F \wedge G)$	$\equiv \neg F \vee \neg G$	ley de De Morgan 1
$\neg(F \vee G)$	$\equiv \neg F \wedge \neg G$	ley de De Morgan 2
Si F es tautología entonces:		
$F \wedge G$	$\equiv G$	ley de tautología 1
$F \vee G$	$\equiv F$	ley de tautología 2
Si F es insatisfactible entonces:		
$F \wedge G$	$\equiv F$	ley de insatisfactible 1
$F \vee G$	$\equiv G$	ley de insatisfactible 2

19. (dificultad 2) Dadas fórmulas F, A, B, ¿es cierto que $F \wedge (A \vee B)$ es satisfactible si y sólo si al menos una de las fórmulas $F \wedge A$ y $F \wedge B$ lo es? Demuéstralo.

20. (dificultad 2) Demuestra que si $A \wedge B \models D$ y $A \wedge C \models D$ entonces $A \wedge (B \vee C) \models D$. ¿Es cierto el recíproco?

21. (dificultad 3) Una *relación binaria* sobre un conjunto S es un subconjunto del producto cartesiano $S \times S$, es decir, un conjunto de pares de elementos de S. Sea $\approx$ una relación binaria sobre S que escribiremos con notación infija, es decir, si para dos elementos a y b de S tenemos $(a, b) \in \approx$, escribiremos $a \approx b$. Se dice que $\approx$ es una *relación de equivalencia* en S si se cumplen las siguientes condiciones:

    a) Para cada $x \in S$ tenemos $x \approx x$ (*reflexividad*).

    b) Para cada $x, y \in S$, si $x \approx y$ entonces $y \approx x$ (*simetría*).

    c) Para cada $x, y, z \in S$, si $x \approx y$ y $y \approx z$ entonces $x \approx z$ (*transitividad*).

Demuestra que la equivalencia lógica $\equiv$ es una relación de equivalencia en el conjunto de las fórmulas proposicionales definidas sobre un conjunto $\mathcal{P}$ de símbolos. Demuestra también que es una relación compatible con las operaciones lógicas $\neg$, $\wedge$ y $\vee$; es decir:

a) Si $F \equiv G$ entonces $\neg F \equiv \neg G$.

b) Si $F_1 \equiv G_1$ y $F_2 \equiv G_2$ entonces $F_1 \wedge F_2 \equiv G_1 \wedge G_2$.

c) Si $F_1 \equiv G_1$ y $F_2 \equiv G_2$ entonces $F_1 \vee F_2 \equiv G_1 \vee G_2$.

22. (dificultad 2) Definimos recursivamente las subfórmulas de una fórmula F de la forma siguiente:

    - Si F es un símbolo de predicado p, entonces p es su única subfórmula.
    - Si F es $\neg G$, entonces las subfórmulas de F son F y las subfórmulas de G.
    - Si F es de la forma $(G \wedge H)$ o de la forma $(G \vee H)$, entonces las subfórmulas de F son F y las subfórmulas de G y las de H (contando con repetición, es decir, todas las ocurrencias).

    Demuestra que para toda fórmula F, el número de subfórmulas de F, denotado $nsf(F)$, cumple que $nsf(F) \leq |F|$, donde $|F|$ es el número de símbolos (conectivas, paréntesis y símbolos de predicado) de F.

23. (dificultad 3) Demuestra que si en una fórmula F sustituimos una ocurrencia de una subfórmula G por otra G' lógicamente equivalente a G, obtenemos una nueva fórmula F' que es lógicamente equivalente a F. Este resultado tiene un interés obvio para poder manipular fórmulas, por lo que tiene nombre: es el *Lema de Sustitución*.

24. (dificultad 2) Demuestra utilizando los resultados de los dos ejercicios anteriores que $q \vee (p \wedge (\neg p \vee r))$ es lógicamente equivalente a $(q \vee p) \wedge ((q \vee r) \vee (p \wedge \neg p))$.

25. (dificultad 2) Demuestra que para todas las fórmulas A, B, C se cumple que $A \wedge B \models C$ si y sólo si $A \models (B \rightarrow C)$.

26. (dificultad 2) Supongamos que $|\mathcal{P}| = 100$ y que nos interesa determinar si una fórmula F construida sobre $\mathcal{P}$ es satisfactible o no. Si el algoritmo está basado en un análisis de la tabla de verdad, y evaluar F en una interpretación I dada cuesta un microsegundo (10^{-6} segundos), ¿cuántos años tardará? Más adelante veremos técnicas que muchas veces funcionan mejor.

27. (dificultad 2) Una *función booleana* de n entradas es una función $f\colon \{0,1\}^n \rightarrow \{0,1\}$, es decir, una función que toma como entrada una cadena de n bits y devuelve un bit. ¿Cuántas funciones booleanas de n entradas hay?

28. (dificultad 2) Cada fórmula F representa una única función booleana: la que devuelve 1 exactamente para aquellas cadenas de bits I tales que $eval_I(F) = 1$. Por eso, dos fórmulas son lógicamente equivalentes si y sólo si representan la misma función booleana. ¿Cuántas funciones booleanas (o cuántas fórmulas lógicamente no-equivalentes) hay en función de $|\mathcal{P}|$?

29. (dificultad 2) Consideremos los siguientes símbolos de predicado:

    p es *María abusa de la comida basura*

    q es *Tomás abusa de la comida basura*

    r es *María está enferma*

    s es *Tomás está enfermo*

    t es *María fuma*

    u es *Tomás fuma*

    a) Traduce las siguientes fórmulas a tu idioma:

    1) $p \wedge \neg q$
    2) $(\neg p \wedge \neg t) \rightarrow \neg r$
    3) $(t \vee u) \rightarrow (t \wedge u)$
    4) $s \leftrightarrow (q \wedge u)$

    b) Traduce las siguientes frases a fórmulas de la lógica proposicional. Si en algún caso hay varias fórmulas posibles, indica si son lógicamente equivalentes o no. Nótese que, si no lo son, estamos ante un caso de ambigüedad del lenguaje natural.

    1) Tomás no fuma salvo que María lo haga.
    2) Ni Tomás ni María abusan de la comida basura.
    3) María esta sana si y sólo si no fuma y no abusa de la comida basura
    4) María no fuma si Tomás no lo hace
    5) Si María no fuma entonces está sana si no abusa de la comida basura

30. (dificultad 3) Demuestra para $n \geq 1$ las dos siguientes equivalencias lógicas:

    $(F_1 \wedge \ldots \wedge F_n) \vee G \equiv (F_1 \vee G) \wedge \ldots \wedge (F_n \vee G)$

    $(F_1 \vee \ldots \vee F_n) \wedge G \equiv (F_1 \wedge G) \vee \ldots \vee (F_n \wedge G)$

31. (dificultad 3) Escribe en una tabla de verdad las 16 funciones booleanas de 2 entradas. ¿Cuántas de ellas sólo dependen de una de las dos entradas? ¿Cuántas dependen de cero entradas? ¿Las otras, vistas como conectivas lógicas, tienen algún nombre? Ya sabemos que podemos expresar cualquier función booleana con el conjunto de tres conectivas $\{\wedge, \vee, \neg\}$, es decir, cualquier función booleana es equivalente a una fórmula construida sobre estas tres conectivas. ¿Es cierto esto también para algún conjunto de sólo dos de las 16 funciones? (Hay varias maneras, pero basta con dar una sola.)

32. (dificultad 3) Demuestra que cualquier función booleana de dos entradas se puede expresar con sólo *nor* o bien con sólo *nand*, donde $nor(F, G)$ es $\neg(F \vee G)$, y $nand(F, G)$ es $\neg(F \wedge G)$.

33. (dificultad 3) Tres estudiantes A, B y C son acusados de introducir un virus en la salas de ordenadores de la FIB. Durante el interrogatorio, las declaraciones son las siguientes:

- *A* dice: "*B* lo hizo y *C* es inocente"
- *B* dice: "Si *A* es culpable entonces *C* también lo es"
- *C* dice: "Yo no lo hice, lo hizo al menos uno de los otros"

a) ¿Son las tres declaraciones contradictorias?

b) Asumiendo que todos son inocentes, ¿quién o quiénes mintieron en la declaración?

c) Asumiendo que nadie mintió, ¿quién es inocente y quién es culpable?

34. (dificultad 2) Inventa y define formalmente alguna otra lógica distinta a la lógica proposicional. Por ejemplo, si las interpretaciones son funciones $I: \mathcal{P} \to \{0, 1, \bot\}$ que también pueden dar "indefinido" $\bot$, se puede adaptar la noción de satisfacción de manera razonable, aunque la respuesta ya no será binaria: la "evaluación" de una fórmula F en una interpretación I puede dar 1 (I satisface F) o 0 (I no satisface F) o $\bot$ (indefinido).

35. (dificultad 2) Como el ejercicio anterior, pero considerando $I: \mathcal{P} \to [0 \ldots 1]$, es decir, la interpretación de un símbolo p es una probabilidad (un número real entre 0 y 1). En este caso, la evaluación de una fórmula F en una interpretación I puede dar algo (remotamente) parecido a la probabilidad de satisfacción de F en I. En la lógica que has definido, ¿la evaluación de F en una I determinada, y la de $F \wedge F$ en esa misma I dan el mismo resultado?

36. (dificultad 2) Tienes delante de tí tres cajas cerradas, dos de ellas vacías y una llena de oro. La tapa de cada caja contiene una afirmación respecto de su contenido. La caja *A* dice "El oro no está aquí", la caja *B* dice "El oro no está aquí" y la caja *C* dice "El oro está en la caja *B*". Si sabemos que una y sólo una las afirmaciones es cierta, qué caja contiene el oro?

37. (dificultad 3) Considera el siguiente fragmento de código, que devuelve un booleano:

```
int i;
bool a, b;
...
if (a and i>0)
  return b;
else if (a and i<=0)
  return false;
else if (a or b)
  return a;
else
  return (i>0);
```

Simplifícalo sustituyendo los valores de retorno por un solo valor de retorno que sea una expresión booleana en $i > 0$, a y b:

```
int i;
bool a, b;
return ...;
```

38. (dificultad 2) Demuestra que I satisface la fórmula $((\cdots(p_1 \leftrightarrow p_2) \leftrightarrow \cdots) \leftrightarrow p_n)$ (donde $n \geq 1$) si y sólo si asigna el valor 0 a un número par de símbolos de predicado de la fórmula.

39. (dificultad 3)

    a) La fórmula $(((p \to q) \to p) \to p)$ es una tautología?

    b) Si definimos recursivamente A_0 como $(p \to q)$ y A_{n+1} como $(A_n \to p)$, ¿para qué valores de n es A_n una tautología?

40. (dificultad 3) Sea A^* la fórmula resultante de intercambiar en A los símbolos $\wedge, \vee$ y reemplazar cada símbolo de predicado por su negación. Demuestra que A^* es lógicamente equivalente a $\neg A$.

41. (dificultad 2) Dada una cadena de símbolos w, escribimos $pa(w)$ para denotar el número de paréntesis de abrir que hay en w. Similarmente, escribimos $pc(w)$ para los de cerrar. Demuestra que para toda fórmula F tenemos $pa(F) = pc(F)$.

1. Formas normales y cláusulas

- **Fórmulas como conjuntos:** Sea F una fórmula construida sólo mediante la conectiva $\wedge$ a partir de subfórmulas $F_1 \ldots F_n$. Por ejemplo, F puede ser la fórmula $(F_1 \wedge ((F_2 \wedge (F_3 \wedge F_4)) \wedge F_5))$. Por las propiedades de asociatividad, conmutatividad e idempotencia del $\wedge$ podemos escribir F equivalentemente como $F_1 \wedge \ldots \wedge F_n$, o incluso como un *conjunto* $\{F_1, \ldots, F_n\}$, porque no importan los paréntesis (asociatividad), ni el orden de los elementos (conmutatividad), ni los elementos repetidos (idempotencia). Lo mismo pasa con la conectiva $\vee$.
- **Literales:** Un *literal* es un símbolo de predicado p (*literal positivo*) o un símbolo de predicado negado $\neg p$ (*literal negativo*).
- **CNF:** Una fórmula está en *forma normal conjuntiva* (CNF, del inglés) si es una conjunción de disyunciones de literales, es decir, si es de la forma

  $(l_{1,1} \vee \ldots \vee l_{1,k_1}) \ \wedge \ \ldots \ \wedge \ (l_{n,1} \vee \ldots \vee l_{n,k_n})$,

  donde cada $l_{i,j}$ es un *literal*.
- **DNF:** Una fórmula está en *forma normal disyuntiva* (DNF, del inglés) si es una disyunción de conjunciones de literales, es decir, si es de la forma

  $(l_{1,1} \wedge \ldots \wedge l_{1,k_1}) \ \vee \ \ldots \ \vee \ (l_{n,1} \wedge \ldots \wedge l_{n,k_n})$,

  donde cada $l_{i,j}$ es un *literal*.
- **Cláusulas:** Una *cláusula* es una disyunción de literales, es decir, una fórmula de la forma $l_1 \vee \ldots \vee l_n$, donde cada l_i es un literal, o, equivalentemente, una fórmula $p_1 \vee \ldots \vee p_m \vee \neg q_1 \vee \ldots \vee \neg q_n$, donde las p_i y q_j son símbolos de predicado.
- **Conjunto de cláusulas:** Una fórmula en CNF es pues una conjunción de cláusulas que puede verse como un *conjunto* de cláusulas.
- **Cláusula vacía:** La *cláusula vacía* es una cláusula $l_1 \vee \ldots \vee l_n$ donde $n = 0$, es decir, es la disyunción de cero literales. La cláusula vacía se suele denotar por $\square$. Nótese que, según la sintaxis de la lógica proposicional, la cláusula vacía no es una fórmula. Por eso, en esta hoja asumiremos las siguientes extensiones de la sintaxis y de la semántica.

  En cuanto a la sintaxis, definimos que, si $n \geq 0$, y $F_1, \ldots, F_n$ son fórmulas, entonces también son fórmulas $\bigwedge_{i \in 1..n} F_i$ y $\bigvee_{i \in 1..n} F_i$.

  En cuanto a la extensión correspondiente de la semántica, si I es una interpretación:

  $$eval_I(\ \textstyle\bigwedge_{i \in 1..n} F_i\ ) \ = \ min\{\ 1,\ eval_I(F_1),\ \ldots,\ eval_I(F_n)\ \} \text{ y}$$
  $$eval_I(\ \textstyle\bigvee_{i \in 1..n} F_i\ ) \ = \ max\{\ 0,\ eval_I(F_1),\ \ldots,\ eval_I(F_n)\ \}.$$

  Intuitivamente, esta definición refleja que la conjuncíon es cierta en I si lo son las n fórmulas: si $n = 0$, la conjunción es trivialmente cierta. Similarmente, la disyunción es cierta si lo es al menos una: si $n = 0$, la disyunción es trivialmente falsa.
- **Cláusula de Horn:** Una *cláusula de Horn* es una cláusula $p_1 \vee \ldots \vee p_m \vee \neg q_1 \vee \ldots \vee \neg q_n$ con $m \leq 1$, es decir, que tiene como máximo un literal positivo.

2. Ejercicios

1. (dificultad 2) Demuestra que, para toda fórmula F, hay al menos una fórmula lógicamente equivalente que está en DNF. Ídem para CNF. Ayuda: obtener las fórmulas en CNF y DNF a partir de la tabla de verdad para F.
2. (dificultad 3) Da una manera de calcular una fórmula $\hat{F}$ en CNF para una fórmula F dada (con $\hat{F} \equiv F$) sin necesidad de construir previamente la tabla de verdad. Ayuda: aplica equivalencias lógicas como las leyes de De Morgan y la distributividad y el Lema de Sustitución.

3. (dificultad 3) Cada fórmula de lógica proposicional puede verse como un circuito electrónico que tiene una puerta lógica por cada conectiva $\wedge$, $\vee$, $\neg$ que aparezca en la fórmula (aunque las fórmulas tienen estructura de árbol, mientras los circuitos en realidad permiten compartir subárboles repetidos, es decir, son grafos dirigidos acíclicos).

   El problema del *diseño lógico* consiste en encontrar un circuito adecuado que implemente una función booleana dada. Para conseguir circuitos *rápidos*, nos va bien representar la función booleana como una fórmula en CNF (o DNF), porque la *profundidad* del circuito será como máximo tres. Pero también es importante utilizar el mínimo número de conectivas (puertas lógicas). Los métodos de obtener CNFs vistos en los ejercicios anteriores, ¿nos dan la CNF más corta en ese sentido? ¿Se te ocurre alguna mejora?

4. (dificultad 1) La cláusula vacía $\square$ es el caso más sencillo de fórmula insatisfactible. Una CNF que es un conjunto de cero cláusulas, ¿es satisfactible o insatisfactible?

5. (dificultad 2) Demuestra que una cláusula es una tautología si, y sólo si, contiene a la vez p y $\neg p$ para un cierto símbolo proposicional p.

6. (dificultad 2) Sea S un conjunto de cláusulas con $\square \notin S$. Demuestra que S es satisfactible (dando un modelo para S) en cada una de las siguientes situaciones:

   a) Toda cláusula de S tiene algún literal positivo.

   b) Toda cláusula de S tiene algún literal negativo.

   c) Para todo símbolo de predicado p se cumple que: o bien p aparece sólo en literales positivos en S, o bien p aparece sólo en literales negativos en S.

7. (dificultad 2) Dados n símbolos proposicionales:

   a) ¿Cuántas cláusulas distintas (como conjuntos de literales) hay?

   b) ¿Cuántas de estas cláusulas son insatisfactibles?

   c) ¿Cuántas cláusulas distintas y que no son tautologías hay?

   d) ¿Cuántas cláusulas distintas que contienen exactamente un literal por cada símbolo proposicional hay?

8. (dificultad 4) Propón un algoritmo eficiente que, dado un conjunto de cláusulas S, retorna un conjunto de cláusulas S' (no necesariamente definido sobre los mismos símbolos de predicado que S) con como mucho 3 literales por cláusula, que es *equisatisfactible* a S (es decir, que es satisfactible si y sólo si S lo es). Ayuda: es posible introducir algún símbolo de predicado p nuevo, que signifique: "$l \vee l'$ es cierto" para algún par de literales l y l'.

9. (dificultad 2) Sea S un conjunto de cláusulas de Horn con $\square \notin S$. Demuestra que S es satisfactible (dando un modelo para S) si no hay ninguna cláusula que sólo conste de un único literal positivo.

10. (dificultad 2) Demuestra que el enunciado del ejercicio previo es falso cuando S no es de Horn.

11. (dificultad 3) Definimos: un literal en una fórmula en CNF es *puro* si aparece o bien siempre negado o bien siempre sin negar. Además, se dice que una cláusula C es *redundante* si contiene al menos un literal puro. Demuestra que, si C' es una cláusula redundante, entonces $\{C_1, \ldots, C_n, C'\}$ es satisfactible si y sólo si $\{C_1, \ldots, C_n\}$ es satisfactible.

3. Nociones informales de decidibilidad y complejidad

Para una interpretación I y una fórmula F dadas, podemos determinar *eficientemente* mediante un programa de ordenador si I satisface F o no. En cambio, son muy *costosos* otros problemas, como el de decidir si una fórmula F dada es satisfactible. Para comprender mejor qué queremos decir con palabras

como "eficiente" o "costoso", aquí a continuación damos *a nivel intuitivo e informal* algunas nociones sobre el *coste* de resolver ciertos problemas. Todos estos conceptos serán formalizados en detalle en otras asignaturas posteriores.

Consideraremos *algoritmos*, o programas, expresados en un lenguaje de programación como `C`, `Java`, o `C++`, que, para resolver un *problema computacional*, toman una *entrada* y calculan una *salida* mediante una secuencia de *pasos* de cómputo. Cada paso se supone que es muy sencillo, por ejemplo, una instrucción de lenguaje máquina o un número pequeño acotado de ellas (como veremos, esta distinción no es muy relevante).

En los problemas llamados *decisionales*, la salida es una respuesta de tipo sí/no. Por ejemplo, es decisional el problema cuya entrada es una lista de números, y que consiste en determinar si su suma es par o no. En cambio, no es decisional el problema de escribir la suma de los números, ni el de escribir la lista de números ordenados de menor a mayor.

Como sabemos, la lógica proposicional es especialmente interesante porque todos los problemas decisionales típicos (por ejemplo, si una fórmula es satisfactible, o si es tautología, etc.) son *decidibles*: para cada uno de ellos hay algún programa de ordenador que siempre termina y nos da una respuesta correcta sí/no.

Por ejemplo, consideremos el problema decisional cuya entrada son dos fórmulas F y G, y que consiste en determinar si son lógicamente equivalentes o no. Un algoritmo puede *decidir* este problema simplemente construyendo su *tabla de verdad*, la lista de todas las posibles interpretaciones I para el conjunto $\mathcal{P}$ de símbolos de predicado que aparecen en F y G, y averiguar si para todas ellas tenemos que $eval_I(F) = eval_I(G)$.

Esto es posible en lógica proposicional porque se cumplen dos propiedades básicas: por un lado, para un conjunto de símbolos $\mathcal{P}$ finito, el número de interpretaciones es finito (aunque puede ser muy grande!) y, por otro, dada una interpretación I y una fórmula F, es también decidible si I satisface F. Como veremos, estas dos propiedades no se suelen tener en lógicas con mayor *poder expresivo*, como, por ejemplo, la lógica de primer orden, que permiten describir/modelar más cosas de la vida real.

3.1. Lo importante es el coste como función del tamaño de la entrada

Volvamos ahora a las cuestiones de eficiencia. Aquí consideraremos sólo el coste computacional en cuanto a *tiempo*, es decir, el número de pasos de cómputo, sin entrar en otros recursos como la cantidad de memoria de ordenador necesaria.

Evidentemente, para todo problema, resulta más costoso (son necesarios más pasos) resolverlo cuando la entrada es grande que cuando es pequeña. Por ejemplo, cuesta más sumar un millón de números que sumar cien.

A modo de ejemplo, supongamos que tenemos en memoria una tabla `int A[N]` con N números enteros distintos, ordenados de menor a mayor, y necesitamos un algoritmo que encuentre en qué posición de la tabla se encuentra el número cero. Veamos dos posibles soluciones.

Algoritmo 1: Búsqueda lineal: podemos recorrer la tabla con un bucle como:

```
while (A[i]!=0) i++;
```

En el caso peor, que se da cuando el cero es el último elemento, el número de pasos será proporcional al número de veces que se ejecuta el bucle, que es proporcional a N, por lo que se dice que *el algoritmo 1 tiene coste lineal en N*, o que funciona en *tiempo lineal*, o simplemente, que *es lineal*.

Algoritmo 2: Búsqueda dicotómica: puesto que `A` está ordenado, también podemos primero inspeccionar sólo el elemento central `A[N/2]`; así ya sabemos en qué mitad de la tabla está el cero (a la izquierda o a la derecha de `A[N/2]`); luego inspeccionamos el centro de esa mitad, y así sucesivamente, reduciendo en cada iteración a la mitad el trozo de tabla donde tenemos que buscar. No es difícil de ver que el número máximo de iteraciones será proporcional a $\log N$, el logaritmo en base 2 de N. Por ejemplo, si $N \leq 2^5 = 32$, en 5 iteraciones encontraremos dónde está el cero. Por eso se dice que el algoritmo `2` tiene *coste logarítmico* en N (o que funciona en tiempo logarítmico, o que es logarítmico).

Si la N es pequeña, no está claro cuál de estos dos algoritmos es mejor. Quizás en ese caso la opción `1` es más rápida porque probablemente necesita menos instrucciones por cada iteración. Pero, aunque el

algoritmo 2 necesite muchas más instrucciones por iteración que el algoritmo 1, *a partir de cierto tamaño de la entrada N suficientemente grande, siempre será mejor un algoritmo logarítmico que uno lineal*. Por ejemplo, si $N \geq 2^{10} = 1024$, el algoritmo 2 será más rápido incluso si cada iteración suya necesita 100 veces más pasos que cada iteración del algoritmo 1!

Por eso, *lo que verdaderamente importa es cuán rápido crece el coste del algoritmo en función del tamaño de la entrada*, más que el número exacto de pasos de cómputo que necesita, o el número exacto de instrucciones de lenguaje máquina o ciclos de reloj de procesador que necesite cada paso.

Por orden creciente de coste, algunas funciones típicas son: $\log N$ (coste *logarítmico*); N (*lineal*); N^2 (*cuadrático*: el coste es proporcional a cierto polinomio de grado 2, por ejemplo, $7N^2 + 3N$); N^3 (*cúbico*, un polinomio de grado 3); o incluso 2^N (*exponencial*). Si el coste es proporcional a un polinomio en N, o inferior, (logarítmico, lineal, cuadrático, cúbico, etc.) se dice que es *polinómico*.

Es fácil de ver que, conforme crece la N, el coste de los algoritmos exponenciales crece muchísimo más deprisa que los polinómicos. Por ejemplo, 1000^2 es *sólo* un millón, ¡mientras que 2^{1000} es muy superior al número de átomos que hay en el universo! Por eso, para un algoritmo exponencial siempre habrá entradas relativamente pequeñas que resulten inabordables; tener un superordenador con un millón de procesadores sólo podrá producir mejoras en un factor constante de un millón, lo cual es irrelevante si un simple incremento en 1 del tamaño de la entrada duplica el número de pasos necesarios.

4. Ejercicios

12. (dificultad 3) Para una fórmula en DNF, ¿cuál es el mejor algoritmo posible para decidir si es satisfactible? ¿Qué coste tiene?

5. Resolución. Corrección y completitud

- Aquí denotaremos las cláusulas por mayúsculas C o D y escribiremos $l \vee C$ para denotar una cláusula que tiene un literal l y en la que C es la disyunción (la cláusula) de los demás literales.

- **Resolución:** Una *regla deductiva* nos dice cómo obtener (o *deducir*) ciertas fórmulas nuevas a partir de fórmulas dadas. Una regla deductiva concreta es la *resolución*: dadas dos cláusulas de la forma $p \vee C$ y $\neg p \vee D$ (las *premisas*), por resolución se deduce la nueva cláusula $C \vee D$ (la *conclusión*). Esto se suele escribir como:

$$\frac{p \vee C \qquad \neg p \vee D}{C \vee D} \qquad \textit{Resolución} \text{ (para lógica proposicional)}$$

- **Clausura bajo resolución**: Sea S un conjunto de cláusulas. La *clausura de S bajo resolución*, denotada por $Res(S)$, es el conjunto de todas las cláusulas que se pueden obtener con cero o más pasos de resolución a partir de cláusulas de S.

  Formalmente se define de la siguiente manera. Sea $Res_1(S)$ el conjunto de las cláusulas que se pueden obtener en exactamente un paso de resolución a partir de S:

$$Res_1(S) = \{C \vee D \mid p \vee C \in S,\ \neg p \vee D \in S\}$$

  es decir, el conjunto de todas las cláusulas $C \vee D$ tales que, para algún símbolo de predicado p, hay cláusulas en S de la forma $p \vee C$ y $\neg p \vee D$.

  Ahora definimos para toda $i \geq 0$ (por inducción):

$$\begin{array}{ll} S_0 & = S \\ S_{i+1} & = S_i \cup Res_1(S_i) \end{array} \qquad \text{y definimos:} \qquad Res(S) = \bigcup_{i=0}^{\infty} S_i$$

  Nótese que esta definición nos da una manera *efectiva* (práctica) de construir $Res(S)$.

- **Clausura bajo una regla deductiva cualquiera**: Sea R una regla deductiva (como, por ejemplo, la resolución) y sea S un conjunto de fórmulas. Denotamos por $R(S)$ la clausura de S bajo R: el conjunto de todas las fórmulas que se pueden obtener con cero o más pasos de deducción de R a partir de fórmulas de S.

- **Corrección y completitud de una regla deductiva:** Si S es un conjunto de fórmulas $\{F_1 \dots F_n\}$, a menudo consideraremos S como la conjunción $F_1 \wedge \dots \wedge F_n$; por ejemplo, escribiremos $S \models F$ en vez de $F_1 \wedge \dots \wedge F_n \models F$.

  Definimos: la regla deductiva R es *correcta* si mediante R sólo podemos deducir fórmulas nuevas que son consecuencias lógicas de las que ya tenemos: si para toda fórmula F y todo conjunto de fórmulas S, se cumple que $F \in R(S)$ implica $S \models F$.

  Definimos: la regla deductiva R es *completa* si mediante R podemos deducir todas las consecuencias lógicas: si para toda fórmula F y todo conjunto de fórmulas S, se cumple que $S \models F$ implica $F \in R(S)$.

  Nótese que es fácil definir una regla deductiva correcta: basta con decir que *ninguna* fórmula se deduce. Igualmente, es fácil definir una regla deductiva completa: basta con decir que *toda* fórmula se deduce. Lo difícil es definir una regla deductiva R que es tanto correcta como completa; en ese caso tenemos $S \models F$ si y sólo si $F \in R(S)$, es decir, que R nos permite deducir todas las consecuencias lógicas, y nada más.

- **Completitud refutacional de la resolución**: La resolución es *refutacionalmente completa*, es decir, si S es insatisfactible, entonces $\square \in Res(S)$.

6. Ejercicios

13. (dificultad 2) Utiliza resolución para demostrar que $p \to q$ es una consecuencia lógica de

$$\begin{array}{rcl} t & \to & q \\ \neg r & \to & \neg s \\ p & \to & u \\ \neg t & \to & \neg r \\ u & \to & s \end{array}$$

14. (dificultad 2) Demuestra por resolución que son tautologías:

    a) $p \to (q \to p)$

    b) $(p \wedge (p \to q)) \to q$

    c) $((p \to q) \wedge \neg q) \to \neg p$

    d) $((p \to q) \wedge \neg q) \to \neg q$

15. (dificultad 2) Demuestra que la resolución es correcta.

16. (dificultad 2) Demuestra que, para todo conjunto finito de cláusulas S, $Res(S)$ es un conjunto finito de cláusulas, si se consideran las cláusulas como conjuntos de literales (por ejemplo, $C \vee p$ es la misma cláusula que $C \vee p \vee p$).

17. (dificultad 3) Sea S un conjunto de cláusulas. Demuestra que $Res(S)$ es lógicamente equivalente a S.

18. (dificultad 2) ¿La resolución es completa? Demuéstralo.

19. (dificultad 2) Sea S un conjunto de cláusulas insatisfactible. Por la completitud refutacional de la resolución, sabemos que existe una demostración por resolución de que $\square \in Res(S)$. ¿Es esta demostración única?

20. (dificultad 4) Demuestra la completitud refutacional de la resolución, esto es, si S es un conjunto de cláusulas insatisfactible entonces $\square \in Res(S)$.

    Ayuda: demuestra el contrarrecíproco por inducción sobre el número N de símbolos de predicado de S.

21. (dificultad 2) Demuestra que el lenguaje de las cláusulas de Horn es cerrado bajo resolución, es decir, a partir de cláusulas de Horn por resolución sólo se obtienen cláusulas de Horn.

22. (dificultad 2) Considera el siguiente caso particular de la resolución:

$$\frac{p \qquad \neg p \vee C}{C} \qquad \textit{Resolución Unitaria}$$

    Demuestra que la resolución unitaria es correcta.

23. (dificultad 2) Demuestra que la resolución unitaria no es refutacionalmente completa para cláusulas que no son de Horn.

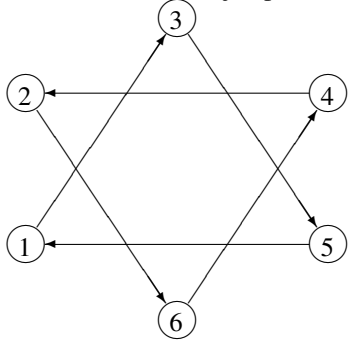
24. (dificultad 3) Demuestra que la resolución unitaria es refutacionalmente completa para cláusulas de Horn. Ayuda: Basta con ver que, si S es un conjunto de cláusulas de Horn y $\square \notin ResUnit(S)$, entonces $ResUnit(S)$ (y por lo tanto S) tiene un modelo I. Define I como $I(p) = 1$ si y sólo si p es una cláusula (de un solo literal) en $ResUnit(S)$ y demuestra $I \models ResUnit(S)$ por inducción sobre el número de literales de las cláusulas.

25. (dificultad 2) ¿Cuál es la complejidad del problema de determinar si un conjunto de cláusulas de Horn S es satisfactible? Ayuda: analiza (informalmente) la corrección y la complejidad del siguiente algoritmo (que intenta construir sistemáticamente el modelo *minimal* I de S):

    (0) inicialmente, $I(p) = 0$ para todo p
    (1) hacer ciertos en I los p que son cláusulas unitarias positivas;
    (2) eliminar de todas las cláusulas los literales $\neg p$ con $I(p) = 1$;
        si esto da lugar a la cláusula vacía: insatisfactible
        si no, si esto da lugar a alguna cláusula unitaria nueva, volver a (1)
            si no, la interpretación I construida es un modelo.

26. (dificultad 2) Las *cláusulas de Krom* son aquellas que tienen a lo sumo dos literales. ¿Cuántas cláusulas de Krom se pueden construir con n símbolos de predicado? Demuestra que basta un número cuadrático de pasos de resolución para decidir si un conjunto de cláusulas de Krom es satisfactible o no.

27. (dificultad 3) Un *grafo* $G = (V, E)$ es un conjunto V de objetos llamados *vértices* conectados por enlaces llamados *aristas*; la arista que une los vértices u y v se representa como el par de vértices (u, v), y el conjunto de aristas se denota por E. Además, se dice que un grafo está *dirigido* si se distingue entre las dos posibles orientaciones de las aristas. Por ejemplo, en el grafo dirigido siguiente:



el conjunto de vértices V es $\{1, 2, 3, 4, 5, 6\}$, y el de aristas E es $\{(1, 3), (3, 5), (5, 1), (2, 6), (6, 4), (4, 2)\}$. Finalmente, se dice que una secuencia de vértices v_0, ..., v_k es un *camino* si se tiene que los vértices v_0, ..., v_k están sucesivamente conectados, es decir, si $(v_{i-1}, v_i) \in E$ para todo $1 \leq i \leq k$. Por ejemplo, 1, 3 y 5 forman un camino del grafo dibujado más arriba, ya que $(1, 3) \in E$ y $(3, 5) \in E$. Los grafos son objetos muy importantes en matemáticas e informática y se estudian en detalle en varias asignaturas posteriores.

Dado S un conjunto de cláusulas con 1 ó 2 literales por cláusula, definido sobre los símbolos proposicionales p_1, ..., p_n, se define el *grafo asociado a* S, denotado G_S, como el grafo dirigido $G_S = (V, E)$, donde $V = \{p_1, \ldots, p_n, \neg p_1, \ldots, \neg p_n\}$ y $E = \{(l, l') \mid \neg l \vee l' \in S\}$ (en este ejercicio, abusando de la notación, dado un literal l de la forma $\neg p$, vamos a considerar que $\neg l$ representa p; además, en la construcción del grafo las cláusulas de 1 literal, o sea de la forma p, se consideran como $p \vee p$).

a) Demuestra que si hay un camino de l a l' en G_S, entonces $\neg l \vee l' \in Res(S)$. Recíprocamente, demuestra que si $l \vee l' \in Res(S)$, entonces hay un camino de $\neg l$ a l' en G_S.

b) Demuestra que S es insatisfactible si y sólo si existe un símbolo proposicional p tal que hay un camino en G_S de p a $\neg p$, y otro camino de $\neg p$ a p.

c) Basándote en el apartado previo, propón un algoritmo para determinar la satisfactibilidad de un conjunto de cláusulas con 1 ó 2 literales por cláusula. ¿Qué complejidad tiene, en términos del número de cláusulas y de símbolos proposicionales de S?

7. Resolver problemas prácticos con la lógica proposicional

En la Sección 3 vimos informalmente qué significa que un algoritmo tenga coste polinómico o exponencial. Hay una importante clase de problemas para los que no se han descubierto algoritmos polinómicos. Para los problemas de esta clase (los llamados problemas *NP-completos*, ver abajo) sólo se conocen algoritmos que, en el caso peor, necesitan un número exponencial de pasos. Esta clase incluye miles de problemas prácticos importantes que surgen, por ejemplo, al trazar rutas de transporte o redes de comunicación, asignar máquinas u otros recursos en procesos industriales, confeccionar horarios de hospitales, escuelas, líneas aéreas, cargar un camión o un barco, etc.

Uno de los problemas NP-completos más famosos es *SAT*: el problema de decidir si una fórmula de lógica proposicional dada es satisfactible o no. Los algoritmos para SAT, los llamados *SAT solvers*, están muy estudiados y a menudo son capaces de tratar fórmulas grandes. Por eso es muy útil saber que los SAT solvers pueden usarse también para intentar resolver casos concretos de los demás problemas NP-completos.

Por ejemplo, podemos expresar fácilmente el problema de resolver un Sudoku (otro problema NP-completo) como un problema de SAT. Vamos a hacerlo aquí con $9^3 = 729$ símbolos de predicado p_{ijk}, que significan: "en la fila i columna j del Sudoku hay el valor k", con $1 \leq i, j, k \leq 9$. Aquí tenemos un Sudoku:

5	7		6			3		9
	2		3		9		7	1
1				8				
	5		7		3		8	6
		6				4		
4	1		8		6		5	
				6				2
8	9		5		2		6	
2		3			4		1	8

Tenemos que expresar:

1. **En cada casilla [i,j] hay al menos un valor.** Para expresar esto, en nuestro problema de SAT incluimos cláusulas de la forma $p_{ij1} \vee p_{ij2} \vee \ldots \vee p_{ij9}$.
   Por ejemplo, la cláusula $p_{111} \vee p_{112} \vee \ldots \vee p_{119}$ expresa que: "en la casilla [1,1] hay un 1, o en la

casilla [1,1] hay un 2, o hay un 3, . . ., o hay un 9".
Para definir más formalmente qué cláusulas incluímos, escribimos:

Para todos los i, j con $1 \leq i, j \leq 9$ tenemos la cláusula $p_{ij1} \vee p_{ij2} \vee \ldots \vee p_{ij9}$

total: 81 claúsulas de 9 literales cada una.

2. **En cada casilla no hay más de un valor.** Para expresar esto, en nuestro problema de SAT incluimos muchas cláusulas de dos literales. Por ejemplo, la cláusula $\neg p_{111} \vee \neg p_{112}$ expresa que "en la casilla $[1, 1]$ no hay un 1 o en la casilla $[1, 1]$ no hay un 2" (es decir, si hay un 1 no hay un 2, y si hay un 2 no hay un 1). Tenemos que expresar esto para todas las casillas $[i, j]$, y todos los pares de valores distintos k y k'. Formalmente:

   Para todos los i, j con $1 \leq i, j \leq 9$, y

   para todos los k, k' con $1 \leq k < k' \leq 9$ tenemos la cláusula $\neg p_{ijk} \vee \neg p_{ijk'}$.

   total: $81 \cdot 36 = 2916$ cláusulas de dos literales.

   (por cada una de las 81 casillas, hay 36 pares k, k' posibles ya que el conjunto $\{1, \ldots, 9\}$ tiene $\binom{9}{2} = 36$ subconjuntos de 2 elementos).

3. **En cada fila (o columna, o cuadrado de 3x3) ningún valor se repite.** Para esto nuevamente incluimos cláusulas de dos literales. Por ejemplo, para las dos primeras casillas de la fila 1, la cláusula $\neg p_{111} \vee \neg p_{121}$ expresa que "en la casilla $[1, 1]$ no hay un 1 o en la casilla $[1, 2]$ no hay un 1". Formalmente, para las filas:

   Para todos los i, k con $1 \leq i, k \leq 9$, y

   para todos los j, j' con $1 \leq j < j' \leq 9$ tenemos la cláusula $\neg p_{ijk} \vee \neg p_{ij'k}$.

   total: $81 \cdot 36 = 2916$ cláusulas de dos literales para las filas,

   y dos veces 2916 más para las columnas y cuadrados de 3x3.

4. **Cada número ya puesto en el sudoku.** Tendremos cláusulas de un solo literal, como p_{115} (el 5 en la casilla $[1, 1]$, la superior izquierda). Estas son las únicas que cambian en cada Sudoku; las cláusulas de los puntos 1,2 y 3 son siempre las mismas.

En `www.lsi.upc.edu/~roberto/il.html` está disponible un sencillo programa (en el lenguaje de programación lógica Prolog, que veremos más adelante en esta asignatura) que expresa (o *traduce*) así Sudokus como problemas de SAT. También hay un SAT solver llamado *Siege*, que resuelve en 0.01 segundos este problema concreto así generado, de 11781 cláusulas, y otro programa Prolog que toma como entrada el modelo que hemos encontrado para el problema de SAT y *lo traduce a una solución del problema de Sudoku que teníamos*.

En la lista de ejercicios veremos cómo resolver mediante SAT otros problemas NP-completos, siempre traduciéndolos directamente a una CNF, es decir, a un conjunto de cláusulas. Si para esta CNF se encuentra una solución (un modelo), podremos hacer la traducción al revés para reconvertirlo en una solución para nuestro problema original!

Un poco de cultura informal sobre los problemas NP-completos. Se dice que un problema *está en NP* si hay algún algoritmo *No-determinista Polinómico* que lo resuelve. Informalmente, esto significa que en tiempo polinómico podemos "adivinar" una posible solución y comprobar si efectivamente lo es. Por ejemplo, en SAT las posibles soluciones son las interpretaciones I; podemos generar una I mediante $|\mathcal{P}|$ "adivinanzas" binarias 1/0, y verificar si esta I concreta es solución para la fórmula dada F (verificar si $I \models F$) es también polinómico. Un algoritmo para SAT que simplemente pruebe todas las interpretaciones posibles será exponencial, porque hay $2^{|\mathcal{P}|}$ de ellas!

Un problema que está en NP se dice que es *NP-completo* si además es posible utilizarlo para *expresar* en tiempo polinómico cualquier otro problema de NP (por ejemplo, hemos usado SAT para expresar el problema de los Sudokus). Como ya hemos dicho, hay miles de problemas prácticos importantes que son

NP-completos. Hoy día no se sabe si es posible resolver los problemas NP-completos en tiempo polinómico, pero se piensa que no[1]. Si tuviéramos un algoritmo polinómico para sólo uno de los miles de problemas NP-completos, ya lo tendríamos para todos, porque podríamos expresarlos todos en términos de ése!

La instancia (o entrada) concreta de SAT obtenida a partir del Sudoku de nuestro ejemplo es relativamente fácil de resolver. Pero, dada la NP-completitud del problema de SAT, no es ninguna sorpresa que existan instancias de SAT no muy grandes que ni los mejores SAT solvers son capaces de tratar en, digamos, una semana. Sin embargo, los SAT solvers actuales a menudo pueden resolver instancias de SAT relativamente grandes. En la siguiente sección veremos cómo funcionan estos algoritmos.

8. Ejercicios

28. (dificultad 3) Dado un mapa de un continente, es posible colorearlo con cuatro colores sin que dos países con frontera común tengan el mismo color (es el famoso *four color problem*). Para grafos, el problema se generaliza al problema NP-completo de *K-coloreado:* dado un grafo G y un número natural K, decidir si podemos asignar a cada vértice un natural entre 1 y K (un *color*), tal que todo par de vértices adyacentes tengan colores distintos.

    Expresa este problema mediante una CNF, de modo que se pueda resolver con un SAT solver. ¿Cuántos símbolos de predicado se necesitan? ¿Cuántas cláusulas se obtienen?

29. (dificultad 3) Dados un edificio de una sola planta con muchos pasillos rectos que se cruzan, y un número natural K, ¿se pueden colocar cámaras giratorias en los cruces de los pasillos de modo que sea posible vigilar todos los pasillos con como mucho K cámaras? Este problema se puede formalizar como el problema de grafos llamado *vertex cover* o, traducido, *recubrimiento de vértices*: ¿existe un subconjunto de tamaño como mucho K de los N vértices, el *recubrimiento*, tal que toda arista tenga al menos un extremo en el recubrimiento (es decir, quede *cubierta*)?

    Expresa este problema mediante una CNF, de modo que se pueda resolver con un SAT solver. Usa los $K \cdot N$ símbolos de predicado $p_{i,j}$ que signifiquen: "el i-ésimo miembro del recubrimiento (con i entre 1 y K) es el vértice j (con j entre 1 y N)". ¿Cuántas cláusulas se obtienen?

30. (dificultad 5) Siguiendo el problema anterior, si hubiésemos usado la codificación con símbolos de predicado p_i que significasen: "hay una cámara en el vértice i", ¿cómo expresarías de forma compacta que no hay más de K cámaras? (Ayuda: piensa en circuitos sumadores y usa símbolos adicionales). Usando estos símbolos de predicado expresa el problema de *vertex cover* mediante una CNF, de modo que se pueda resolver con un SAT solver.

31. (dificultad 3) Dado un grupo de estudiantes con las listas de asignaturas que estudia cada uno, y un natural K, ¿hay algún subconjunto de exactamente K estudiantes tal que toda asignatura tenga algún estudiante que la curse?

    Expresa este problema mediante una CNF, de modo que se pueda resolver con un SAT solver. ¿Cuántos símbolos de predicado se necesitan? ¿Cuántas cláusulas se obtienen?

32. (dificultad 3) ¿Cuál crees que es el coste mínimo de un algoritmo que calcule una DNF lógicamente equivalente para una fórmula F?

9. El procedimiento de Davis-Putnam-Logemann-Loveland (DPLL)

Casi todos los SAT solvers actuales (como Siege) utilizan variantes modernas del algoritmo de Davis-Putnam-Logemann-Loveland (DPLL). Este algoritmo sirve para CNFs, es decir, para conjuntos de cláusulas. Gracias a los avances en algoritmos DPLL, los SAT solvers están siendo usados cada vez más para resolver todo tipo de problemas NP-completos prácticos.

[1] Éste es el famoso problema de "P vs. NP" (donde P significa polinómico), uno de los siete problemas matemáticos abiertos más importantes según el Clay Mathematics Institute, que ofrece un premio de un millón de dólares a quien lo resuelva, tanto si demuestra que sí es posible como si no; ver `www.claymath.org/millennium`.

Aquí presentaremos una versión sencilla del DPLL, basada en reglas, en la que el conjunto de cláusulas F dado no cambia a lo largo de la ejecución. El algoritmo explora de una manera compacta todas las posibles interpretaciones. En cada momento se tiene una interpretación parcial, representada como una secuencia de literales M, los que son ciertos en ese momento. M nunca contiene a la vez un literal l y su negado $\neg l$, ni tampoco contiene literales repetidos. Decimos que *una cláusula C es falsa en M* si $\neg l \in M$ para todo literal l de C. La secuencia M se va extendiendo *decidiendo* (o adivinando) nuevos literales, y cada vez que una cláusula se vuelve falsa en M, se *invierte* la última decisión tomada (esto se llama *backtracking*). A veces un literal l figura *marcado* como l^{d}. Esta marca significa que se trata de un literal de *decisión* (adivinado), lo cual indica que aún debe ser probado también su negado. Inicialmente, M es la secuencia vacía $\emptyset$, y el algoritmo simplemente va aplicando cualquier regla de las cuatro siguientes:

Propaga :

$M \quad \Longrightarrow \quad M\ l \qquad$ **SI** $\left\{ \begin{array}{l} \text{En } F \text{ hay alguna cláusula } l \vee C \text{ cuya parte } C \\ \text{es falsa en } M, \text{ y ni } l \text{ ni su negado están en } M. \end{array} \right.$

Decide :

$M \quad \Longrightarrow \quad M\ l^{\mathsf{d}} \qquad$ **SI** $\left\{ \begin{array}{l} \text{El literal } l \text{ o su negado aparece en } F, \text{ y ni } l \text{ ni} \\ \text{su negado están en } M. \end{array} \right.$

Falla :

$M \quad \Longrightarrow \quad$ "*Insat*" **SI** $\left\{ \begin{array}{l} \text{En } F \text{ hay alguna cláusula que es falsa en } M, \text{ y} \\ M \text{ no contiene literales de decisión.} \end{array} \right.$

Backtrack :

$M\ l^{\mathsf{d}}\ N \quad \Longrightarrow \quad M\ \neg l \qquad$ **SI** $\left\{ \begin{array}{l} \text{En } F \text{ hay alguna cláusula que es falsa en} \\ M\ l^{\mathsf{d}}\ N, \quad \text{y } N \text{ no contiene literales de decisión.} \end{array} \right.$

Nótese que, en la última regla, el literal $\neg l$ ya no está marcado como decisión, porque su negado ya ha sido probado. La regla Propaga aprovecha que a menudo no hace falta adivinar: para que la cláusula $l \vee C$ se haga cierta, estamos *forzados* a poner l a cierto (se dice que *propagamos* la información de la que disponemos en M). Por motivos de eficiencia, es bueno aplicar Falla y Backtrack con mayor preferencia, y después Propaga. Si F es un conjunto finito de cláusulas, tenemos los siguientes resultados:

1. Cualquier aplicación de las reglas *termina*, es decir, no existe ninguna secuencia infinita de aplicaciones: $\emptyset \Longrightarrow M_1 \Longrightarrow M_2 \Longrightarrow \ldots$
2. Si $\emptyset \Longrightarrow \ldots \Longrightarrow$ "*Insat*", entonces F es insatisfactible.
3. Si $\emptyset \Longrightarrow \ldots \Longrightarrow M$ y a M no se le puede aplicar ninguna regla, entonces M es un modelo de F.

Ejemplo: Sea F el siguiente conjunto de cláusulas (donde los símbolos de predicado se representan como naturales, y la negación con una rayita):

$$\begin{array}{lllllllllll} 1. & 1 & \vee & \overline{2} & \vee & 3 & \vee & \overline{4} & \vee & \overline{5} \\ 2. & 1 & \vee & & & 3 & \vee & 4 & \vee & 5 \\ 3. & 1 & \vee & & & \overline{3} & \vee & 4 & & \\ 4. & 1 & \vee & & & 3 & \vee & \overline{4} & \vee & 5 \\ 5. & 1 & \vee & & & \overline{3} & \vee & \overline{4} & & \\ 6. & \overline{1} & \vee & \overline{2} & & & & & & \\ 7. & & & 2 & & & & & & \\ 8. & & & \overline{2} & \vee & 3 & \vee & 4 & \vee & \overline{5} \end{array}$$

Anotando cada $\Longrightarrow$ con la primera letra de la regla aplicada (Propaga, Decide, Falla, o Backtrack), y el número de la cláusula usada, tenemos:

$$\emptyset \ \Longrightarrow_{p7}\ 2 \ \Longrightarrow_{p6}\ 2\,\overline{1} \ \Longrightarrow_{d}\ 2\,\overline{1}\,3^{\mathsf{d}} \ \Longrightarrow_{p5}\ 2\,\overline{1}\,3^{\mathsf{d}}\,\overline{4} \ \Longrightarrow_{b3}\ 2\,\overline{1}\,\overline{3} \ \Longrightarrow_{d}$$

$$2\,\overline{1}\,\overline{3}\,4^{\mathsf{d}} \;\Longrightarrow_{p1}\; 2\,\overline{1}\,\overline{3}\,4^{\mathsf{d}}\,\overline{5} \;\Longrightarrow_{b4}\; 2\,\overline{1}\,\overline{3}\,\overline{4} \;\Longrightarrow_{p2}\; 2\,\overline{1}\,\overline{3}\,\overline{4}\,5 \;\Longrightarrow_{f8}\; \text{“}\mathit{Insat}\text{”}$$

con lo cual hemos demostrado su insatisfactibilidad. Se invita al lector a comprobar el trabajo necesario para hacer lo mismo mediante resolución u otros métodos deductivos. Sin la cláusula 8., y con la misma secuencia de pasos, el algoritmo habría acabado después del penúltimo paso, encontrando el modelo $2\,\overline{1}\,\overline{3}\,\overline{4}\,5$.

10. Ejercicios

33. (dificultad 2) Di cuáles de las siguientes fórmulas son satisfactibles utilizando el procedimiento DPLL:

    a) $(p \vee \neg q \vee r \vee \neg s) \wedge (\neg r \vee s) \wedge q \wedge \neg p$

    b) $(p \vee q) \wedge (\neg p \vee \neg q) \wedge (p \vee r) \wedge (\neg p \vee \neg r)$

    c) $(p \vee q) \wedge (\neg p \vee \neg q) \wedge (p \vee r) \wedge (\neg p \vee \neg r) \wedge (q \vee r) \wedge (\neg q \vee \neg r)$

34. (dificultad 2) Utiliza el procedimiento DPLL para demostrar que $p \to q$ es una consecuencia lógica de

$$\begin{array}{rcl} t & \to & q \\ \neg r & \to & \neg s \\ p & \to & u \\ \neg t & \to & \neg r \\ u & \to & s \end{array}$$

35. (dificultad 2) Demuestra que son tautologías utilizando el procedimiento DPLL:

    a) $p \to (q \to p)$

    b) $(p \wedge (p \to q)) \to q$

    c) $((p \to q) \wedge \neg q) \to \neg p$

    d) $((p \to q) \wedge \neg q) \to \neg q$

36. (dificultad 3) (*Invariantes de DPLL*) Supongamos que se aplica el procedimiento DPLL a un conjunto de cláusulas F y que se obtiene la traza $\emptyset \Longrightarrow \cdots \Longrightarrow M$, con $M \neq$ “*Insat*”. Demuestra que entonces se cumplen las propiedades siguientes:

    a) Todos los símbolos proposicionales en M son símbolos proposicionales de F.

    b) M no contiene ningún literal más de una vez y no contiene p y $\neg p$ para ningún símbolo proposicional p.

    c) Si M es de la forma $N_0\; l_1^{\mathsf{d}}\; N_1\; l_2^{\mathsf{d}}\; \cdots\; l_n^{\mathsf{d}}\; N_n$, donde l_1, ..., l_n son los literales de decisión de M, entonces $F \cup \{l_1, \ldots, l_i\} \models N_i$ para cada $i = 0 \ldots n$, interpretando N_i como la conjunción de todos sus literales.

37. (dificultad 3) (*Corrección y completitud del procedimiento DPLL*) Supongamos que se aplica el procedimiento DPLL a un conjunto de cláusulas F, y que se obtiene la traza $\emptyset \Longrightarrow \cdots \Longrightarrow M$, a partir de donde ya no se puede aplicar más ninguna de las reglas Propaga, Decide, Falla o Backtrack. Utilizando los invariantes de DPLL, demuestra que:

    a) Si $M =$ “*Insat*” entonces F es insatisfactible.

    b) Si $M \neq$ “*Insat*” entonces $M \models F$ (y en particular F es satisfactible).

38. (dificultad 4) (*Terminación del procedimiento DPLL*) Supongamos que se aplica el procedimiento DPLL a un conjunto de cláusulas. Demuestra que no existen secuencias infinitas de la forma $\emptyset \Longrightarrow \cdots$

39. (dificultad 3) El problema de *AllSAT* consiste en obtener todos los modelos de una fórmula proposicional F. Desarrolla un algoritmo para *AllSAT* utilizando llamadas independientes al procedimiento DPLL.

1. Lógica de primer orden

En esta sección reproducimos y extendemos para la lógica de primer orden todas las definiciones y resultados vistos en la sección previa para la lógica proposicional. Haremos especial hincapié en las mayores posibilidades en cuanto a poder expresivo, y en el precio a pagar a cambio: la pérdida de la decidibilidad para la mayor parte de los problemas interesantes.

1.1. Sintaxis

Si en la lógica proposicional únicamente disponíamos de un conjunto de símbolos de predicado, aquí tenemos un vocabulario más rico:

- sea $\mathcal{F}$ un conjunto de *símbolos de función*, que denotaremos por $f, g, h, \ldots$
- sea $\mathcal{P}$ un conjunto de *símbolos de predicado*, denotados aquí por $p, q, r, \ldots$
- sea $\mathcal{X}$ un conjunto de *símbolos de variable*, denotados aquí por $x, y, z, \ldots$

Cada símbolo de función f y cada símbolo de predicado p tiene asociado un número natural, que es su *aridad*. Si la aridad de un símbolo f (o p) es n, solemos escribir f^n (o p^n) para indicar este hecho. A los símbolos de función de aridad 0 se los llama *constantes*, y se suelen denotar con la letra c.

Los *términos* se definen como sigue:

- toda variable es un término
- toda constante (símbolo de función de aridad cero) es un término
- $f(t_1, \ldots, t_n)$ es un término si $t_1, \ldots, t_n$ son términos y f es un símbolo de función de aridad n con $n > 0$.
- nada más es un término

Los *átomos* se definen como sigue:

- todo símbolo de predicado de aridad cero es un átomo
- $p(t_1, \ldots, t_n)$ es un átomo si $t_1, \ldots, t_n$ son términos y p es un símbolo de predicado de aridad n con $n > 0$.
- nada más es un átomo

Las *fórmulas de la lógica de primer orden* se definen como sigue:

- todo átomo es una fórmula
- si F y G son fórmulas y x es una variable, entonces son fórmulas:
  $\neg F \qquad (F \vee G) \qquad (F \wedge G) \qquad \forall x\, F \qquad \exists x\, F$
- nada más es una fórmula

Los símbolos $\forall$ y $\exists$ son los *cuantificadores universal y existencial* respectivamente.

1.2. Interpretación

Una *interpretación en lógica de primer orden I* consta de:

- Un conjunto no vacío D_I, llamado el *dominio* de I
- Por cada símbolo de función f de aridad n, una función f_I (*la interpretación de f en I*), que, dados n valores del dominio, devuelve un resultado del dominio. Similarmente por cada símbolo de predicado p tenemos p_I, la interpretación de p en I, sólo que p_I devuelve un resultado Booleano (0 o 1):
  - un valor del dominio $c_I \in D_I$ para cada $c^0 \in \mathcal{F}$
  - una función $f_I\colon D_I \times \ldots^{n)} \times D_I \to D_I$ para cada $f^n \in \mathcal{F}$ con $n > 0$
  - un valor booleano $p_I \in \{0, 1\}$ para cada $p^0 \in \mathcal{P}$
  - una función $p_I\colon D_I \times \ldots^{n)} \times D_I \to \{0, 1\}$ para cada cada $p^n \in \mathcal{P}$ con $n > 0$

1.3. Satisfacción

Asignación: Dada una interpretación I, una *asignación* es una función $\alpha\colon \mathcal{X} \to D_I$ (*asigna* un valor del dominio a cada símbolo de variable).

Denotaremos por $\alpha[x \mapsto d]$ la asignación que es como α, excepto que a x le asigna d, es decir, es la asignación α' tal que $\alpha'(y) = \alpha(y)$ si $y \neq x$, y tal que $\alpha'(x) = d$.

Evaluación de términos: La *evaluación de un término en una interpretación I y una asignación α* viene dada por la función $eval_I^\alpha$ que para cada término devuelve un valor de D_I:

- si x es una variable entonces $eval_I^\alpha(x) = \alpha(x)$
- si $c^0 \in \mathcal{F}$ (símbolo de constante) entonces $eval_I^\alpha(c) = c_I$
- si $f(t_1, \ldots, t_n)$ es un término, $eval_I^\alpha(\ f(t_1, \ldots, t_n)\ ) = f_I(\ eval_I^\alpha(t_1),\ \ldots,\ eval_I^\alpha(t_n)\ )$.

Evaluación de fórmulas: Definimos la *evaluación de una fórmula en una interpretación I y una asignación α*. La denotamos igual que para términos, $eval_I^\alpha$, pero aquí es una función que para cada fórmula devuelve un valor de $\{0, 1\}$:

- si $p^0 \in \mathcal{P}$ (símbolo de predicado de aridad cero) entonces $eval_I^\alpha(p) = p_I$
- si $p(t_1, \ldots, t_n)$ es un átomo, $eval_I^\alpha(\ p(t_1, \ldots, t_n)\ ) = p_I(\ eval_I^\alpha(t_1),\ \ldots,\ eval_I^\alpha(t_n)\ )$
- $eval_I^\alpha(F \wedge G) = \text{mín}\{\ eval_I^\alpha(F),\ eval_I^\alpha(G)\ \}$
- $eval_I^\alpha(F \vee G) = \text{máx}\{\ eval_I^\alpha(F),\ eval_I^\alpha(G)\ \}$
- $eval_I^\alpha(\neg F) = 1 - eval_I^\alpha(F)$
- $eval_I^\alpha(\forall x F) = \text{mín}\{\ eval_I^{\alpha[x\mapsto d]}(F)\ \mid\ d \in D_I\ \}$
- $eval_I^\alpha(\exists x F) = \text{máx}\{\ eval_I^{\alpha[x\mapsto d]}(F)\ \mid\ d \in D_I\ \}$

Noción de satisfacción (de F en I y α): Una interpretación I y una asignación α *satisfacen* F si $eval_I^\alpha(F) = 1$.

1.4. Definición de la Lógica de primer orden: fórmulas cerradas

Apariciones libres y ligadas de variables: En $\forall y \exists x\, p(x,y) \wedge q(x,z)$ se dice que la aparición de x en $p(x,y)$ está *ligada* al cuantificador existencial, y la aparición de x en $q(x,z)$ es *libre*. La aparición de la variable z es también libre. Si denotamos por $Vars(t)$ el conjunto de variables que aparecen en el término t, entonces, el conjunto $Libres(F)$ de las *variables que tienen alguna aparición libre en la fórmula F* se define:

- $Libres(p) = \emptyset$ si p es un símbolo de predicado de aridad 0.
- $Libres(p(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} Vars(t_i)$
- $Libres(\neg F) = Libres(F)$
- $Libres(F \vee G) = Libres(F \wedge G) = Libres(F) \cup Libres(G)$
- $Libres(\forall x\, F) = Libres(\exists x\, F) = Libres(F) - \{x\}$
  (aquí el '$-$' denota la resta de conjuntos).

Fórmulas cerradas: Una fórmula F es *cerrada* si $Libres(F) = \emptyset$.

Evaluación de fórmulas cerradas: No resulta difícil de ver que si F es una fórmula cerrada, el resultado de $eval_I^{\alpha}(F)$ no depende de α. Por ello, si F es una fórmula cerrada, escribiremos simplemente $eval_I(F)$, en vez de $eval_I^{\alpha}(F)$.

Satisfacción de fórmulas cerradas: Una interpretación I *satisface* una fórmula cerrada F, denotado $I \models F$, si $eval_I(F) = 1$. También se dice que F es cierta en I o que I es modelo de F.

2. Explicaciones sobre la definición de la Lógica de primer orden

Salvo que se diga lo contrario, trabajaremos siempre con fórmulas cerradas. Para favorecer la escritura de fórmulas claras, también evitaremos aquellas que cuantifican una variable que ya está ligada, como pasa, por ejemplo, en $\forall x\, (p(x) \wedge \exists x\, q(x))$, que es equivalente a $\forall x\, (p(x) \wedge \exists y\, q(y))$. Al igual que para la Lógica proposicional, omitiremos los paréntesis en las fórmulas cuando no dé lugar a ambigüedades, y se considera que los cuantificadores son más prioritarios que las otras conectivas, cuyas prioridades son las de la lógica proposicional.

La Lógica de primer orden tiene un mayor poder expresivo que la proposicional. No sólo podemos modelar "proposiciones" (propiedades que son ciertas o falsas sin matices). También podemos hablar de propiedades que son ciertas para algunos individuos (o combinaciones de individuos), y no para otros. Por ejemplo, podemos expresar cosas como "para todo x existe un y tal que x es menor que y" de la forma $\forall x \exists y\, menor(x,y)$.

Como en la Lógica proposicional, una interpretación nos da una manera de dar un significado a cada símbolo: nos dice en qué dominio pueden tomar valores las variables

(cuantificadas universalmente o existencialmente) y para cada símbolo de función y predicado, qué función lo interpreta.

Todas las nociones definidas en general para una lógica, como las de fórmula válida, contradicción, consecuencia lógica y equivalencia lógica se aplican, evidentemente, a la Lógica de primer orden con fórmulas cerradas.

2.1. Ejemplo de satisfacción

En la definición de $I \models F$ aparece algún tecnicismo que no debería despistar al estudiante, para lo que a continuación presentamos un ejemplo de cómo se evalúan fórmulas en una interpretación.

Queremos determinar si $I \models F$, donde F es $\forall x \exists y\ p(x, y)$, e I es la interpretación donde $D_I = \{a, b\}$ y la función booleana p_I está definida como:

$$\begin{aligned} p_I(a, a) &= 0 \\ p_I(a, b) &= 1 \\ p_I(b, a) &= 1 \\ p_I(b, b) &= 0 \end{aligned}$$

Como F es una fórmula cerrada, hemos de ver si $eval_I(F) = 1$ (sin que importe la asignación α de partida que se considere). Por el cuantificador universal, debe cumplirse la fórmula tanto si x es a como si es b. Por cada uno de esos casos, debemos ver si existe un elemento del dominio para la y tal que la fórmula se satisfaga. Lo podemos analizar usando la siguiente tabla de casos:

x	y	$eval_I(p(x, y))$
a	a	$p_I(a, a) = 0$
	b	$p_I(a, b) = 1$
b	a	$p_I(b, a) = 1$
	b	$p_I(b, b) = 0$

Vemos que, por cada uno de los dos posibles valores d_1 del dominio de x, existe al menos un valor d_2 del dominio para la y que hace que $eval_I(p(d_1, d_2))$ sea 1: si d_1 es a, escogemos b como d_2, y si d_1 es b, escogemos a como d_2. Luego tenemos $I \models F$.

El análisis realizado con esta tabla corresponde a la definición formal de la siguiente manera. Puesto que F es una fórmula cerrada, podemos evaluarla en I con cualquier asignación α (no importa cuál). Por la definición de $eval_I^\alpha$ sobre el cuantificador universal, deben cumplirse los dos casos siguientes:

1. $eval_I^{\alpha[x \mapsto a]}(\ \exists y\ p(x, y)\ ) = 1$
2. $eval_I^{\alpha[x \mapsto b]}(\ \exists y\ p(x, y)\ ) = 1$

Hacemos sólo el primer caso (el segundo es análogo). Llamemos α' a la asignación $\alpha[x \mapsto a]$. Hemos de ver si existe algún valor $d \in D_I$ para el que se cumpla
$eval_I^{\alpha'[y \mapsto d]}(\ p(x, y)\ ) = 1$.
Escogemos $d = b$. Denotando $\alpha'[y \mapsto b]$ como α'', tenemos:

$eval_I^{\alpha''}(\ p(x,y)\ ) =$

$p_I(\ eval_I^{\alpha''}(x),\ \ eval_I^{\alpha''}(y)\ ) =$

$p_I(\ \alpha''(x),\ \ \alpha''(y)\ ) = p_I(a,b) = 1.$

3. Lógica de Primer Orden con igualdad (LPOI)

La LPOI es una ampliación de la Lógica de Primer Orden con un símbolo de predicado binario específico para la igualdad. Su significado no va a ser cualquier función que devuelve un resultado en $\{0,1\}$, sino que se interpretará siempre como la relación binaria "ser el mismo elemento del dominio". Formalmente, la sintaxis y la semántica son las siguientes:

1. **Sintaxis:** Es la misma que para la LPO salvo que ahora añadimos un símbolo adicional de predicado *igual* de aridad 2.
2. **Semántica:** Es la misma que para la LPO, salvo que fijamos la interpretación del símbolo *igual*. La función $igual_I : D_I \times D_I \to \{0,1\}$ se define para cada interpretación I de la manera siguiente:
   - $igual_I(a,a) = 1$ para cada $a \in D_I$.
   - $igual_I(a,b) = 0$ para cada par de elementos distintos $a, b \in D_I$.

Notación: Como suele ser habitual, vamos a denotar el símbolo *igual* mediante $=$ con notación infija. Así, el átomo $igual(s,t)$ lo escribiremos con la notación $s = t$. También es habitual usar $s \neq t$ para denotar $\neg igual(s,t)$.

Esta pequeña ampliación en el lenguaje aumenta enormemente el poder expresivo de las fórmulas. Como veremos en los ejercicios, se puede expresar por ejemplo la *unicidad*, como: "hay un único x tal que $p(x)$", o, en general, "el número de elementos del dominio que cumplen la propiedad p es 5" (o cualquier otro número finito).

4. Formalización del lenguaje natural

Por *formalización* de una frase del lenguaje natural entenderemos *formular* esa frase *mediante una fórmula de la lógica de primer orden (con o sin igualdad).*

El lenguaje natural no tiene una semántica formal por lo que existen frases ambiguas, esto es, que pueden ser expresadas con, al menos, dos fórmulas no lógicamente equivalentes entre sí. Por ejemplo, la frase

"María es sabia si y sólo si estudia y no mira la televisión"

puede formalizarse de dos formas no equivalentes:

$$es.sabia(maria) \leftrightarrow (estudia(maria) \wedge \neg television(maria))$$

$$(es.sabia(maria) \leftrightarrow estudia(maria)) \wedge \neg television(maria)$$

Así pues, no podemos más que dar algunos consejos para hacer esta formalización:

1. La formalización depende de cómo se defina la sintaxis: del conjunto de *símbolos de función* $\mathcal{F}$, del conjunto de *símbolos de predicado* $\mathcal{P}$.

   Por ejemplo, *"todos los sabios buscan la felicidad"* puede formalizarse como:

   - $\forall x \ \ (es.sabio(x) \rightarrow busca.felicidad(x))$
     donde $\mathcal{F} = \emptyset$ y $\mathcal{P} = \{es.sabio^1, busca.felicidad^1\}$
   - $\forall x \ \ (es.sabio(x) \rightarrow busca(x, felicidad))$
     donde $\mathcal{F} = \{felicidad^0\}$ y $\mathcal{P} = \{es.sabio^1, busca^2\}$

2. Una operación puede representarse por un predicado de aridad $n + 1$ o por un símbolo de función de aridad n; por ejemplo, en lugar del símbolo de función $suma(x, y)$ se puede considerar el predicado $es.suma.de(x, y, z)$. El uso de símbolos de función en lugar de predicados simplifica, en general, la complejidad de la fórmula.

   Podemos formalizar *"todos los maestros instruyen a sus discípulos"* como:

   $$\forall x \ \ \forall y \ \ (es.maestro.de(x, y) \rightarrow instruye(x, y))$$

   Otra posibilidad es:

   $\forall x \ instruye(maestro(x), x))$, donde $maestro(x)$ denota una función que devuelve el maestro de x.

   Sin embargo hay una diferencia importante entre estas dos maneras dos formalizar. Cuando usamos un símbolo de función asumimos que cada operación tiene un único resultado (esto está implícito en la semántica), mientras que si lo formalizamos con un símbolo de predicado esto no es así (para que las dos formalizaciones resultasen equivalentes habría que añadir una fórmula que expresase esta propiedad). Por ejemplo, cuando hemos usado el símbolo de función $maestro(x)$ hemos supuesto que cada alumno tiene un solo maestro.

3. Como ya hemos visto en ejemplos anteriores, los nombres comunes o propios son símbolos de función constantes; por ejemplo *"Juan no es sabio"* lo podemos formalizar como

   $$\neg es.sabio(juan)$$

   La frase *"Juan no es sabio ni trabajador"* puede formalizarse como

   $$\neg es.sabio(juan) \wedge \neg es.trabajador(juan)$$

4. El *"o"* del lenguaje natural es ambiguo pues a veces deberá ser formalizado con la conectiva $\vee$ (*o inclusivo*) y otras deberá usarse el *o exclusivo* definido por $p \ \ xor \ \ q = (p \wedge \neg q) \vee (\neg p \wedge q)$. Por ejemplo, cuando se dice *"O haces bien tu trabajo o te despido"* se está refiriendo al *xor*.

   También el *si condicional* del lenguaje natural puede ser ambiguo. Por ejemplo, cuando los padres dicen *"si te comes las espinacas, saldrás a jugar"* a menudo quieren decir también que *"si no las comes, no saldrás"*. En realidad, los padres se están refiriendo a una doble implicación.

5. Mediante la fórmula

$$\exists x \ \ (es.autodidacta(x) \wedge es.sabio(x))$$

representamos las frases

"algún autodidacta es sabio"

"no todos los autodidactas son ignorantes"

(donde *ignorante* quiere decir *no sabio*).

6. Mediante la fórmula

$$\exists x \ \ (es.autodidacta(x) \wedge \neg es.sabio(x))$$

representamos las frases

"no todos los autodidactas son sabios"

"hay autodidactas ignorantes"

7. Mediante la fórmula

$$\forall x \ \ (es.sabio(x) \rightarrow \neg es.egoista(x))$$

representamos las frases

"todos los sabios carecen de egoísmo"

"ningún sabio es egoísta"

8. Mediante la fórmula

$$\forall x \ \ (es.sabio(x) \rightarrow es.curioso(x))$$

representamos las frases

"todos los sabios son curiosos"

"nadie es sabio a menos que sea curioso"

"no hay ningún sabio que no sea curioso"

"sólo los curiosos son sabios"

"ser curioso es necesario para ser sabio"

"ser sabio es suficiente para ser curioso"

5. Ejemplos de formalización del lenguaje natural

1. Queremos formalizar las siguientes frases:

   (a) *"Todo profesor está feliz si todos sus estudiantes aman la lógica"*

   (b) *"Todo profesor está feliz si no tiene estudiantes"*

   Si el conjunto de símbolos de predicado es $\mathcal{P} = \{fe^1, al^1, est^2\}$, con el significado:

   $fe(x)$: x está feliz

   $al(x)$: x ama la lógica

$est(x, y)$: x es estudiante de y

la frase (a) se puede formalizar de la forma:

$$\forall x(\ \forall y(\ est(y, x) \rightarrow al(y)\ ) \rightarrow fe(x)\ ) \qquad (F)$$

y la frase (b) se puede formalizar de la forma:

$$\forall x(\ \forall y \neg est(y, x) \rightarrow fe(x)\ ) \qquad (G)$$

2. Queremos formalizar el siguiente texto (extraído de Pelletier F.J. (1986), Seventy-Five Problems for Testing Automatic Theorem Provers):

   Alguien que vive en la mansión Dreadbury asesinó a tía Ágata. Ágata, el mayordomo y Carlos son las únicas personas que viven en la mansión Dreadbury. Todo asesino odia siempre a sus víctimas y nunca es más rico que sus víctimas. Carlos sólo puede odiar a las personas que odia tía Ágata. Ágata odia a todo el mundo excepto al mayordomo. El mayordomo odia a todos los que no son más ricos que tía Ágata. El mayordomo odia a todos los que tía Ágata odia. Nadie odia a todo el mundo. Ágata no es el mayordomo. Luego, Ágata se suicidó.

   El conjunto de símbolos de función es $\mathcal{F} = \{agata^0, mayordomo^0, carlos^0\}$.

   El conjunto de símbolos de predicado es $\mathcal{P} = \{vive^1, as^2, o^2, mr^2, =^2\}$ con el siguiente significado:

   $vive(x)$: x vive en la mansión Dreadbury

   $as(x, y)$: x es el asesino de y

   $o(x, y)$: x odia a y

   $mr(x, y)$: x es más rico que y

   El enunciado se formaliza como $F \models G$ donde $F = \bigwedge_{i=1}^{13} F_i$ y las fórmulas F_i, G se definen de la forma siguiente:

   - *Alguien que vive en la mansión Dreadbury asesinó a tía Ágata.*

     $$\exists x\ (vive(x) \wedge as(x, agata)) \qquad (F_1)$$

   - *Ágata, el mayordomo y Carlos son las únicas personas que viven en la mansión Dreadbury.*

     $$vive(agata) \qquad (F_2)$$
     $$vive(mayordomo) \qquad (F_3)$$
     $$vive(carlos) \qquad (F_4)$$
     $$\forall x\ (vive(x) \rightarrow x = agata \vee x = mayordomo \vee x = carlos) \qquad (F_5)$$

   - *Todo asesino odia siempre a sus víctimas y nunca es más rico que sus víctimas.*

     $$\forall x\ \ (\forall y\ \ (as(x, y) \rightarrow o(x, y))) \qquad (F_6)$$
     $$\forall x\ \ (\forall y\ \ (as(x, y) \rightarrow \neg\ \ mr(x, y))) \qquad (F_7)$$

- *Carlos sólo puede odiar a las personas que odia tía Ágata.*
  $$\forall x \ (o(carlos, x) \to o(agata, x)) \qquad (F_8)$$
- *Tía Ágata odia a todo el mundo excepto al mayordomo.*
  $$\forall x \ (x \neq mayordomo \leftrightarrow o(agata, x)) \qquad (F_9)$$
- *El mayordomo odia a todos los que no son más ricos que tía Ágata.*
  $$\forall x \ (\neg mr(x, agata) \to o(mayordomo, x)) \qquad (F_{10})$$
- *El mayordomo odia a todos los que tía Ágata odia.*
  $$\forall x \ (o(agata, x) \to o(mayordomo, x)) \qquad (F_{11})$$
- *Nadie odia a todo el mundo.*
  $$\forall x \ (\exists y \ (\neg o(x, y))) \qquad (F_{12})$$
- *Tía Ágata no es el mayordomo.*
  $$agata \neq mayordomo \qquad (F_{13})$$
- *Ágata se suicidó.*
  $$as(agata, agata) \qquad (G)$$

6. Ejercicios de la definición de LPO

1. (dificultad 1) Sea $\mathcal{F}$ el conjunto $\{c^0, f^1, g^2\}$. Evalúa los términos $f(f(g(c, f(c))))$ y $f(g(f(f(x)), g(c, f(c))))$ en las tres interpretaciones y asignaciones siguientes:

   a) La interpretación I se define por: $D_I = \mathbb{N}$, $c_I = 1$, $f_I(n) = n + 1$ y $g_I(n, m) = n + m$. La asignación α cumple $\alpha(x) = 7$.

   b) $D_I = P(\mathbb{N})$, c_I es el conjunto de los números pares, $f_I(A) = \mathbb{N} - A$ (el complementario de A) y $g_I(A, B) = A \cap B$. Además, $\alpha(x) = \emptyset$.

   c) El dominio D_I son las cadenas binarias, $c_I = 1$, $f_I(c) = c0$ y g_I es la concatenación. La asignación α cumple $\alpha(x) = 1111$.

2. (dificultad 2) Este ejercicio es una continuación del anterior. Definimos los términos s_n y t_n recursivamente de la manera siguiente $s_0 := c$, $t_0 := c$, $s_{n+1} := f(s_n)$ y $t_{n+1} := f(g(t_n, t_n))$. Determina por inducción su evaluación en función de n en las tres interpretaciones anteriores (en la tercera interpretación evalúa sólo los 4 primeros términos t_n).

3. (dificultad 2) Sea el conjunto de símbolos de función $\mathcal{F} = \{+^2, s^1, z^0\}$. Consideremos también una interpretación I tal que D_I es $\mathbb{N}$ (los naturales), y los símbolos de función $+, s, z$ se interpretan como la suma en los naturales, el sucesor de un número natural y el número natural cero, respectivamente.

   a) Da dos términos distintos y sin variables tales que su evaluación en I sea 2.

   b) Demuestra que para todo número natural n existe un término t sin variables tal que la evaluación de t en I es n.

c) Demuestra que dado un número natural n existen infinitos términos sin variables tales que su evaluación en I es n.

4. (dificultad 1) Dados los símbolos de función $\mathcal{F} = \{ \mathit{cero}^0, \mathit{suc}^1, \mathit{resta}^2\}$ y los símbolos de predicado $\mathcal{P} = \{\mathit{escero}^1, \mathit{espositivo}^1, \mathit{esmenor}^2\}$ define interpretaciones I_1 a I_6 donde:

   - $D_{I_1} = \mathbb{Z}$, conjunto de los números enteros
   - $D_{I_2} = \{a, b\}$
   - $D_{I_3} = \{a\}$
   - $D_{I_4} = \mathcal{P}(\mathbb{N})$, el conjunto de todos los subconjuntos de $\mathbb{N}$
   - $D_{I_5} = \mathbb{N}^*$, el conjunto de todas las cadenas de naturales
   - $D_{I_6} = \mathbb{Z}$ *modulo* 3, el conjunto de las clases de equivalencia de los enteros módulo 3 (x e y están relacionados si x *mod* 3 = y *mod* 3, donde *mod* es el resto de la división entera).

   Para cada una de las interpretaciones anteriores, indica cuáles satisfacen las fórmulas

   - $\forall x\ \mathit{escero}(\mathit{resta}(x, x))$
   - $\forall x\ (\mathit{espositivo}(x) \rightarrow \mathit{esmenor}(\mathit{cero}, \mathit{suc}(x)))$
   - $\forall x\ (\mathit{espositivo}(x) \rightarrow \mathit{esmenor}(x, \mathit{suc}(x)))$
   - $\exists x \forall y\ (\mathit{espositivo}(y) \rightarrow \mathit{esmenor}(x, y))$

5. (dificultad 1) Sea F la fórmula $\exists x \exists y \exists z\ \ (p(x, y) \land p(z, y) \land p(x, z) \land \neg p(z, x))$. Cuáles de las siguientes interpretaciones son modelos de F?

   a) $D_I = \mathbb{N}$ y $p_I(m, n) = 1$ si y sólo si $m \leq n$.

   b) $D_I = \mathbb{N}$ y $p_I(m, n) = 1$ si y sólo si $n = m + 1$.

   c) $D_I = \mathcal{P}(\mathbb{N})$ (esto denota *partes de* $\mathbb{N}$, es decir, el conjunto de todos los subconjuntos de $\mathbb{N}$), y $p_I(A, B) = 1$ si y sólo si $A \subseteq B$.

6. (dificultad 2) Expresa con tres fórmulas las propiedades de reflexividad, simetría y transitividad de un predicado binario p y demuestra que ninguna de las tres fórmulas es consecuencia lógica de (la conjunción de) las otras dos.

7. (dificultad 3) Sea $\mathcal{F}$ el conjunto $\{f^2, c^0\}$ y sea $\mathcal{P}$ el conjunto $\{p^2\}$. Consideremos una interpretación I tal que $D_I = \mathbb{N}$ y $p_I(n, m) = 1$ si y sólo si $n \leq m$. Encuentra tres interpretaciones distintas para f y c que satisfagan la fórmula

   $$\forall x\ (\ p(f(x, c), x)\ \land\ p(x, f(x, c)\ )$$

   y que sólo dos de ellas satisfagan la fórmula

   $$\forall x \forall y\ (\ p(f(x, y), f(y, x)) \land p(f(y, x), f(x, y))\ )$$

8. (dificultad 2) Sea F la fórmula $\forall x \exists y\ p(x,y) \wedge \forall x \exists y\ \neg p(x,y)$. Demuestra que F es satisfactible.

   Si I es una interpretación, decimos que *el número de elementos de I* es $|D_I|$, el número de elementos de D_I. Asimismo, decimos que I es un *modelo finito* cuando D_I es finito, y hablamos de *la cardinalidad de I* para referirnos a la cardinalidad de D_I.

   ¿Cual es el mínimo número de elementos que debe tener un modelo de F?

9. (dificultad 3) Considera los conjuntos de símbolos y pares de interpretaciones I_1 e I_2 siguientes. Para cada caso, da una fórmula F que es cierta en una de ellas y falsa en la otra, y razona informalmente por qué es así.

   a) $\mathcal{P} = \{r^2\}$, I_1 tiene como dominio los naturales y el predicado se interpreta como el orden (es decir $r_I(n,m) = 1$ si y sólo si $n \leq m$); I_2 tiene como dominio los enteros y el predicado también se interpreta también como el orden;

   b) $\mathcal{P} = \{r^2\}$, I_1 tiene como dominio los enteros y el predicado se interpreta como el orden; I_2 tiene como dominio los racionales y el predicado se interpreta también como el orden.

   c) $\mathcal{P} = \{r^2\}$. El dominio tanto de I_1 como de I_2 son los números enteros, para I_1 el predicado r se interpreta como 'tener el mismo resto módulo 2' y para I_2 el predicado r se interpreta como 'tener el mismo resto módulo 3'.

10. (dificultad 2) Supón que en $\mathcal{P}$ sólo hay símbolos de predicado de aridad cero. Entonces, la sintaxis de las fórmulas, ¿en qué se diferencia de la de la lógica proposicional? ¿Y la semántica?

11. (dificultad 2) Sea F una fórmula con una única variable libre x. $\exists x\, F$ es consecuencia lógica de $\forall x\, F$? Demuéstralo aplicando la definición.

12. (dificultad 3) Sea F una fórmula y sean α y β dos asignaciones tales que $\alpha(x) = \beta(x)$ para toda variable libre x de F (en este caso se dice que α y β *coinciden* en las variables libres de F).

    Demuestra que $eval_I^\alpha(F) = eval_I^\beta(F)$. En particular, si F es cerrada el valor de $eval_I^\alpha(F)$ no depende de α y podemos escribir simplemente $eval_I(F)$.
    Ayuda: demuestra primero que $eval_I^\alpha(t) = eval_I^\beta(t)$ para todo término t tal que α y β coinciden en las variables de t.

13. (dificultad 2) Vamos a extender la noción de equivalencia a fórmulas cualesquiera (no necesariamente cerradas). Diremos que dos fórmulas F y G *son equivalentes* (y lo denotaremos por $F \equiv G$) si $eval_I^\alpha(F) = eval_I^\alpha(G)$ para toda interpretación I y toda asignación α. Observa que esta nueva definición extiende la dada para fórmulas cerradas.

    a) Demuestra que la equivalencia de fórmulas en Lógica de primer orden es una relación de equivalencia (es decir, es reflexiva, simétrica y transitiva).

b) Demuestra que si $F \equiv F'$ y $G \equiv G'$ entonces se cumplen:

- $F \wedge G \equiv F' \wedge G'$
- $F \vee G \equiv F' \vee G'$
- $\neg F \equiv \neg F'$
- $\forall x F \equiv \forall x F'$
- $\exists x F \equiv \exists x F'$

14. (dificultad 4) Enuncia y demuestra el lema de sustitución para la lógica de primer orden.
Ayuda: ver el lema del mismo nombre en los ejercicios de lógica proposicional.

15. (dificultad 2) Las equivalencias de fórmulas que aparecen en los ejercicios de lógica proposicional (como por ejemplo, las leyes de De Morgan) son también ciertas para la lógica de primer orden. Demuestra alguna de ellas.

16. (dificultad 2) Demuestra alguna de las siguientes equivalencias:

$$
\begin{array}{lcl}
\neg\forall x F & \equiv & \exists x \neg F \\
\neg\exists x F & \equiv & \forall x \neg F \\
\forall x \forall y F & \equiv & \forall y \forall x F \\
\exists x \exists y F & \equiv & \exists y \exists x F \\
\forall x F \wedge \forall x G & \equiv & \forall x (F \wedge G) \\
\exists x F \vee \exists x G & \equiv & \exists x (F \vee G) \\
\forall x F \to \exists x G & \equiv & \exists x (F \to G) \\
\forall x F \vee G & \equiv & \forall x (F \vee G), \text{ si } x \text{ no es libre en } G \\
\forall x F \wedge G & \equiv & \forall x (F \wedge G), \text{ si } x \text{ no es libre en } G \\
\exists x F \vee G & \equiv & \exists x (F \vee G), \text{ si } x \text{ no es libre en } G \\
\exists x F \wedge G & \equiv & \exists x (F \wedge G), \text{ si } x \text{ no es libre en } G
\end{array}
$$

17. (dificultad 2) Demuestra que las equivalencias siguientes **no** son ciertas en general (es decir, para cualquier par de fórmulas F, G):

    $\forall x F \vee \forall x G \equiv \forall x (F \vee G)$

    $\exists x F \wedge \exists x G \equiv \exists x (F \wedge G)$

    En ambos casos, hay alguna de la dos fórmulas que sea consecuencia lógica de la otra? (no hace falta que demuestres esto último, ya lo haremos cuando tengamos un cálculo deductivo).

18. (dificultad 2) Demuestra que las fórmulas $\forall x \exists y\ F$ y $\exists y \forall x\ F$ no son equivalentes en general. ¿Hay alguna de la dos fórmulas que sea consecuencia lógica de la otra? (no hace falta que demuestres esto último, ya lo haremos cuando tengamos un cálculo deductivo).

19. (dificultad 2) Las fórmulas $\exists x (F \to G)$ y $\exists x F \to \exists x G$, ¿son equivalentes? Responde a las mismas preguntas para las fórmulas $\forall x (F \to G)$ y $\forall x F \to \forall x G$.

20. (dificultad 3) Demuestra que la fórmula:

    $\exists x\,(p(x) \wedge q(x)) \wedge \exists x\,(p(x) \wedge \neg q(x)) \wedge \exists x\,(\neg p(x) \wedge q(x)) \wedge \exists x(\neg p(x) \wedge \neg q(x))$

    es satisfactible y que todo modelo tiene por lo menos 2^2 elementos. Da un modelo con exactamente 2^2 elementos. Escribe una fórmula con una propiedad análoga para 2^3 elementos. Generalízalo a 2^n.

21. (dificultad 3) Da una fórmula F_3 tal que todo modelo de F_3 tenga al menos 3 elementos. Generalízalo a n cualquiera.
    Ayuda: define la propiedad reflexiva de un símbolo de predicado binario p, y además expresa que hay pares de elementos e_i y e_j en el dominio tales que $p_I(e_i, e_j) = 0$.

22. (dificultad 5) Escribe una fórmula F tal que si $I \models F$ entonces D_I tiene infinitos elementos. Ayuda: piensa en la relación 'ser estrictamente menor que' y expresa (entre otras cosas) que 'no hay máximo' tal como ocurre en los naturales.

23. (dificultad 5) Demuestra que si una fórmula tiene modelos con n elementos, también tiene modelos con m elementos para cualquier $m \geq n$ e incluso modelos infinitos. Ayuda. Si I es una interpretación, tomar un elemento cualquiera a del dominio y añadir 'clones' de este elemento de la manera siguiente. Si A es un conjunto (el conjunto de los 'clones' de a) añadimos todos los elementos de A al dominio y extendemos la interpretación de un símbolo de predicado n-ario así: $(a_1, \ldots, a_n)$ evalúa el predicado a cierto si y sólo si evalúa el predicado a cierto en I al reemplazar cada uno de los clones por a. De manera análoga se define la interpretación de un símbolo de función. Llamemos I' a esta nueva interpretación. Si $\alpha\colon \mathcal{X} \to D_{I'}$ es una asignación para I', consideramos la asignación $\alpha'\colon \mathcal{X} \to D_I$ consistente en reemplazar cada 'clon' por a. Ahora demostrad por inducción los hechos siguientes:

    a) para cada término t, si $eval^{\alpha}_{I'}(t)$ es un elemento de D_I entonces $eval^{\alpha}_{I'}(t) = eval^{\alpha'}_{I}(t)$; si $eval^{\alpha}_{I'}(t)$ es un elemento de A entonces $eval^{\alpha'}_{I}(t) = a$.

    b) para cada fórmula F no necesariamente cerrada se cumple que:

    $$eval^{\alpha}_{I'}(F) = eval^{\alpha'}_{I}(F).$$

    Observad que I e I' satisfacen exactamente las mismas fórmulas cerradas.

6.1. Ejercicios de la definición de LPOI

24. (dificultad 2) Escribe una fórmula F que exprese que para todo modelo I de F:

    a) hay como máximo 1 elemento en el dominio de I

    b) hay como máximo 2 elementos en el dominio de I

    c) hay como máximo n elementos en el dominio de I, para una n dada

    d) hay exactamente n elementos en el dominio de I, para una n dada

¿Se podría hacer esto sin utilizar la igualdad? (ver el último ejercicio del apartado previo).

25. (dificultad 1) En este ejercicio $\mathcal{F}$ consta de un único símbolo de función 1-aria f. Utilizando la igualdad, expresa con una fórmula F que f_I es inyectiva (es decir $I \models F$ si y sólo si f_I es inyectiva). Expresa que f_I es exhaustiva y también que f_I es biyectiva.

26. (dificultad 2)

    a) Sea p un símbolo de predicado unario. Escribe una fórmula F que exprese que hay un único elemento que cumple p. Esto quiere decir: que exprese que para todo modelo I de F hay un único elemento a en D_I con $p_I(a) = 1$.

    b) Escribe otra F expresando que hay exactamente 2.

    c) Generalízalo a un número natural n cualquiera.

    d) Generalízalo más, considerando en vez de p una fórmula G con una única variable libre x: expresa que hay n elementos x tales que se cumple G.

27. (dificultad 2) Un *monoide* es un modelo de la siguiente fórmula:

$$\forall x \forall y \forall z \;\; (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad \wedge \quad \forall x \;\; x \cdot e = x \quad \wedge \quad \forall x \;\; e \cdot x = x$$

    donde $\cdot$ es un símbolo de función binaria y e es un símbolo de constante. Observa que hemos usado notación infija (como hacemos con el símbolo = para la igualdad). Con la notación habitual (y con f en vez de $\cdot$) la fórmula $\forall x \forall y \forall z \;\; (x \cdot y) \cdot z = x \cdot (y \cdot z)$ se escribiría $\forall x \forall y \forall z \;\; f(f(x, y), z) = f(x, f(y, z))$.

    Demuestra que las siguientes interpretaciones son monoides:

    a) El dominio es el conjunto de los números racionales, $\cdot$ se interpreta como la suma y e se interpreta como 0.

    b) El dominio es el conjunto de los números naturales, $\cdot$ se interpreta como el producto y y e se interpreta como 1.

    c) El dominio es $\mathcal{P}(\mathbb{N})$ (el conjunto de los subconjuntos de $\mathbb{N}$), $\cdot$ se interpreta como la intersección y e se interpreta como $\mathbb{N}$.

    d) El dominio es el conjunto de las cadenas binaria ('strings' de ceros y unos), $\cdot$ se interpreta como la concatenación y e se interpreta como la cadena vacía.

    e) El dominio es $\{0, 1, \cdots, n-1\}$, $\cdot$ se interpreta como la suma módulo n (es decir $i \cdot_I i = i + j \bmod n$) y e se interpreta como 0.

    f) El dominio es el conjunto $\{\alpha, \beta\}$, $\cdot$ se interpreta mediante $\alpha \cdot_I \alpha = \alpha, \alpha \cdot_I \beta = \beta, \beta \cdot_I \alpha = \beta, \beta \cdot_I \beta = \alpha$, y e se interpreta como α.

28. (dificultad 2) Este ejercicio es una continuación del anterior. Un *grupo* es un monoide que además satisface la fórmula:

$$\forall x \exists y \;\; (\; x \cdot y = e \;\; \wedge \;\; y \cdot x = e \;)$$

¿Cuales de las interpretaciones anteriores eran grupos?

Un monoide se llama conmutativo o *Abeliano* cuando satisface la fórmula:

$$\forall x \forall y \;\; x \cdot y = y \cdot x$$

Cuales de las interpretaciones anteriores eran monoides Abelianos?

29. En este ejercicio $\mathcal{F}$ consta de dos símbolos de función 1-aria f, g.

    a) Utilizando la igualdad, expresa los hechos siguientes:

    1) (dificultad 1) Las funciones f_I y g_I son iguales.
    2) (dificultad 2) La función f_I es constante.
    3) (dificultad 2) La imagen de f_I está contenida en la imagen de g_I.
    4) (dificultad 2) La imagen de f_I y la de g_I son iguales.
    5) (dificultad 2) La imagen de f_I y la de g_I tienen un único elemento en común.
    6) (dificultad 3) La imagen de f_I contiene exactamente la imagen de g_I y 2 elementos más.

    b) Dadas las formulas:

    $F_1 : \quad \forall x \quad f(x) = g(x)$
    $F_2 : \quad \forall x \, \forall y \quad f(x) = g(y)$
    $F_3 : \quad \forall x \, \exists y \quad f(x) = g(y)$
    $F_4 : \quad \exists x \, \forall y \quad f(x) = g(y)$
    $F_5 : \quad \exists x \, \exists y \quad f(x) = g(y)$

    construye un modelo de cada uno de las siguientes fórmulas:

    $F_1 \wedge \neg F_2, \quad F_2, \quad \neg F_1 \wedge F_3, \quad \neg F_1 \wedge F_4, \quad \neg F_3 \wedge \neg F_4 \wedge F_5, \quad \neg F_5.$

30. (dificultad 2) Demuestra, aplicando la definición, que la fórmula $\forall x \;\; x = x$ es válida.

31. (dificultad 2) Razona (puedes aplicar la semántica de manera informal sin necesidad de recurrir a la definición de *eval*) por qué la fórmula $\forall x \forall y \; (x = y \rightarrow f(x) = f(y))$ y la fórmula $\forall x \forall y \; (x = y \rightarrow (p(x) \leftrightarrow p(y)))$ son válidas. Demuestra que si en estas fórmulas reemplazamos la igualdad por un símbolo de predicado q de aridad 2 cualquiera dejan de ser fórmulas válidas. Expresa mediante fórmulas propiedades análogas para símbolos de función y predicado n-arios. Son también fórmulas válidas?

32. Para los conjuntos de símbolos y pares de interpretaciones siguientes, escribe una fórmula que es cierta en una de ellas y falsa en la otra.

    a) (dificultad 2) $\mathcal{F} = \{f^2\}$, I_1 tiene como dominio los naturales y f se interpreta como el producto; I_2 tiene como dominio $P(\mathbb{N})$ y f se interpreta como la intersección.

    b) (dificultad 2) $\mathcal{F} = \{f^1\}$, el dominio de I_1 son los naturales y el de I_2 son los números enteros: En ambos casos el símbolo f se interpreta como la función 'siguiente', es decir $f_I(n) = n + 1$.

c) (dificultad 4) $\mathcal{F} = \{f^2, g^2\}$. El dominio de I_1 son los números reales, f y g se interpretan como la suma y el producto respectivamente. I_2 es análogo salvo que ahora el dominio son los números racionales.
Ayuda: fabrica el dos y expresa que raíz de dos existe.

33. (dificultad 3) Tenemos los símbolos de constante $\{c_n \mid n \in \mathbb{N}\}$, los símbolos de predicado p^1, q^1, d^2, y los símbolos de función su^2, pr^2. Sea I la interpretación que tiene por dominio los números naturales, la constante c_n se interpreta como el número natural n, p se interpreta como 'ser número primo', q se interpreta como 'ser un cuadrado', d se interpreta como el predicado de divisibilidad, *su* y *pr* se interpretan respectivamente como la suma y el producto de naturales. Expresa con una fórmula las siguientes propiedades:

    a) El número 1 no es primo

    b) 2 es el único natural que es primo y par

    c) Todo número natural es suma de 4 cuadrados

    d) Ningún número primo es un cuadrado

    e) Todo cuadrado par es divisible por 4

    f) Todo número par mayor que 2 es suma de dos primos

    g) Hay números impares que no son primos

    h) Si un número impar es producto de otros dos, éstos también deben ser impares

    i) La suma de dos impares es par

    j) Si un número primo divide al producto de otros dos números, debe dividir alguno de ellos

34. (dificultad 4) Demuestra que si una fórmula no contiene $\neg$ ni ningún símbolo de predicado (salvo la igualdad) es cierta en toda interpretación cuyo dominio tiene un solo elemento.

35. (dificultad 5) Demuestra que la fórmula

$$\forall x \forall y\ (f(x) = f(y) \to x = y) \quad \wedge \quad \exists x \forall y\ x \neq f(y)$$

    es satisfactible pero que todos sus modelos son infinitos.

36. (dificultad 5) Sea F la fórmula $\forall x\ f(f(x)) = x$.

    a) Demuestra que $\forall x \forall y\ (f(x) = f(y) \to x = y)$ es consecuencia lógica de F (puedes aplicar la semántica de manera informal, es decir sin necesidad de recurrir a la definición literal de *eval*). Ídem para $\forall x \exists y\ \ x = f(y)$.

    b) Demuestra que $F \wedge \forall x\ f(x) \neq x$ es satisfactible y que todo modelo finito de dicha fórmula tiene un número par de elementos. Para cada $n \geq 1$ construye un modelo de dicha fórmula que tenga exactamente $2n$ elementos. Podría ocurrir lo mismo con una fórmula sin igualdad?

7. Ejercicios de formalización del lenguaje natural

37. (dificultad 2) Formaliza en lógica proposicional y estudia la validez lógica de la siguiente argumentación, esto es, si la conclusión es consecuencia lógica de la conjunción de las premisas:

    "Si Dios no existe y todo está permitido, entonces vamos inexorablemente hacia el caos. Dios no existe. No vamos hacia el caos. Luego, no todo está permitido"

    (Extraído de Hortalá, Leach, Rodríguez "Matemática discreta y lógica matemática, Ed. Complutense 2001)

38. (dificultad 2) Si tenemos los símbolos de predicado binario ami, con, pa (*ami* para 'amigos', *con* para 'conocidos', y *pa* para 'ser padre de') y los símbolos de constante m, e, r (m para Manuel, e para Enrique y r para Ramón) , formaliza la frases siguientes:

    a) Manuel es hijo de Enrique

    b) Manuel tiene amigos

    c) Ramón conoce a todos los amigos de sus hijos

    d) Enrique y Ramón tienen los mismos amigos

    e) Los amigos comunes de Ramón y Enrique son conocidos de Manuel

    f) Todos conocen a sus amigos pero no son amigos de todos sus conocidos

    g) Todos conocen a los amigos de sus hijos

    h) Todos los padres conocen a los padres de los amigos de sus hijos

    i) Hay padres que no conocen a todos los conocidos de sus hijos

    j) Si dos personas tienen los mismos amigos entonces también tienen los mismos conocidos

    k) Hay personas que tienen los mismos conocidos pero no los mismos amigos

1. Formas normales y cláusulas

En lógica de primer orden, las definiciones de literal, CNF, DNF, cláusula, cláusula vacía, cláusula de Horn, etc., son iguales a las de lógica proposicional (véanse las *Notas de Clase para IL 3.*), excepto que, en vez de símbolos de predicado de aridad cero, ahora hay átomos cualesquiera.

Así, recordemos: una *cláusula* es una disyunción de literales, es decir, una fórmula de la forma $l_1 \vee \ldots \vee l_m$ con $m \geq 0$ donde cada l_i es un literal, o, equivalentemente, una disyunción $A_1 \vee \ldots \vee A_p \vee \neg B_1 \vee \ldots \vee \neg B_n$ de p literales positivos y n negativos, con $p + n \geq 0$, donde las A_i y B_j son átomos.

Todas las variables de una cláusula están universalmente cuantificadas. Así, si C es una cláusula cuyas variables son $x_1, \ldots, x_k$, es como si tuviéramos la cláusula $\forall x_1 \ \ldots \ \forall x_k \ C$. Pero normalmente estos cuantificadores no se escriben.

Para lo que viene a continuación, necesitamos extender a lógica de primer orden la noción de *subfórmula* de una fórmula F que vimos en la Hoja 2 en lógica proposicional:

- Si F es un átomo entonces F es su única subfórmula.
- Si F es de la forma $\neg G$ o $\forall x G$ o $\exists x G$, entonces las subfórmulas de F son F y las subfórmulas de G.
- Si F es de la forma $(G \wedge H)$ o de la forma $(G \vee H)$, entonces las subfórmulas de F son F y las subfórmulas de G y las de H.

Si en una fórmula F tenemos una subfórmula $\forall x \, F_1$, que a su vez tiene a $\exists y \, G$ como subfórmula, decimos que *y se encuentra en el ámbito de x en F*, o que *y depende de x*.

1.1. Transformación a forma clausal

Toda fórmula de lógica de primer orden F puede ser transformada en un conjunto (conjunción) de cláusulas S *preservando la satisfactibilidad:* F es satisfactible si y sólo si lo es S. Esta transformación consta de los siguientes pasos, donde el único paso que no preserva la equivalencia lógica es el de la Skolemización:

1. **Movimiento de las negaciones hacia dentro**, aplicando las reglas:

$$\begin{array}{lcl} \neg(F \wedge G) & \Rightarrow & \neg F \vee \neg G \\ \neg(F \vee G) & \Rightarrow & \neg F \wedge \neg G \\ \neg\neg F & \Rightarrow & F \\ \neg\exists x F & \Rightarrow & \forall x \neg F \\ \neg\forall x F & \Rightarrow & \exists x \neg F \end{array}$$

   Después de este paso aplicando exhaustivamente las reglas (es decir, hasta que no se puedan aplicar más), todas las negaciones estarán aplicadas directamente a los átomos.

2. **Eliminación de conflictos de nombre:**

   En una fórmula (cerrada) F la variable x *tiene un conflicto de nombre* si hay al menos dos subfórmulas de F de la forma $Q_1 x G_1$ y $Q_2 x G_2$ y donde Q_1 y Q_2 son cuantificadores ($\forall$ o $\exists$).

   Claramente, toda fórmula F se puede convertir en una fórmula lógicamente equivalente sin conflictos de nombre, introduciendo variables *frescas* (variables que no aparecen en F). Se reemplazan subfórmulas de la forma QxG por $Qx'G'$, donde G' se obtiene sustituyendo en G todas las apariciones de x por la variable fresca x'.

   Por ejemplo, $\forall x\, P(x) \wedge \forall x\, \neg Q(x)$ se puede convertir en la fórmula equivalente: $\forall x\, P(x) \wedge \forall y\, \neg Q(y)$.

   Es necesario comenzar este reemplazamiento por las subfórmulas más internas. Por ejemplo, en $\forall x\ (\ P(x) \wedge \exists x \neg Q(x)\ )$ hay que comenzar por $\exists x \neg Q(x)$.

   Después de estos dos primeros pasos, la fórmula tendrá todas las negaciones aplicadas directamente a los átomos, y no tendrá conflictos de nombre.

3. **[Opcional:] Movimiento de cuantificadores hacia dentro, mientras sea posible**, aplicando las reglas:

$$
\begin{array}{lcll}
\forall x(F \vee G) & \Rightarrow & \forall xF \ \vee G & \text{Si } x \text{ no aparece en } G \\
\forall x(F \wedge G) & \Rightarrow & \forall xF \ \wedge G & \text{Si } x \text{ no aparece en } G \\
\exists x(F \vee G) & \Rightarrow & \exists xF \ \vee G & \text{Si } x \text{ no aparece en } G \\
\exists x(F \wedge G) & \Rightarrow & \exists xF \ \wedge G & \text{Si } x \text{ no aparece en } G
\end{array}
$$

   Nótese que aplicar estas reglas a fórmulas donde x no aparece en G preserva la equivalencia, y que, por conmutatividad de $\vee$ y de $\wedge$, también tenemos reglas como

$$\exists x(F \wedge G) \Rightarrow F \wedge \exists xG \quad \textbf{Si } x \text{ no aparece en } F.$$

   Este paso (llamado *miniscoping* porque reduce el ámbito (*scope*) de los cuantificadores) no es imprescindible, pero a menudo ayuda a producir fórmulas más sencillas en el siguiente paso, la Skolemización.

4. **Eliminación de cuantificadores existenciales o *Skolemización***:

   Un paso de *Skolemización* de F consiste en reemplazar una subfórmula $\exists y\, G$ por otra G' sin la variable y. Concretamente, G' se obtiene a partir de G reemplazando todas las apariciones de y por un término t, tal que:

   - t es una constante fresca c_y, si y no se encuentra en el ámbito de ninguna variable universalmente cuantificada.
   - t es $f_y(x_1, \ldots, x_n)$, donde f_y es un símbolo de función fresco, si $\{x_1, \ldots, x_n\}$ es el conjunto no-vacío de las variables universalmente cuantificadas en cuyo ámbito se encuentra y.

Ejemplo 1: Intuitivamente, la idea es que una fórmula $\forall x \exists y\ p(x, y)$ es satisfactible si y sólo si, $\forall x\ p(x, f_y(x))$ es satisfactible, es decir, si existe una función que interprete el símbolo f_y de manera que "escoge" la y adecuada para cada x.

En cambio, una fórmula $\exists y \forall x\ p(x, y)$ es satisfactible si y sólo si, $\forall x\ p(x, c_y)$ es satisfactible, es decir, si en el dominio existe un único elemento c_{y_I} (la interpretación en I de c_y) que sirve para todas las x.

Ejemplo 2: Podemos decir que todo ser humano tiene madre:

$$\forall y(\ \neg humano(y) \vee \exists x\ esmadre(y, x)\ ).$$

Esta fórmula se tranformaría en:

$$\forall y(\ \neg humano(y) \vee esmadre(y, f_x(y))\ )$$

Aquí, intuitivamente, el símbolo de función f_x denota la función "madre-de".

Después de los tres primeros pasos y de aplicar exhaustivamente (es decir, mientras sea posible) la Skolemización, la fórmula tendrá todas las negaciones aplicadas directamente a los átomos, no tendrá conflictos de nombre, y solamente tendrá cuantificadores universales.

5. **Movimiento de cuantificadores universales hacia fuera**, aplicando las reglas:

   $$(\forall x F) \vee G \quad \Rightarrow \quad \forall x(F \vee G)$$
   $$(\forall x F) \wedge G \quad \Rightarrow \quad \forall x(F \wedge G)$$

   Después de aplicar exhaustivamente estas reglas (que son correctas puesto que en fórmulas sin conflicto de nombre x no aparecerá en G), la fórmula tendrá todas las negaciones aplicadas directamente a los átomos, no tendrá conflictos de nombre, y sólo tendrá cuantificadores universales, que estarán al principio.

6. **Distribución de $\wedge$ sobre $\vee$**, aplicando (exhaustivamente) la regla:

   $$(F \wedge G) \vee H \Rightarrow (F \vee H) \wedge (G \vee H)$$

   Nótese que, teniendo en cuenta la conmutatividad de $\vee$, en realidad también tenemos la regla $F \vee (G \wedge H) \Rightarrow (F \vee G) \wedge (F \vee H)$.

Después de este último paso, tenemos una expresión de la forma:

$$\forall x_1 \ldots \forall x_k \left((l_{11} \vee \ldots \vee l_{1n_1}) \ \wedge \ldots \wedge \ (l_{m1} \vee \ldots \vee l_{mn_m}) \right)$$

donde cada l_{ij} es un literal. Esta expresión puede ser vista como el conjunto (la conjunción) de m cláusulas:

$$\begin{array}{ll} \forall x_1 \ldots \forall x_k & (l_{11} \vee \ldots \vee l_{1n_1}) \\ \quad \ldots & \quad \ldots \\ \forall x_1 \ldots \forall x_k & (l_{m1} \vee \ldots \vee l_{mn_m}) \end{array}$$

donde, puesto que $\forall x F \wedge \forall x G \quad \equiv \quad \forall x(F \wedge G)$, equivalentemente, hemos colocado en cada cláusula los cuantificadores. Pero debido a que se sabe y se asume que en una cláusula todas las variables están universalmente cuantificadas, normalmente los cuantificadores no se escriben. Al igual que en la lógica proposicional, hablaremos de *conjuntos* de cláusulas (y de literales) por la asociatividad, conmutatividad e idempotencia de $\wedge$ (y $\vee$).

1.2. Unificación

Esta subsección trata de la *unificación* de términos. Consideramos sólo términos porque no es relevante la distinción entre si lo que se unifica son términos o átomos.

Sustituciones: Una *sustitución* es un conjunto de pares $\{x_1 = t_1, \ldots, x_n = t_n\}$, donde las x_i son variables distintas y donde las t_i son términos. El conjunto de variables $\{x_1, \ldots, x_n\}$ es el *dominio* de esta sustitución.

A partir de ahora denotaremos los términos por s o por t, las variables por x, las sustituciones por σ (la letra griega sigma), todos ellos posiblemente con subíndices, y el dominio de una sustitución σ se denotará por $Dom(\sigma)$.

Aplicación y composición de sustituciones: Dada una sustitución σ de la forma $\{x_1 = t_1, \ldots, x_n = t_n\}$, y un término t, el término $t\sigma$ es el que se obtiene al reemplazar simultáneamente cada variable x_i en t por el término t_i correspondiente. Similarmente, definimos la aplicación de sustituciones a átomos o a cláusulas.

Si σ y σ' son dos sustituciones $\{x_1 = s_1, \ldots, x_n = s_n\}$ y $\{y_1 = t_1, \ldots, y_m = t_m\}$, la *composición* de σ y σ' es una nueva sustitución $\sigma\sigma'$ que es:

$\{x_1 = s_1\sigma', \ldots, x_n = s_n\sigma'\} \cup \{\, y_i = t_i \mid i \in 1 \ldots m,\ y_i \notin Dom(\sigma) \,\}$.

Nótese que para todo término t, tenemos $t\sigma\sigma' = (t\sigma)\sigma'$, es decir, aplicar la composición es lo mismo que aplicar primero σ y después σ'.

Unificación y unificadores: Dos términos s y t son *unificables* si existe una sustitución σ tal que $s\sigma$ y $t\sigma$ son el mismo término. En este caso σ es un *unificador* de s y t.

El unificador más general: El *unificador más general* σ de s y t, escrito $\sigma = mgu(s, t)$, es un unificador de s y t tal todo unificador σ' de s y t es un caso particular suyo, es decir, existe una sustitución σ'' tal que $\sigma' = \sigma\sigma''$.

Nota: $mgu(s, t)$ es único salvo cambios de nombre de variables equivalentes. Por ejemplo, podemos expresar el *mgu* de $g(f(x), x)$ y $g(y, z)$ como $\{y = f(x), z = x\}$, o también como $\{y = f(z), x = z\}$.

Unificador simultáneo: En vez de trabajar sobre un solo problema de unificación $s = t$ en el que se busca $mgu(s, t)$, partiremos de un problema más general: un conjunto de pares de términos $\{s_1 = t_1, \ldots, s_n = t_n\}$ para el que se busca un *mgu* σ *simultáneo*, es decir, tal que $s_i\sigma$ es $t_i\sigma$ para todo $i \in 1 \ldots n$. Nótese que en un problema de unificación una expresión $s = t$ es equivalente a $t = s$; el orden entre los dos términos no importa.

Variables resueltas y problemas de unificación resueltos: En un problema de unificación de la forma $P \cup \{x = t\}$, decimos que x es una variable *resuelta* si no aparece en P ni en t, es decir, aparece una sola vez en todo el problema.

Un problema de unificación P de la forma $\{x_1 = t_1, \ldots, x_n = t_n\}$, donde todas las x_i son variables resueltas, se dice que está *resuelto*.

No resulta difícil de ver que encontrar el *mgu* de un problema resuelto P es trivial: el *mgu* es el propio P.

Algoritmo de unificación basado en reglas: Considera el siguiente conjunto de reglas de transformación que, dado un problema inicial $P_0 = \{s_1 = t_1, \ldots, s_n = t_n\}$, o

bien lo convierte en un problema resuelto o bien falla indicando que no hay unificador simultáneo:

$$
\begin{array}{ll}
P \cup \{t = t\} & \Rightarrow P \\
P \cup \{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\} & \Rightarrow P \cup \{t_1 = s_1, \ldots, t_n = s_n\} \\
P \cup \{f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)\} & \Rightarrow \text{“fallo”} \\
 & \quad \textbf{Si } f \neq g \\
P \cup \{x = t\} & \Rightarrow P\{x = t\} \cup \{x = t\} \\
 & \quad \textbf{Si } x \in vars(P) \text{ y } x \notin vars(t) \\
 & \quad \text{y } t \text{ no es variable resuelta en} \\
 & \quad P \cup \{x = t\} \\
P \cup \{x = t\} & \Rightarrow \text{“fallo”} \\
 & \quad \textbf{Si } x \in vars(t) \text{ y } x \neq t
\end{array}
$$

Nótese que la segunda y la tercera regla también se aplican al caso donde f y/o g son constantes (es decir, si n y/o m son 0). Por ejemplo, tenemos $P \cup \{a = a\} \Rightarrow P$, y también tenemos $P \cup \{a = c\} \Rightarrow$ “fallo” si a y c son símbolos de función constantes (es decir, de aridad cero) distintas y también $P \cup \{a = f(x)\} \Rightarrow$ “fallo”.

Nótese que en la cuarta regla $P\{x = t\}$ es la aplicación a P de la sustitución $\{x = t\}$.

Este algoritmo funciona, es decir, acaba en “fallo” si no hay unificador, y acaba con el mgu si lo hay. Esto es cierto porque:

1. Cualquier estrategia de aplicación de las reglas termina, es decir, no existe ningún problema finito P_0 con una secuencia infinita de aplicaciones de reglas $P_0 \Rightarrow P_1 \Rightarrow P_2 \Rightarrow \ldots$

2. Un problema al que ya no se le puede aplicar ninguna regla está resuelto o bien es “fallo”.

3. Las reglas transforman cualquier problema de unificación en otro *equivalente*, en el sentido de que tiene el mismo conjunto, posiblemente vacío, de unificadores.

Los apartados 2 y 3 son fáciles de ver. Para demostrar 1, se sabe que las demostraciones de terminación *de cualquier programa, proceso, o máquina* siempre pueden descomponerse en dos pasos:

- la definición de un orden *bien fundado* $\succ$ sobre el conjunto de *estados* del proceso ($\succ$ es bien fundado si no existe ninguna secuencia infinita de la forma $a_0 \succ a_1 \succ a_2 \succ \ldots$).

- la demostración de que, en cada paso del proceso, el estado decrece con respecto a $\succ$.

En nuestro algoritmo, el estado es el problema de unificación simultánea que tenemos en cada momento. Un orden que funciona es un orden lexicográfico sobre pares de naturales: la primera componente es el número de variables no-resueltas y la segunda

es el tamaño (número de símbolos) del problema. Todas las reglas reducen la primera componente o bien reducen la segunda sin cambiar la primera.

Mucho más que un solo algoritmo. Este tipo de reglas resultan muy convenientes porque en realidad no se proporciona un solo algoritmo, sino toda una *familia de algoritmos*. Una *estrategia* utilizada en la aplicación de las reglas define en cada momento qué regla se aplica, y sobre qué pareja $s = t$ del problema, y cada estrategia distinta da lugar a un algoritmo distinto.

Por ejemplo, determinada estrategia, implementada con cuidado, nos dará el conocido algoritmo de Robinson, otra nos dará el de Martelli y Montanari, otras nos darán algoritmos de coste exponencial. Además, los resultados 1-3 nos dan la *corrección de cualquier estrategia* de éstas.

Este tipo de algoritmos es muy importante porque la unificación es la operación fundamental (tan básica como la suma en la aritmética) en la resolución y sus aplicaciones a la demostración automática, la programación lógica o las bases de datos deductivas.

1.3. Resolución y factorización

Haciendo uso de la unificación, podemos definir las reglas de deducción de resolución y factorización. Véanse las *Notas de Clase para IL 3.*, acerca de las nociones de reglas deductivas, y propiedades como la corrección y la completitud. En las reglas siguientes, A y B denotan átomos (literales positivos) y C y D denotan cláusulas (posiblemente vacías). En la resolución, siempre se asume que las dos premisas *no comparten variables;* si no es así, antes de aplicar resolución hay que realizar los renombramientos necesarios:

(i) *resolución:*

$$\frac{A \vee C \qquad \neg B \vee D}{C\sigma \vee D\sigma} \qquad \text{donde } \sigma = mgu(A, B)$$

(ii) *factorización:*

$$\frac{A \vee B \vee C}{A\sigma \vee C\sigma} \qquad \text{donde } \sigma = mgu(A, B)$$

La resolución y la factorización son correctas. La resolución en lógica de primer orden es la extensión de su versión de lógica proposicional para literales con variables.

Intuitivamente, la resolución en lógica de primer orden es correcta, porque si $I \models \forall x_1 \ldots \forall x_n\ (A \vee C)$, en particular tendremos $I \models A\sigma \vee C\sigma$, y similarmente para la otra premisa, $I \models \neg B\sigma \vee D\sigma$. Pero puesto que $A\sigma$ y $B\sigma$ son el mismo átomo, igual que en el caso proposicional tendremos $C\sigma \vee D\sigma$ como consecuencia lógica (esto sólo es una idea intuitiva, no una demostración).

Un razonamiento similar da intuición sobre la corrección de la regla de factorización: si $I \models \forall x_1 \ldots \forall x_n\ (A \vee B \vee C)$, en particular tendremos $I \models A\sigma \vee B\sigma \vee C\sigma$, y como $A\sigma$ y $B\sigma$ son el mismo átomo, por idempotencia del $\vee$ tendremos $I \models A\sigma \vee C\sigma$.

La resolución y la factorización no son completas. Como en el caso proposicional: por ejemplo, tenemos $p \models p \vee q$, pero no podemos obtener $p \vee q$ a partir de p mediante estas reglas deductivas.

La resolución y la factorización sí son refutacionalmente completas: Si S es un conjunto de cláusulas de primer orden sin igualdad que es insatisfactible, entonces la cláusula vacía $\square$ pertenece a $ResFact(S)$, la clausura de S bajo resolución y factorización.

Para la lógica de primer orden con igualdad, la resolución y la factorización son incompletas refutacionalmente. Es necesaria una regla adicional, llamada *paramodulación*, cuya definición está fuera del alcance de estos apuntes.

La resolución y la factorización pueden no terminar: Al contrario que en la lógica proposicional, para un conjunto finito de cláusulas S, aquí la clausura de S bajo resolución y factorización puede ser infinita.

Por ejemplo, si S tiene la cláusula $p(a)$ y la cláusula $\neg p(x) \vee p(f(x))$, entonces $ResFact(S)$ contiene infinitas cláusulas:
$p(a),\ p(f(a)),\ p(f(f(a))),\ p(f(f(f(a)))), \ldots$

Uso de las reglas deductivas: Como en la lógica proposicional, problemas como determinar si una fórmula es tautología, o satisfactible, o la consecuencia o equivalencia lógica entre fórmulas, etc., pueden ser transformados a un problema de satisfactibilidad de un conjunto de cláusulas. Por ejemplo:

$F \models G$	si, y sólo si,
$F \wedge \neg G$ es insatisfactible	si, y sólo si,
S, la forma clausal de $F \wedge \neg G$, es insatisfactible,	si, y sólo si,
$\square \in ResFact(S)$	

donde $ResFact(S)$ es la clausura bajo resolución y factorización de S (como ya hemos dicho, este último "si, y sólo si" sólo se cumple en la LPO sin igualdad).

SAT en lógica de primer orden no es decidible. En las *Notas de Clase para IL 2 y 3* vimos que el problema de SAT para lógica proposicional era *decidible*: existe un programa (un programa de ordenador, escrito en un lenguaje de programación) que toma como entrada una fórmula proposicional arbitraria F, y

- si F **es** satisfactible, siempre acaba con salida "sí".
- si F **no es** satisfactible, siempre acaba con salida "no".

Por ejemplo, un programa para SAT en lógica proposicional puede construir la tabla de verdad y evaluar F en cada interpretación que existe. Otra opción es, puesto que la resolución en lógica proposicional termina, calcular la clausura bajo resolución y ver si está la cláusula vacía o no. Estos métodos pueden ser costosos en tiempo de cómputo, pero, con suficientes recursos de tiempo y memoria, siempre terminan y contestan correctamente. A estos programas se les llama *procedimientos de decisión*.

Al contrario de lo que ocurre en la lógica proposicional, está demostrado que no puede existir ningún procedimiento de decisión para SAT en lógica de primer orden: se dice que *SAT en lógica de primer orden es indecidible*.

SAT en lógica de primer orden es co-semi-decidible: En cambio, sí existe un programa que toma como entrada una fórmula arbitraria F de lógica de primer orden, y:

- si F **es** satisfactible, o bien no acaba, o bien acaba con salida "sí"
- si F **no es** satisfactible, siempre acaba con salida "no"

Por ejemplo, el programa puede pasar F a forma clausal S_0, e ir calculando la clausura bajo resolución y factorización (y, en el caso de la LPOI, bajo paramodulación) de S_0 por niveles: $S_1, S_2, \ldots$ Si F es insatisfactible, la cláusula vacía aparecerá (al cabo de un tiempo finito) en algún S_j y podemos acabar con salida "no". Si la resolución y la factorización acaban sin generar la cláusula vacía, podemos acabar con salida "sí". Si la resolución no acaba, el programa no acaba. Puesto que existe tal programa, se dice que el problema de SAT en lógica de primer orden es *co-semi-decidible*.

Problemas semi-decidibles: También existe un programa que toma como entrada una fórmula arbitraria F de lógica de primer orden, y:

- si F **es** una tautología, siempre acaba con salida "sí"
- si F **no es** una tautología, o bien no acaba, o bien acaba con salida "no"

Un programa así es un *procedimiento de semi-decisión* para el problema de determinar si una fórmula de lógica de primer orden es una tautología. Este programa puede obtenerse así: puesto que el problema es equivalente a la insatisfactibilidad de $\neg F$, puede utilizar resolución como procedimiento de co-semi-decisión para SAT. De la misma manera, el problema de determinar para dos fórmulas dadas F y G, si $F \models G$, también es semi-decidible.

2. Ejercicios

1. (Dificultad 2) Transforma a forma clausal las siguientes fórmulas:

   a) $\forall x\ (\ \forall y\ (p(y) \rightarrow q(x,y)) \rightarrow \exists y\ q(y,x)\ \ )$

   b) $\forall y\ (\ \neg p(y) \rightarrow \forall y \exists x\ q(y,x)\ )$

   c) $\exists x\ \forall y\ (\ \ \forall z\ (p(y,z) \vee x \neq y)\ \ \rightarrow\ \ (\forall z\ q(y,z) \wedge \neg r(x,y))\ \ )$

2. (Dificultad 2) Demuestra que la Skolemización no preserva la equivalencia lógica.
   Ayuda: considera $\forall x \exists y\ p(x,y)$ y su transformación $\forall x\ p(x, f(x))$, y define una I de dos elementos sobre los símbolos f y p.

3. (Dificultad 1) Unifica $p(\ f(x, g(x)),\ h(y),\ v\ )$ con $p(\ y,\ h(v),\ f(g(z), w)\ )$.
   Calcula un unificador más general y otro que no sea el más general (si hay).

4. (Dificultad 1) Sean los términos:

   t_1 : $f(x, h(g(x)), x')$
   t_2 : $f(a, y, y)$
   t_3 : $f(z, h(z), h(b))$

   ¿Cuáles de los pares (t_i, t_j) son unificables? Da un *mgu* si hay.

5. (Dificultad 2) Demuestra por resolución la insatisfactibilidad del conjunto de cláusulas:

   1 : $p(a, z)$
   2 : $\neg p(f(f(a)), a)$
   3 : $\neg p(x, g(y)) \vee p(f(x), y)$

6. (Dificultad 3) Demuestra que es importante que no haya ninguna variable x que aparezca, con ese nombre, en los dos literales que se unifican cuando se hace resolución.
   Ayuda: Da un ejemplo de incompletitud refutacional de la regla de resolución "sin renombramiento previo de variables".

7. (Dificultad 3) (Schöning, Exercise 85) Formaliza los siguientes hechos:

   (a) *"Todo dragón está feliz si todos sus hijos pueden volar"*

   (b) *"Los dragones verdes pueden volar"*

   (c) *"Un dragón es verde si es hijo de al menos un dragón verde"*

   Demuestra por resolución que la conjunción de (a), (b) y (c) implica que:

   (d) *"Todos los dragones verdes son felices"*

8. (Dificultad 2) Demuestra por resolución que es válida la fórmula:

   $\forall x \exists y \ (\ p(f(f(x)), y) \ \wedge \ \forall z \ (\ p(f(x), z) \rightarrow p(x, g(x, z)) \ ) \ ) \ \rightarrow \ \forall x \exists y \ p(x, y)$

9. (Dificultad 3) Demuestra que la resolución sin la factorización no es refutacionalmente completa.
   Ayuda: da un contraejemplo de dos cláusulas con dos literales cada una.

10. (Dificultad 3) Una cláusula es *de base* si no contiene variables. Demuestra que la satisfactibilidad de conjuntos de cláusulas de base sin igualdad es decidible.

11. (Dificultad 2) Considera la regla de *resolución unitaria:*

$$\frac{A \qquad \neg B \vee C}{C\sigma} \qquad \text{donde } \sigma = mgu(A, B)$$

    Esta regla es refutacionalmente completa para cláusulas de Horn sin igualdad. Demuestra que no lo es sin la restricción a cláusulas de Horn.

12. (Dificultad 4) Sea S un conjunto de cláusulas de Horn sin igualdad donde todos los símbolos de función que aparecen son de aridad cero (son símbolos de constante). Observa que puede haber símbolos de predicado de todo tipo.

a) Demuestra que la resolución unitaria termina para S (es decir, la clausura bajo resolución unitaria es finita).

b) Utilizando este hecho, y la completitud refutacional mencionada en el ejercicio previo, demuestra que la satisfactibilidad de conjuntos de Horn S sin símbolos de función de aridad mayor que cero es decidible.

Nota: las cláusulas de este tipo forman el lenguage del *Datalog*, y este resultado de decidibilidad hace que el *Datalog* sirva para bases de datos deductivas.

13. (Dificultad 3) Escribe un conjunto de cláusulas S sin símbolos de función para el que la resolución no termina.

1. Cálculo de respuestas mediante resolución

El modelo de base de datos más utilizado en la actualidad es el *relacional*. Su idea fundamental es expresar los datos mediante *relaciones* o *tuplas*, como en:

$$\begin{array}{l} padre(juan, pedro) \\ \quad \vdots \\ padre(maria, pedro) \\ \\ hermano(pedro, vicente) \\ \quad \vdots \\ hermano(pedro, alberto) \end{array}$$

A diferencia de otros modelos más antiguos como el jerárquico y el de red, el modelo relacional ofrece una gran flexibilidad en las consultas (normalmente con el lenguaje SQL) y en su administración, al hacer abstracción de la forma concreta en que se almacenan los datos (es decir, no es necesario tener en cuenta los detalles de implementación).

1.1. Bases de datos deductivas

Supongamos ahora que queremos tener también la relación *tio*, que exprese quién es tío de quién, por ejemplo $tio(juan, vicente)$. Una opción sería añadir a esta base de datos todas las relaciones de *tio*. Esto requeriría una cantidad cuadrática de espacio, y podría introducir inconsistencias entre relaciones, al ser la relación *tio* una consecuencia lógica de las otras dos. Una solución mejor puede ser el modelo de las bases de datos *deductivas*, que permite añadir *reglas deductivas* como:

$$\forall x \forall y \; (\quad tio(x,y) \quad \textbf{IF} \quad \exists z \; (\; padre(x,z) \wedge hermano(z,y) \;) \quad)$$

que puede escribirse como:

$$\forall x \forall y \; (\quad tio(x,y) \quad \leftarrow \quad \exists z \; (\; padre(x,z) \wedge hermano(z,y) \;) \quad)$$

y como:

$$\forall x \forall y \; (\quad tio(x,y) \quad \vee \quad \neg \exists z \; (\; padre(x,z) \wedge hermano(z,y) \;) \quad)$$

que es lógicamente equivalente a:

$$\forall x \forall y \; (\quad tio(x,y) \quad \vee \quad \forall z \; \neg (\; padre(x,z) \wedge hermano(z,y) \;) \quad)$$

y a:

$$\forall x \forall y \; (\quad tio(x,y) \quad \vee \quad \forall z \; (\; \neg padre(x,z) \vee \neg hermano(z,y) \;) \quad)$$

y a:

$$\forall x \forall y \forall z \; (\quad tio(x,y) \quad \vee \quad (\; \neg padre(x,z) \vee \neg hermano(z,y) \;) \quad)$$

y que, sin los cuantificadores universales, puede escribirse como la cláusula de Horn:

$$tio(x,y) \vee \neg padre(x,z) \vee \neg hermano(z,y)$$

1.2. Programas lógicos

La misma base de datos también puede escribirse como un *programa lógico*. Un programa en el lenguaje de programación *Prolog*, sería por ejemplo:

$$
\begin{array}{l}
padre(juan, pedro).\\
padre(maria, pedro).\\
hermano(pedro, vicente).\\
hermano(pedro, alberto).\\
tio(x, y) \ \texttt{:-} \ padre(x, z), \ hermano(z, y).
\end{array}
$$

A pesar de las diferencias sintácticas, se trata del mismo conjunto de cláusulas que antes. En este ejemplo, hay cuatro cláusulas de un solo literal y una cláusula con tres literales que, desde el punto de vista lógico, es $tio(x, y) \vee \neg padre(x, z) \vee \neg hermano(z, y)$. Todas las cláusulas de un programa Prolog son cláusulas de Horn con exactamente un literal positivo. Son las llamadas *cláusulas de programa* (*program clauses*). Ahora vamos a ver que las consultas a los programas se hacen mediante cláusulas de Horn con sólo literales negativos. Son las llamadas *cláusulas de objetivo* (*goal clauses*).

2. Cálculo de respuestas

Al programa lógico (o la base de datos deductiva) formado por la conjunción de las cinco cláusulas de arriba lo llamaremos F. Supongamos ahora que queremos saber si $F \models \exists u \exists v \ tio(u, v)$, es decir, si en nuestro programa/base de datos existe un sobrino u cuyo tío es v. Por resolución, esto se averigua viendo si $F \ \wedge \ \neg \exists u \exists v \ tio(u, v)$ es insatisfactible, o, equivalentemente si $F \ \wedge \ \forall u \forall v \ \neg tio(u, v)$ es insatisfactible, lo cual en forma clausal (numerada) puede escribirse como:

$$
\begin{array}{ll}
1: & padre(juan, pedro)\\
2: & padre(maria, pedro)\\
3: & hermano(pedro, vicente)\\
4: & hermano(pedro, alberto)\\
5: & tio(x, y) \vee \neg padre(x, z) \vee \neg hermano(z, y)\\
6: & \neg tio(u, v)
\end{array}
$$

donde la cláusula 6 es una cláusula objetivo (proviene de la negación de lo que queremos demostrar). Ahora, por resolución, podemos obtener la siguiente refutación:

$$
\begin{array}{lll}
7: & \neg padre(u, z) \vee \neg hermano(z, v) & (de \ 5 \ y \ 6)\\
8: & \neg hermano(pedro, v) & (de \ 7 \ y \ 1)\\
9: & \square & (de \ 8 \ y \ 3)
\end{array}
$$

Puesto que hemos obtenido la cláusula vacía, sabemos que efectivamente
$F \models \exists u \exists v \ tio(u, v)$.

Pero lo interesante es que en realidad sabemos más: podemos saber quiénes son este sobrino u y su tío v. Si inspeccionamos la refutación, vemos que u se ha *instanciado* con *juan*, y v con *vicente*. Hemos pues *calculado* la *respuesta* $(juan, vicente)$ para la pregunta "$F \models \exists u \exists v \ tio(u, v)$?".

Para ver más cómodamente qué respuesta se ha calculado para u y v, también es posible añadir un literal "fantasma" $resp(u, v)$, es decir, reemplazar la cláusula 6 por $\neg tio(u, v) \vee resp(u, v)$. Entonces la refutación se convierte en:

$$\begin{array}{lll} 7: & \neg padre(u, z) \vee \neg hermano(z, v) \vee resp(u, v) & (de\ 5\ y\ 6) \\ 8: & \neg hermano(pedro, v) \vee resp(juan, v) & (de\ 7\ y\ 1) \\ 9: & resp(juan, vicente) & (de\ 8\ y\ 3) \end{array}$$

y la cláusula vacía se manifiesta en forma de respuesta $resp(juan, vicente)$ a nuestra pregunta.

Podemos calcular tres respuestas más:

$$\begin{array}{lll} 10: & resp(juan, alberto) & (de\ 8\ y\ 4) \\ 11: & \neg hermano(pedro, v) \vee resp(maria, v) & (de\ 7\ y\ 2) \\ 12: & resp(maria, vicente) & (de\ 11\ y\ 3) \\ 13: & resp(maria, alberto) & (de\ 11\ y\ 4) \end{array}$$

2.1. Completitud para el cálculo de respuestas

Existe un teorema de completitud para el cálculo de respuestas (que aquí no demostraremos), y que esencialmente dice lo siguiente:

- Considera un conjunto F de cláusulas de Horn, y un objetivo, una conjunción de átomos de la forma $\exists x_1 \ldots \exists x_n\ A_1 \wedge \ldots \wedge A_m$.
- Sea σ una sustitución *respuesta* (*answer substitution*) de la forma

  $$\{\ x_1 = t_1,\ \ldots,\ x_n = t_n\ \}$$

  que representa una *respuesta correcta* para este programa y objetivo, es decir:

  $$F \models A_1\sigma \wedge \ldots \wedge A_m\sigma$$

- Entonces, por resolución a partir del conjunto de cláusulas

  $$F \quad \wedge \quad (\neg A_1 \vee \ldots \vee \neg A_m \vee resp(x_1, \ldots, x_n))$$

  hay una manera de obtener una cláusula vacía (o *cláusula respuesta*) de la forma $resp(s_1, \ldots, s_n)$, que representa una sustitución σ' de la forma $\{\ x_1 = s_1,\ \ldots,\ x_n = s_n\ \}$ que es al menos tan general como σ, es decir, que $\sigma = \sigma'\sigma''$ para alguna sustitución σ''.

Ejemplo: Si el programa F tiene dos cláusulas $p(a, x)$ y $p(b, b)$, y la pregunta es $\exists u \exists v\ p(u, v)$, una posible respuesta es $\{u = a, v = b\}$, porque $F \models p(a, b)$. Efectivamente, de las cláusulas $\{\ p(a, x),\ p(b, b),\ \neg p(u, v) \vee resp(u, v)\ \}$ obtenemos las cláusulas de respuesta $resp(a, x)$ y $resp(b, b)$. La primera de ellas nos da la respuesta $\{u = a, v = x\}$, que es más general que $\{u = a, v = b\}$.

3. La ejecución de programas Prolog: resolución SLD

Escribamos de nuevo nuestro ejemplo como un programa Prolog, en un fichero llamado `familia.pl`. Las variables de Prolog comienzan con mayúscula o el carácter subrayado "_". Los símbolos de función y de predicado comienzan con minúsculas:

```
padre(juan,pedro).
padre(maria,pedro).
hermano(pedro,vicente).
hermano(pedro,alberto).
tio(X,Y):- padre(X,Z), hermano(Z,Y).
```

Ahora podemos invocar un entorno de Prolog, por ejemplo en Linux con el comando `gprolog` [1]. Nos sale (algo así como) el siguiente mensaje:

```
GNU Prolog 1.2.18
By Daniel Diaz
Copyright (C) 1999-2004 Daniel Diaz
| ?-
```

que es el intérprete de comandos del Prolog. Ahora podemos decirle que compile nuestro programa:

```
| ?- [familia].
compiling familia.pl for byte code...
familia.pl compiled, 5 lines read - 943 bytes written, 39 ms

(4 ms) yes
| ?-
```

El intérprete está ahora preparado para que le hagamos consultas sobre nuestro programa. Las cláusulas de objetivo se escriben ahora sin la negación, y, como las cláusulas de programa, siempre acaban en un punto. Si preguntamos: `| ?- tio(U,V).` Nos contesta:

```
U = juan
V = vicente ?
```

Si queremos que nos dé más respuestas, contestamos con un punto y coma `;`

```
| ?- tio(U,V).

U = juan
V = vicente ? ;

U = juan
V = alberto ? ;
```

[1] GNU Prolog está disponible en `http://www.gprolog.org`

```
U = maria
V = vicente ? ;

U = maria
V = alberto

yes
| ?-
```

3.1. La resolución SLD

La estrategia de resolución de Prolog se llama *resolución SLD*. No vamos a entrar aquí en el significado exacto del acrónimo SLD (*Selection-rule driven Linear resolution for Definite clauses*). Basta con saber lo siguiente.

Dado un objetivo como `?- tio(U,V),` el Prolog busca la primera cláusula por orden de aparición cuyo literal positivo (la *cabeza* de la cláusula) unifique con el objetivo. En este caso es la cláusula `5. tio(X,Y):- padre(X,Z), hermano(Z,Y).` La resolución entre el objetivo y esta cláusula nos da un nuevo objetivo que es `padre(U,Z), hermano(Z,V).` En este nuevo objetivo, procedemos de izquierda a derecha, de la misma manera, unificando con la cláusula `1. padre(juan,pedro).` Pero aquí también unifica con la cláusula `2. padre(maria,pedro)`, por lo que se guarda ésta como una *alternativa para backtracking* para el presente objetivo. Concretamente, se guarda en una *pila de backtracking* que podemos representar como:

```
|                            |                          |
| padre(U,Z), hermano(Z,V). | 2. padre(maria,pedro).  |
_____________________________________________________
        objetivo                alternativa para
                                  backtracking
```

Después de resolver `padre(U,Z), hermano(Z,V)` con la cláusula `1. padre(juan,pedro)`, el nuevo objetivo es `hermano(pedro,V).` Éste se resuelve con la tercera cláusula `hermano(pedro,vicente)`, y se escribe la respuesta

```
U = juan
V = vicente ?
```

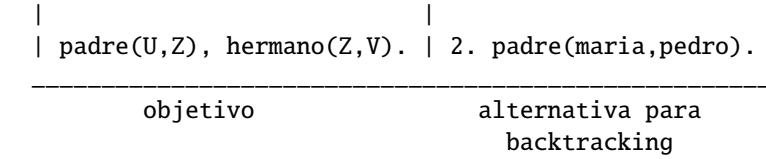
Además, puesto que también era aplicable `4. hermano(pedro,alberto)`, ésta también se empila como alternativa para backtracking y la pila queda:

```
|                            |                              |
| hermano(pedro,V).          | 4. hermano(pedro,alberto).  |
| padre(U,Z), hermano(Z,V). | 2. padre(maria,pedro).      |
_____________________________________________________________
        objetivo                alternativa para
                                  backtracking
```

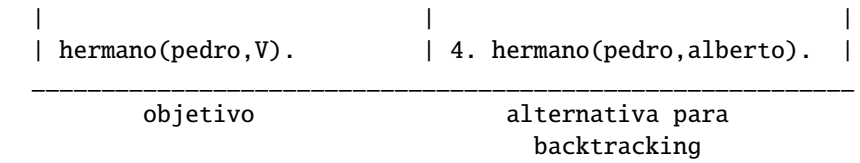
Si ahora pedimos más respuestas (con el punto y coma), el intérprete desempila el estado que hay en la cima de la pila de backtracking: hay que re-hacer el objetivo `hermano(pedro,V)`, esta vez con la cláusula `4. hermano(pedro,alberto).` Esto le permite inmediatamente escribir la respuesta:

```
U = juan
V = alberto ?
```

Puesto que, en nuestro programa, debajo de la cuarta cláusula no hay más alternativas para este objetivo, en este momento no se empila nada, y la pila queda otra vez así:



Si ahora pedimos de nuevo más respuestas, se desempila y se hace resolución sobre el objetivo `padre(U,Z), hermano(Z,V)` con la cláusula `2. padre(maria,pedro).` Después de esto la pila queda vacía. El nuevo objetivo es `hermano(pedro,V)`, y de nuevo podemos unificar con la tercera cláusula, guardando la cuarta como alternativa en la pila:



En este momento se escribe la respuesta:

```
U = maria
V = vicente ?
```

Si pedimos más respuestas, se desempila y se rehace el objetivo `hermano(pedro,V)`, esta vez con la cláusula `4. hermano(pedro,alberto)` y se escribe la respuesta:

```
U = maria
V = alberto ?
```

Después de esto la pila queda vacía: no existen más respuestas.

3.2. Ejemplos de Prolog. Unificación, listas.

Podemos hacer consultas al intérprete de Prolog sobre unificación, usando el predicado "=" y su negación "\=" (no confundir con la igualdad de la LPOI):

```
| ?- f(X,a)=f(b,Y).

X = b
```

```
Y = a
yes

| ?- f(f(X),a)=f(Y,Y).
no

| ?- f(f(X),a)\=f(Y,Y).
yes

| ?- f(f(X),a)=f(Y,Z).

Y = f(X)
Z = a
yes
```

Y también podemos utilizar la notación para *listas*, en la que el operador "|" separa los primeros elementos de la lista de la lista con los demás elementos, y `[]` denota la lista vacía:

```
| ?- [X,Y|L]=[a,b,c].

L = [c]
X = a
Y = b
yes

| ?- [X|L]=[Y,Z,a,g(b),c].

L = [Z,a,g(b),c]
Y = X
yes
```

Podemos ahora escribir un programa `listas.pl` que define la pertenencia a una lista, y también la concatenación de listas:

```
pert(X,[X|_]).             % o bien X es el primero, o bien
pert(X,[_|L]):- pert(X,L). % pertenece a la lista de los demás

concat([],L,L).
concat([X|L1],L2,[X|L3]):- concat(L1,L2,L3).
```

Aquí el subrayado denota una variable cuyo nombre no nos importa y los `%` indican que el resto de la línea es un comentario. Nótese que programar en Prolog consiste en *declarar* o definir las cosas, en vez de *mandar* con instrucciones. Por eso se habla de *programación declarativa*, en contraposición con los lenguajes *imperativos* tradicionales. Ahora podemos hacer preguntas al programa sobre listas:

```
| ?- [listas].
```

```
compiling listas.pl for byte code...
listas.pl compiled, 4 lines read - 1570 bytes written, 32 ms

yes
| ?- pert(X,[a,b,c]).

X = a ? ;
X = b ? ;
X = c ? ;
no

| ?- concat([a,b,c],[c,d,e],L).

L = [a,b,c,c,d,e]
yes

| ?- concat(L1,L2,[a,b,c]).

L1 = []
L2 = [a,b,c] ? ;

L1 = [a]
L2 = [b,c] ? ;

L1 = [a,b]
L2 = [c] ? ;

L1 = [a,b,c]
L2 = [] ? ;
no

| ?- concat([X|L],[a,Y,b,c],[1,2,a,Z,b,Z]).

L = [2]
X = 1
Y = c
Z = c ? ;
no
```

Nótese que el predicado `pert` también podría haber sido definido alternativamente con `pert(X,L):- concat(_,[X|_],L).`

3.3. La programación recursiva y la inducción

Tanto en matemáticas como en programación, a menudo se trabaja con *definiciones recursivas*: se define un concepto en términos de un caso más sencillo de ese mismo

concepto.

Por ejemplo, sabemos que el *factorial* $n!$ de un número n es $n \cdot (n-1) \cdots 1$. Pero a menudo no es posible hacer este tipo de definiciones con puntos suspensivos, o no resultan suficientemente precisas (por ejemplo, 0! qué es?). Una definición recursiva del factorial dice, *como caso base*, que $fact(0) = 1$, y, *como caso recursivo*, que si $n > 0$ entonces $fact(n) = n \cdot fact(n-1)$.

Es obvio que las definiciones recursivas *están estrechamente relacionadas con las demostraciones por inducción*. Por ejemplo, podemos demostrar por inducción que para todo número natural k el factorial de k está bien definido con nuestra definición recursiva: para el caso $k = 0$ lo está, y para el caso $k > 0$ lo está, asumiendo que es así para $k - 1$.

En informática, las definiciones recursivas resultan extremadamente útiles, porque se pueden convertir de manera fácil en programas. Por ejemplo, en el lenguaje `C` o en `C++`, podemos definir la función

```
unsigned int factorial( unsigned int n ) {
  if (n==0) return(1);
  else return(n * factorial(n-1));
}
```

En Prolog pasa lo mismo. Hemos visto la definición recursiva de pertenencia a listas:

```
pert(X,[X|_]).              % caso base: X es el primero,
pert(X,[_|L]):- pert(X,L).  % caso recursivo: lista mas corta
```

Para construir un programa recursivo como éste, y al mismo tiempo convencernos de que es correcto, es necesario pensar por inducción. En este caso, la primer cláusula es el caso base: `X` pertenece a cualquier lista de la cual es el primer elemento. El caso inductivo, la segunda cláusula, dice que `X` también pertenece a una lista si pertenece a la lista obtenida al quitar el primer elemento.

Otra forma de convencernos de que un programa recursivo es correcto es intentando ejecutarlo "a mano" y ver que el comportamiento es el esperado. Pero eso resulta muy engorroso, es fácil equivocarse, y tampoco es posible hacerlo para entradas no-triviales. Por eso es muy preferible limitarse a pensar por inducción, directamente al escribir el programa[2]. Hagamos otro ejemplo, la definición recursiva que vimos de la concatenación de listas:

```
concat([],L,L).                              % caso base: []
concat([X|L1],L2,[X|L3]):- concat(L1,L2,L3). % al menos un elem
```

Este programa es correcto por inducción sobre la longitud k de la primera lista. Caso base: si $k = 0$ (lista vacía), es correcto por la primera cláusula; paso de inducción ($k > 0$): si hay un primer elemento `X` (lista no-vacía), el primer elemento de la concatenación debe ser `X`, y el resto de los elementos (la lista `L3`) se construye con la llamada recursiva `concat(L1,L2,L3)` que es correcta por hipótesis de inducción ya que la longitud de `L1` es $k - 1$.

[2]Como decía el ilustre investigador pionero en informática, E. Dijkstra, "la programación y la verificación han de ir de la mano".

3.4. Los aspectos extra-lógicos de Prolog

Hasta ahora los programas Prolog que hemos visto son puramente lógicos: su ejecución consiste puramente en la estrategia SLD de resolución. Por eso son capaces de contestar a preguntas formuladas de todas las maneras posibles. Ya lo hemos visto en el `concat`, y lo mismo pasa con el de `familia.pl`: además de calcular todas las parejas sobrino-tío, podemos preguntar cosas como

```
| ?- tio(S,alberto).
S = juan ? ;
S = maria
yes

| ?- tio(juan,T).
T = vicente ? ;
T = alberto
yes

| ?- tio(juan,alberto).
yes
```

Pero en la práctica, por razones de eficiencia (entre otras), no basta con la resolución como único mecanismo de cómputo.

Aritmética predefinida: Por ejemplo, es necesario disponer de aritmética predefinida. En Prolog, si un término está formado por operadores aritméticos y constantes (enteras o reales), entonces se puede *evaluar* mediante el predicado `is`. Si no es evaluable, se produce un error de ejecución. Entre muchos otros, existen los operadores `+, -, *, /, mod, //` (este último es la división entera) y los predicados `is, <, >, =<, >=`. Nótese la diferencia entre la unificación con "=" y el `is`:

```
| ?- X = 2+3.
X = 2+3
yes

| ?- X is 2+3.
X = 5
yes

| ?- X is 1/3.
X = 0.33333333333333331
yes

| ?- X is (7+3) mod 2.
X = 0
yes

| ?- 3 is 2+3.
```

```
no

| ?- 2 is X+1.
uncaught exception: error(instantiation_error,(is)/2)

| ?- X is f(1).
uncaught exception: error(type_error(evaluable,f/1),(is)/2)
```

Un ejemplo de programa que utiliza la aritmética es, por ejemplo, el siguiente predicado `fact(N,F)` que significa que el factorial de `N` es `F`:

```
fact(0,1).
fact(X,F):-  X1 is X - 1, fact(X1,F1), F is X * F1.
```

Nótese que este programa sólo puede calcular `F` a partir de `N`, pero no al revés: no puede calcular `N` a partir de `F`. Además, este programa bajo backtracking genera llamadas `fact(-1,F)`, `fact(-2,F)`, etc., por lo que se produce un error de ejecución si se le piden más resultados:

```
| ?- fact(6,F).
F = 720 ?
yes

| ?- fact(N,720).
uncaught exception: error(instantiation_error,(is)/2)

| ?- fact(6,F).
F = 720 ? ;
Fatal Error: local stack overflow
```

Entrada/salida: Otro aspecto extra-lógico es el de la entrada/salida. Por ejemplo, el predicado unario `write` desde el punto de vista lógico siempre se cumple, pero tiene el efecto secundario extra-lógico de escribir algo. Eso mismo pasa con el `nl` (salto de línea) o el `read`, etc, (también hay muchos otros predicados, gestión de ficheros, etc.). Podemos listar directamente todas las respuestas a una pregunta (sin usar el punto y coma) si usamos el `write` y forzamos el backtracking con el `fail`, un predicado que siempre falla (como `1 is 2` o algo similar):

```
| ?- tio(U,V), write([U,V]), nl, fail.
[juan,vicente]
[juan,alberto]
[maria,vicente]
[maria,alberto]
no
```

Consideremos ahora el generador de números naturales:

```
nat(0).
nat(N):- nat(N1), N is N1 + 1.
```

Este programa escribe "todos" (pero no acaba) los números naturales con la pregunta:

```
| ?- nat(N), write(N), nl, fail.
0
1
2
3
...
```

En base a él, podemos definir el mínimo común múltiplo (mcm) de dos números:

```
mcm(X,Y,M):- nat(M), M>0, 0 is M mod X, 0 is M mod Y.
```

u otra versión un poco más eficiente:

```
mcm(X,Y,M):- nat(N), N>0, M is N * X,   0 is M mod Y.
```

El operador de corte "!": Para obtener una versión del programa del factorial que no dé error de ejecución al pedirse más respuestas, podemos usar el *operador de corte*, que se denota por "`!`". En el siguiente programa:

```
fact(0,1):-!.
fact(X,F):-  X1 is X - 1, fact(X1,F1), F is X * F1.
```

desde el punto de vista lógico un corte "`!`" (como el que hay en la primera cláusula) siempre se satisface. Pero aquí además indica que, si se consigue unificar con `fact(0,1)`, entonces no hay que considerar la siguiente cláusula como alternativa: no hay que empilarla en la pila de backtracking como opción alternativa. Si ahora repetimos el experimento con esta nueva versión del `fact`, ya no nos da opción a pedir más respuestas:

```
| ?- fact(6,F).
F = 720
yes
| ?-
```

En general, el comportamiento del "`!`" se define como sigue. Sea un programa de la forma

```
p(...):- ...
...
p(...):- q1(...), ..., qn(...), !, r(...), ...
...
p(...):- ...
```

y considera un objetivo en curso de la forma `p(...), A1, ... Ak`. Asume además que la pila de backtracking está en estado E antes de comenzar a tratarse la llamada `p(...)` del objetivo en curso. Si la ejecución llega al "`!`", se elimina de la pila todas las alternativas para tratar `p(...)` y para tratar los objetivos `q1(...), ..., qn(...)`,

es decir, la pila vuelve a estar en estado E y se continúa con el objetivo `r(...), A1, ... Ak`.

Más ejemplos: Considera las siguientes definiciones:

```
pert_con_resto(X,L,Resto):- concat(L1,[X|L2], L    ),
                            concat(L1,   L2,  Resto).

long([],0).   % longitud de una lista
long([_|L],M):- long(L,N), M is N+1.

factores_primos(1,[]) :- !.
factores_primos(N,[F|L]):- nat(F), F>1, 0 is N mod F,
                N1 is N // F, factores_primos(N1,L),!.

permutacion([],[]).
permutacion(L,[X|P]) :- pert_con_resto(X,L,R), permutacion(R,P).

subcjto([],[]).  %subcjto(L,S) es: "S es un subconjunto de L".
subcjto([X|C],[X|S]):-subcjto(C,S).
subcjto([_|C],S):-subcjto(C,S).
```

Permiten hacer consultas como:

```
| ?- pert_con_resto(X,[a,b,c],R), write(X), write(R), nl, fail.
a[b,c]
b[a,c]
c[a,b]
no

| ?- long([a,b,c,d],N).
N = 4
yes

| ?- factores_primos(120,L).
L = [2,2,2,3,5]
yes

| ?- permutacion([a,b,c],P), write(P), write('  '), fail.
[a,b,c]  [a,c,b]  [b,a,c]  [b,c,a]  [c,a,b]  [c,b,a]
no

| ?- subcjto([a,b,c],S), write(S), write(' '), fail.
[a,b,c] [a,b] [a,c] [a] [b,c] [b] [c] []
no
```

En base a estas definiciones, podemos hacer por ejemplo el juego llamado *cifras*: dada una lista L de enteros y un entero N, encontrar una manera de obtener N a base de su-

mar, restar y multiplicar elementos de L (usando cada uno tantas veces como aparezca en L):

```
cifras(L,N):- subcjto(L,S), permutacion(S,P), expresion(P,E),
              N is E, write(E),nl,fail.

expresion([X],X).
expresion(L,E1+E2):- concat(L1,L2,L),  L1\=[],L2\=[],
                     expresion(L1,E1), expresion(L2,E2).
expresion(L,E1-E2):- concat(L1,L2,L),  L1\=[],L2\=[],
                     expresion(L1,E1), expresion(L2,E2).
expresion(L,E1*E2):- concat(L1,L2,L),  L1\=[],L2\=[],
                     expresion(L1,E1), expresion(L2,E2).
```

Este programa simplemente genera y comprueba todas las expresiones formadas a partir de subconjuntos permutados de L. Escribe miles de soluciones en pocos segundos:

```
| ?- cifras( [4,9,8,7,100,4], 380 ).
4 * (100 - 7) + 8
((100 - 9) + 4) * 4
4 * (9 + 100) - 7 * 8
7 * (8 * (9 - 4)) + 100,
...
```

Otro ejemplo, cuya programación en un lenguaje imperativo como `C`, `C++`, o `Java` ocuparía muchas líneas y costaría muchas horas, es el del cálculo de derivadas, donde `der(E,X,D)` significa que la derivada de la expresión `E` con respecto de `X` es `D`:

```
der(X,X,1):-!.
der(C,_,0):-               number(C).
der(A+B,X,DA+DB):-         der(A,X,DA),der(B,X,DB).
der(A-B,X,DA-DB):-         der(A,X,DA),der(B,X,DB).
der(A*B,X,A*DB+B*DA):-     der(A,X,DA),der(B,X,DB).
der(sin(A),X,cos(A)*DA):-  der(A,X,DA).
der(cos(A),X,-sin(A)*DA):- der(A,X,DA).
der(e^A,X,DA*e^A):-        der(A,X,DA).
der(ln(A),X,DA*1/A):-      der(A,X,DA).
...
```

Este programa simplemente enuncia las regla de derivación. Podemos usarlo, por ejemplo, así:

```
| ?- der( 2*x*x + 3*x, x, D), simplifica(D,D1).
D  = 2*x*1+x*(2*1+x*0)+(3*1+x*0)
D1 = 4*x + 3
yes
| ?-
```

Todas las definiciones Prolog mencionadas aquí (incluyendo las de `simplifica` de este último ejemplo) están disponibles en la página web de la asignatura.

4. Ejercicios

Notación: "dado N" significa que la variable N estará instanciada inicialmente. "Debe ser capaz de generar todas las respuestas posibles" significa que si hay backtracking debe poder generar la siguiente respuesta, como el `nat(N)` puede generar todos los naturales.

1. (dificultad 1) Demuestra por inducción que son correctos los programas para `pert_con_resto`, `long`, `permutación` y `subcjto`.

2. (dificultad 1) Escribe un predicado Prolog `prod(L,P)` que signifique "`P` es el producto de los elementos de la lista de enteros dada `L`". Debe poder generar la `P` y también comprobar una `P` dada.

3. (dificultad 1) Escribe un predicado Prolog `pescalar(L1,L2,P)` que signifique "`P` es el producto escalar de los dos vectores `L1` y `L2`". Los dos vectores vienen dados por las dos listas de enteros `L1` y `L2`. El predicado debe fallar si los dos vectores tienen una longitud distinta.

4. (dificultad 2) Representando conjuntos con listas sin repeticiones, escribe predicados para las operaciones de intersección y unión de conjuntos dados.

5. (dificultad 2) Usando el `concat`, escribe predicados para el último de una lista dada, y para el inverso de una lista dada.

6. (dificultad 3) Escribe un predicado Prolog `fib(N,F)` que signifique "`F` es el `N`-ésimo número de Fibonacci para la N dada". Estos números se definen como: $fib(1) = 1$, $fib(2) = 1$, y, si $N > 2$, como: $fib(N) = fib(N-1) + fib(N-2)$.

7. (dificultad 3) Escribe un predicado Prolog `dados(P,N,L)` que signifique "la lista `L` expresa una manera de sumar `P` puntos lanzando `N` dados". Por ejemplo: si `P` es 5, y `N` es 2, una solución sería `[1,4]`. (Nótese que la longitud de `L` es `N`). Tanto `P` como `N` vienen instanciados. El predicado debe ser capaz de generar todas las soluciones posibles.

8. (dificultad 2) Escribe un predicado `suma_demas(L)` que, dada una lista de enteros `L`, se satisface si existe algún elemento en `L` que es igual a la suma de los demás elementos de `L`, y falla en caso contrario.

9. (dificultad 2) Escribe un predicado `suma_ants(L)` que, dada una lista de enteros `L`, se satisface si existe algún elemento en `L` que es igual a la suma de los elementos anteriores a él en `L`, y falla en caso contrario.

10. (dificultad 2) Escribe un predicado `card(L)` que, dada una lista de enteros `L`, escriba la lista que, para cada elemento de `L`, dice cuántas veces aparece este elemento en `L`. Por ejemplo, `card( [1,2,1,5,1,3,3,7] )` escribirá

    `[[1,3],[2,1],[5,1],[3,2],[7,1]]`.

11. (dificultad 2) Escribe un predicado Prolog `está_ordenada(L)` que signifique "la lista `L` de números enteros está ordenada de menor a mayor". Por ejemplo, con `?-está_ordenada([3,45,67,83]).` dice `yes`

    Con `?-está_ordenada([3,67,45]).` dice `no`.

12. (dificultad 2) Escribe un predicado Prolog `ordenación(L1,L2)` que signifique "`L2` es la lista de enteros `L1` ordenada de menor a mayor". Por ejemplo: si `L1` es `[8,4,5,3,3,2]`, `L2` será `[2,3,3,4,5,8]`. Hazlo en una línea, utilizando sólo los predicados `permutación` y `está_ordenada`.

13. (dificultad 2) ¿Qué número de comparaciones puede llegar a hacer en el caso peor el algoritmo de ordenación basado en `permutación` y `está_ordenada`?

14. (dificultad 3) Escribe un predicado Prolog `ordenación(L1,L2)` basado en el método de la inserción, usando un predicado `insercion(X,L1,L2)` que signifique: "`L2` es la lista obtenida al insertar `X` en su sitio en la lista de enteros `L1` que está ordenada de menor a mayor".

15. (dificultad 2) ¿Qué número de comparaciones puede llegar a hacer en el caso peor el algoritmo de ordenación basado en la inserción?

16. (dificultad 3) Escribe un predicado Prolog `ordenación(L1,L2)` basado en el método de la fusión (*merge sort*): si la lista tiene longitud mayor que 1, con `concat` divide la lista en dos mitades, ordena cada una de ellas (llamada recursiva) y después fusiona las dos partes ordenadas en una sola (como una "cremallera"). Nota: este algoritmo puede llegar a hacer como mucho $n \log n$ comparaciones (donde n es el tamaño de la lista), lo cual es demostrablemente óptimo.

17. (dificultad 4) Escribe un predicado `diccionario(A,N)` que, dado un alfabeto `A` de símbolos y un natural `N`, escriba todas las palabras de `N` símbolos, por orden alfabético (el orden alfabético es según el alfabeto `A` dado). Ejemplo: `diccionario( [ga,chu,le],2)` escribirá:

    ```
    gaga gachu gale chuga chuchu chule lega lechu lele.
    ```

    Ayuda: define un predicado `nmembers(A,N,L)`, que utiliza el `pert` para obtener una lista `L` de `N` símbolos, escríbe los símbolos de `L` todos pegados, y provoca backtracking.

18. (dificultad 3) Escribe un predicado `palíndromos(L)` que, dada una lista de letras `L`, escriba todas las permutaciones de sus elementos que sean palíndromos (capicúas).

    Ejemplo: `palindromos([a,a,c,c])` escribe `[a,c,c,a]` y `[c,a,a,c]`.

19. (dificultad 4) ¿Qué 8 dígitos diferentes tenemos que asignar a las letras `S,E,N,D,M,O,R,Y`, de manera que se cumpla la suma `SEND+MORE=MONEY`? Resuelve el problema en Prolog con un predicado `suma` que sume listas de dígitos. El programa debe decir "no" si no existe solución.

20. (dificultad 4) Escribe el predicado `simplifica` que se ha usado con el programa de calcular derivadas.

21. (dificultad 4) Tres misioneros y tres caníbales desean cruzar un río. Solamente se dispone de una canoa que puede ser utilizada por 1 ó 2 personas: misioneros ó caníbales. Si los misioneros quedan en minoría en cualquier orilla, los caníbales se los comerán. Escribe un programa Prolog que halle la estrategia para que todos lleguen sanos y salvos a la otra orilla.