Lógica Computacional, 2025-2
Unidad 2: Lógica Proposicional

Sintaxis

Manuel Soto Romero Luis Fernando Loyola Cruz

14 de febrero de 2025 ♥
Facultad de Ciencias UNAM

En esta nota exploraremos los fundamentos del lenguaje de la lógica proposicional, comenzando por definir su sintaxis de manera formal y precisa. Estudiaremos el concepto de alfabeto, los símbolos que lo componen y cómo combinarlos para formar expresiones bien formadas o fórmulas. Veremos también las reglas de producción mediante gramáticas, lo que nos permitirá entender cómo se construyen las fórmulas válidas dentro del lenguaje formal.

Posteriormente, profundizaremos en aspectos como la precedencia y asociatividad de los operadores, el uso de árboles de sintaxis para representar fórmulas, y las operaciones sintácticas de inducción y sustitución. Estos conceptos no solo son esenciales para la construcción y análisis de fórmulas lógicas, sino que también sientan las bases para aplicaciones avanzadas en teoría de lenguajes y computación teórica. Al finalizar, estaremos preparados para aplicar estos conocimientos en contextos más complejos y prácticos.

1. Introducción

La lógica proposicional es un sistema lógico fundamental en el estudio de la lógica y sus aplicaciones en la computación. Se basa en el análisis de proposiciones, que se declaran verdaderas o falsas, y su conexión mediante operadores lógicos como la conjunción, disyunción y negación. Este sistema utiliza un lenguaje formal con reglas de sintaxis y semántica que permiten construir proposiciones bien formadas y asignarles valores de verdad. Además, a través de sistemas deductivos y reglas de inferencia como el modus ponens, es posible deducir conclusiones válidas a partir de premisas establecidas. Comenzaremos estudiando la sintaxis, que define las reglas para formar proposiciones válidas mediante símbolos lógicos.

Sus aplicaciones abarcan áreas clave de la computación, como el diseño de circuitos digitales, la verificación de software y el desarrollo de sistemas de inteligencia artificial mediante motores de inferencia y SAT-solvers. Estos últimos son herramientas esenciales en la resolución de problemas de satisfacibilidad lógica y su optimización sigue siendo un tema de investigación. La lógica proposicional también se extiende a nuevos ámbitos, como el razonamiento explicable en inteligencia artificial y la computación cuántica, mostrando su papel tanto en la teoría como en el avance de tecnologías prácticas.

El lenguaje de un sistema lógico debe ser formal, ya que, como hemos estudiado previamente, debe estar libre de ambigüedades. Existen distintas maneras de describir un lenguaje formal; en esta nota lo abordaremos desglosando sus componentes y construyendo una gramática.

2. El Lenguaje de la Lógica Proposicional

Como en todo lenguaje formal, lo primero que debemos definir es el alfabeto, el cual establece los símbolos que componen el lenguaje.

Definición 1. (Alfabeto)

*Un **alfabeto** es un conjunto finito y no vacío de símbolos, denotado comúnmente por Σ. Es la base para la construcción de cadenas y lenguajes formales. Cada símbolo en Σ es un elemento único, y el orden en el conjunto no tiene relevancia.*

El concepto de alfabeto en lógica formal y teoría de lenguajes se refiere al conjunto de símbolos que se utilizan para construir cadenas. Por ejemplo, en un contexto de programación, un alfabeto puede consistir de caracteres como $\{a, b, c, 0, 1\}$, mientras que en matemáticas puede incluir símbolos como $\{+, -, \times, /, =\}$. Es importante que el alfabeto sea finito y no vacío, ya que esto garantiza que se puedan formar cadenas bien definidas y que no haya ambigüedades en su uso. El alfabeto actúa como la base sobre la que se definen lenguajes, al especificar las combinaciones permitidas de sus elementos.

En este caso, distinguimos cuatro clases de símbolos:

- ⋆ Variables proposicionales $p, q, r, s, \ldots$
- ⋆ Constantes lógicas $\bot, \top$
- ⋆ Conectivos lógicos $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$
- ⋆ Símbolos auxiliares $(,)$

Por cierto, estos símbolos tienen nombres específicos. Es probable que ya los conozcas, pero nunca está de más repasarlos:

Operador	Nombre
$\bot$	Constante de falsedad (bottom)
$\top$	Constante de verdad (top)
$\neg$	Negación
$\vee$	Disyunción
$\wedge$	Conjunción
$\rightarrow$	Condicional
$\leftrightarrow$	Bicondicional

Ahora que hemos definido los símbolos de nuestro lenguaje, podemos utilizarlos para combinar y formar expresiones.

Definición 2. (Expresión)

*Una **expresión** es una secuencia finita de símbolos tomada de un alfabeto, organizada de acuerdo a reglas sintácticas específicas. Estas reglas determinan si la secuencia es válida dentro del contexto de un lenguaje formal.*

En términos generales, una expresión es cualquier combinación de símbolos que sigue un conjunto de reglas establecidas por un lenguaje. Por ejemplo, en matemáticas, $a + b$ es una expresión válida porque respeta

las reglas de la aritmética, mientras que + por sí sólo no lo sería. a noción de expresión es fundamental en diversos campos, como la lógica, los lenguajes de programación y las matemáticas, ya que permite representar información, realizar cálculos y expresar relaciones de manera estructurada y comprensible. Una expresión puede ser tan simple como un único símbolo o tan compleja como una fórmula matemática o un programa de computadora.

Podemos formar expresiones simplemente combinando los símbolos de nuestro alfabeto S. Por ejemplo, combinaciones como $pppppp, paqrs, \wedge\neg \rightarrow p$ o $p \vee q$ podrían considerarse expresiones. Sin embargo, como recordarás del curso de Estructuras Discretas, existen reglas específicas que dictan cómo deben combinarse estos símbolos. Solo aceptaremos como válidas las expresiones que cumplan con dichas reglas.

Llamaremos fórmulas o expresiones bien formadas a aquellas expresiones que cumplen con las reglas sintácticas establecidas. El conjunto de todas estas fórmulas constituye lo que se conoce como el Lenguaje de la Lógica Proposicional, que se denota como: LProp.

Definición 3. (Lenguaje)

*Un **lenguaje** es un subconjunto de todas las cadenas posibles que se pueden formar con los símbolos de un alfabeto finito Σ. Formalmente, un lenguaje L sobre un alfabeto Σ es un conjunto de cadenas, es decir $L \subseteq \Sigma^\star$, donde $\Sigma^\star$ representa el conjunto de todas las cadenas finitas, incluyendo la cadena vacía, que se pueden construir con los símbolos de Σ.*

Un lenguaje es un conjunto de cadenas que se forman a partir de un alfabeto siguiendo ciertas reglas. Estas cadenas pueden representar elementos válidos dentro de un sistema, como instrucciones en un programa o fórmulas en lógica. En lógica proposicional, el lenguaje incluye todas las expresiones bien formadas que cumplen las reglas sintácticas definidas, como $p \wedge q$, pero no cadenas como $\wedge \vee p$, que no tienen sentido lógico. La definición formal de lenguaje proporciona una base para estudiar su estructura y propiedades. Describiremos cuáles expresiones de $S^\star$ pertenecen al conjunto LProp.

3. Fórmulas

Podríamos describir las fórmulas de nuestro lenguaje en español. Sin embargo, el español es un lenguaje natural y, como tal, propenso a ambigüedades, lo cual resulta indeseable al describir un lenguaje formal. Por ello, utilizaremos un formalismo probablemente familiar para ti: las *gramáticas*. Estas emplean reglas de producción para definir cómo se pueden combinar los símbolos del lenguaje y así generar expresiones bien formadas.

A continuación, se presenta la gramática del lenguaje LProp junto con una descripción en español. Para ello, utilizaremos la notación EBNF (*Extended Backus-Naur Form*), una herramienta estándar para describir gramáticas de manera clara y estructurada.

$$
\begin{array}{rcl}
\langle \text{cte} \rangle & ::= & \bot \mid \top \\
\langle \text{var} \rangle & ::= & p \mid q \mid \ldots \\
\langle \text{atom} \rangle & ::= & \langle \text{cte} \rangle \\
 & \mid & \langle \text{var} \rangle \\
\langle \text{binop} \rangle & ::= & \vee \mid \wedge \mid \rightarrow \mid \leftrightarrow \\
\langle \text{expr} \rangle & ::= & \langle \text{atom} \rangle \\
 & \mid & (\neg \langle \text{expr} \rangle) \\
 & \mid & (\langle \text{expr} \rangle \langle \text{binop} \rangle \langle \text{expr} \rangle)
\end{array}
$$

Fórmulas Atómicas

Las reglas de la gramática

$$\begin{array}{rcl} \langle \text{atom} \rangle & ::= & \langle \text{cte} \rangle \\ & | & \langle \text{var} \rangle \end{array}$$

definen una categoría sintáctica a la que llamaremos fórmula atómica. Estas fórmulas representan nuestras proposiciones más simples. Por ejemplo, con una variable p podemos expresar la proposición *Hace frío en invierno*. Se les llama atómicas porque no pueden descomponerse en otras proposiciones más básicas.

Las reglas de la gramática

$$\begin{array}{rcl} \langle \text{expr} \rangle & ::= & \langle \text{atom} \rangle \\ & | & (\neg \langle \text{expr} \rangle) \\ & | & (\langle \text{expr} \rangle \langle \text{binop} \rangle \langle \text{expr} \rangle) \end{array}$$

son las más importantes, ya que definen las expresiones que constituyen el lenguaje LProp.

Estas reglas establecen lo siguiente:

1. Las fórmulas atómicas son consideradas fórmulas.
2. Si tenemos dos fórmulas φ y ψ, entonces también son fórmulas las siguientes expresiones:

$$(\neg\varphi)\,, \quad (\varphi \vee \psi)\,, \quad (\varphi \wedge \psi)\,, \quad (\varphi \to \psi)\,, \quad (\varphi \leftrightarrow \psi)\,.$$

3. Solo y únicamente las expresiones construidas siguiendo estas reglas son consideradas fórmulas o expresiones bien formadas.

Ejemplo 1

Veamos la derivación de la siguiente fórmula:

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

$$\begin{aligned} \langle \text{expr} \rangle &\Rightarrow (\langle \text{expr} \rangle \langle \text{binop} \rangle \langle \text{expr} \rangle) \\ &\Rightarrow ((\langle \text{expr} \rangle \langle \text{binop} \rangle \langle \text{expr} \rangle) \langle \text{binop} \rangle \langle \text{expr} \rangle) \\ &\Rightarrow ((\langle \text{atom} \rangle \langle \text{binop} \rangle \langle \text{expr} \rangle) \langle \text{binop} \rangle \langle \text{expr} \rangle) \\ &\Rightarrow ((\langle \text{var} \rangle \langle \text{binop} \rangle \langle \text{expr} \rangle) \langle \text{binop} \rangle \langle \text{expr} \rangle) \\ &\Rightarrow ((p \langle \text{binop} \rangle \langle \text{expr} \rangle) \langle \text{binop} \rangle \langle \text{expr} \rangle) \\ &\Rightarrow ((p \to \langle \text{expr} \rangle) \langle \text{binop} \rangle \langle \text{expr} \rangle) \\ &\Rightarrow ((p \to (\neg \langle \text{expr} \rangle)) \langle \text{binop} \rangle \langle \text{expr} \rangle) \\ &\Rightarrow ((p \to (\neg \langle \text{atom} \rangle)) \langle \text{binop} \rangle \langle \text{expr} \rangle) \end{aligned}$$

$$
\begin{aligned}
\langle \text{expr} \rangle &\Rightarrow ((p \to (\neg\langle \text{var} \rangle))\langle \text{binop} \rangle \langle \text{expr} \rangle) \\
&\Rightarrow ((p \to (\neg q))\langle \text{binop} \rangle \langle \text{expr} \rangle) \\
&\Rightarrow ((p \to (\neg q)) \wedge \langle \text{expr} \rangle) \\
&\Rightarrow ((p \to (\neg q)) \wedge \langle \text{atom} \rangle) \\
&\Rightarrow ((p \to (\neg q)) \wedge \langle \text{cte} \rangle) \\
&\Rightarrow ((p \to (\neg q)) \wedge \top)
\end{aligned}
$$

Cuando intentamos derivar una expresión que no pertenece a la gramática de LProp, el proceso de derivación se bloquea o falla porque no es posible aplicar una secuencia válida de reglas de producción para generar la fórmula. Esto ocurre porque la estructura de la expresión viola las restricciones sintácticas impuestas por la gramática, lo que demuestra que dicha expresión no forma parte del lenguaje definido por ella.

Ejercicio 1

Determina cuáles de las siguientes expresiones son **fórmulas bien formadas** del lenguaje LProp y cuáles no. Justifica tu respuesta.

1. $(t \wedge \vee v)$
2. $((r \to t) \vee (t \to r))$

4. Precedencia y Asociatividad

En ocasiones, colocar tantos paréntesis en nuestras fórmulas puede resultar tedioso. Para simplificar, podemos eliminar algunos de ellos siguiendo una convención basada en las reglas de precedencia y asociatividad de los operadores. Estas reglas forman parte de lo que se conoce como la semántica de los lenguajes, tema que exploraremos con mayor profundidad más adelante.

A continuación, se presentan estas reglas:

Precedencia	**Operador**	**Asociatividad**
1	$\neg$	*No aplica, ya que es unario.*
2	$\wedge, \vee$	*Izquierda a derecha.*
3	$\to$	*Derecha a izquierda.*
4	$\leftrightarrow$	*Izquierda a derecha.*

Es decir, primero resolvemos las negaciones, seguidas de las conjunciones y disyunciones, luego los condicionales y, finalmente, los bicondicionales.

Ejemplo 2

Eliminemos los paŕentesis rendundantes de la siguiente fórmula:

$$(((p \to (\neg q)) \land (r \lor s)) \leftrightarrow ((\top \to u) \land ((\neg v) \to w)))$$

Paso 1: Eliminar paréntesis internos redundantes
Los siguientes elementos pueden simplificarse:

- ⋆ $(\neg q)$ se simplifica como $\neg q$.
- ⋆ $(p \to (\neg q))$ conserva sus paréntesis, ya que el operador $\to$ asocia a la derecha.
- ⋆ $(\neg v)$ se simplifica como $\neg v$.

Esto nos da:

$$(((p \to \neg q) \land (r \lor s)) \leftrightarrow ((\top \to u) \land (\neg v \to w)))$$

Paso 2: Aplicar la asociación de los conectivos

- ⋆ El operador $\land$ asocia a la izquierda, por lo que no son necesarios paréntesis adicionales entre $(p \to \neg q) \land (r \lor s)$.
- ⋆ Del lado derecho, $(\top \to u) \land (\neg v \to w)$ tampoco requiere paréntesis adicionales.

Esto nos deja:

$$(p \to \neg q) \land (r \lor s) \leftrightarrow (\top \to u) \land (\neg v \to w)$$

Paso 3: Eliminar los paréntesis externos innecesarios
Los paréntesis externos que envuelven la expresión completa pueden eliminarse, ya que la notación es clara.
Fórmula final simplificada:

$$(p \to \neg q) \land (r \lor s) \leftrightarrow (\top \to u) \land (\neg v \to w)$$

Ejemplo 3

Restauremos los paréntesis de la siguiente fórmula

$$p \land q \lor \neg r \to s \leftrightarrow \top \to u \land v \lor w$$

Paso 1: Restaurar Negaciones

$$p \land q \lor (\neg r) \to s \leftrightarrow \top \to u \land v \lor w$$

Paso 2: Restaurar Conjunciones y Disyunciones

$$((p \land q) \lor (\neg r)) \to s \leftrightarrow \top \to ((u \land v) \lor w)$$

Paso 3: Restaurar Condicionales

$$(((p \land q) \lor (\neg r)) \to s) \leftrightarrow (\top \to ((u \land v) \lor w))$$

Paso 4: Restaurar Bicondicionales

$$((((p \land q) \lor (\neg r)) \to s) \leftrightarrow (\top \to ((u \land v) \lor w)))$$

Ejercicio 2

Coloca los paréntesis a la siguiente fórmula siguiendo las reglas de **precedencia** y **asociatividad** de los operadores:

$$s \wedge t \to u \leftrightarrow \neg u \vee \neg s \wedge t$$

Ejercicio 3

Elimina los paréntesis innecesarios de la siguiente fórmula, siguiendo las reglas de **precedencia** y **asociatividad** de los operadores:

$$\neg\left(\left(\left(r \wedge \left(r \to (\neg s)\right)\right) \wedge t\right) \wedge r\right)$$

5. Árboles de Sintaxis

Otra forma de representar la construcción de una fórmula es mediante su árbol de sintaxis, en el cual el operador principal se encuentra en la raíz del árbol. Estos árboles son binarios, lo que significa que cada nodo operador puede tener, como máximo, dos descendientes. Los descendientes izquierdo y derecho representan los árboles de sintaxis de las subfórmulas correspondientes. En las hojas del árbol se encuentran las variables proposicionales y las constantes lógicas.

Ejemplo 4

Árbol de sintaxis de la fórmula

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

Ejercicio 4

Dibuja el árbol de sintaxis de la siguiente fórmula, siguiendo las reglas de construcción basadas en operadores y su precedencia:

$$(((r \wedge s) \vee t) \to (\neg r))$$

Observación 1

Contar con reglas de precedencia y asociatividad asegura que toda fórmula proposicional tenga uno y sólo un árbol de sintaxis. En otras palabras, cada fórmula tiene una única descomposición sintáctica.

6. Operaciones Sintácticas: Inducción y Recursión

El lenguaje LProp define un conjunto de fórmulas que, aunque infinito, es estructurado. Este conjunto posee, por definición, una estructura inductiva: *las nuevas fórmulas se construyen a partir de otras utilizando los conectivos lógicos.*

En los conjuntos con estructura inductiva, las definiciones y demostraciones se realizan mediante métodos recursivos e inductivos, respectivamente. Esto permite, en pocas líneas, ofrecer una definición aplicable a cualquier fórmula, por compleja que sea, o demostrar que todas y cada una de las fórmulas infinitas satisfacen una propiedad determinada.

El Principio de Inducción Matemática establece que solo se requiere un paso inductivo: el que lleva de un número a su sucesor. Para demostrar una propiedad sobre los números naturales, es suficiente probar que la propiedad se cumple para el primer número y que, dado un número cualquiera, también se cumple para su sucesor.

En el conjunto de fórmulas, existen múltiples opciones para la generación de nuevas fórmulas. Esto requiere un Principio de Inducción diferente, que establezca tanto el caso base como cada una de las alternativas de construcción. Este tipo de inducción se conoce como Inducción Estructural, un concepto estudiado en cursos como Estructuras Discretas.

Definición 4. (Principio de Inducción Estructural para LProp)

*Sea $P(\varphi)$ una propiedad que deseamos demostrar para todas las fórmulas $\varphi \in \text{LProp}$. El **Principio de Inducción Estructural** establece que $P(\varphi)$ se cumple para todas las fórmulas φ si se satisfacen las siguientes condiciones:*

1. ***Caso base:*** *$P(\varphi)$ es verdadera para todas las fórmulas atómicas, es decir, $\varphi \in \{\bot, \top, p, q, \dots\}$.*
2. ***Paso inductivo:*** *Si $P(\varphi)$ es verdadera para $\varphi, \psi \in \text{LProp}$, entonces $P(\varphi)$ es verdadera para cualquier fórmula construida mediante las reglas de la gramática:*

$$(\neg\varphi), \qquad (\varphi \vee \psi), \quad (\varphi \wedge \psi), \qquad (\varphi \to \psi), \quad (\varphi \leftrightarrow \psi).$$

El Principio de Inducción Estructural para LProp permite demostrar propiedades sobre todas las fórmulas del lenguaje de manera sistemática. La idea se basa en que cada fórmula se genera a partir de las reglas en la gramática de LProp. Primero se prueba que la propiedad P se cumple en las fórmulas atómicas (caso base). Luego, se demuestra que si P se cumple para subfórmulas φ y ψ, también se cumple para fórmulas más complejas construidas a partir de estas mediante conectivos lógicos (paso inductivo). Este enfoque garantiza que la propiedad se cumple para cualquier fórmula porque todas las fórmulas de LProp se construyen siguiendo esta estructura inductiva.

Por otro lado, una definición recursiva se utiliza para definir propiedades o funciones en una estructura inductiva mediante un análisis de casos, es decir, considerando las diferentes formas sintácticas que

caracterizan a los elementos de la estructura (reconocimiento de patrones). Este tema se aborda con mayor profundidad en el laboratorio. A partir de este punto, se empleará la sintaxis del lenguaje de programación Haskell para presentar algunas definiciones.

Ejemplo 5

Número de símbolos de una fórmula

$$\begin{aligned}
\text{ns } \bot &= 1 \\
\text{ns } \top &= 1 \\
\text{ns } p &= 1 \\
\text{ns } (\neg\varphi) &= 1 + \text{ns } \varphi \\
\text{ns } (\varphi \wedge \psi) &= 1 + \text{ns } \varphi + \text{ns } \psi \\
\text{ns } (\varphi \vee \psi) &= 1 + \text{ns } \varphi + \text{ns } \psi \\
\text{ns } (\varphi \to \psi) &= 1 + \text{ns } \varphi + \text{ns } \psi \\
\text{ns } (\varphi \leftrightarrow \psi) &= 1 + \text{ns } \varphi + \text{ns } \psi
\end{aligned}$$

Ejemplo 6

Número de conectivos de una fórmula

$$\begin{aligned}
\text{nc } \bot &= 0 \\
\text{nc } \top &= 0 \\
\text{nc } p &= 0 \\
\text{nc } (\neg\varphi) &= 1 + \text{nc } \varphi \\
\text{nc } (\varphi \wedge \psi) &= 1 + \text{nc } \varphi + \text{nc } \psi \\
\text{nc } (\varphi \vee \psi) &= 1 + \text{nc } \varphi + \text{nc } \psi \\
\text{nc } (\varphi \to \psi) &= 1 + \text{nc } \varphi + \text{nc } \psi \\
\text{nc } (\varphi \leftrightarrow \psi) &= 1 + \text{nc } \varphi + \text{nc } \psi
\end{aligned}$$

Ejemplo 7

Proposición. *Sea φ una fórmula bien formada de LProp. Entonces el número de conectivos $nc(\varphi)$ en φ es menor o igual que el número total de símbolos $ns(\varphi)$ en φ:*

$$nc(\varphi) \leq ns(\varphi)$$

Demostraremos el resultado por **inducción estructural** sobre la definición de las fórmulas lógicas.

Caso base:
Las fórmulas proposicionales más simples son:

- ⋆ Una variable proposicional, como p.

- ⋆ Las constantes lógicas $\top$ y $\bot$.

En todos estos casos, el número de conectivos es 0 y el número total de símbolos es 1. Claramente, $0 \leq 1$. Por lo tanto, el caso base se cumple.

Hipótesis de Inducción
Supongamos que el resultado es cierto para dos fórmulas proposicionales φ y ψ. Es decir, supongamos

que para φ y ψ, el número de conectivos es menor o igual que el número total de símbolos.

$$nc(\varphi) \leq ns(\varphi), \qquad nc(\psi) \leq ns(\psi).$$

Negación: Si φ es una fórmula, entonces $\neg\varphi$ es una fórmula. En este caso:

- ⋆ El número de conectivos en $\neg\varphi$ es $1 + nc(\varphi)$.
- ⋆ El número total de símbolos en $\neg\varphi$ es $1 + ns(\varphi)$.

Por la hipótesis de inducción, $nc(\varphi) \leq ns(\varphi)$, y al sumar 1 a ambos lados, se obtiene $1 + nc(\varphi) \leq 1 + ns(\varphi)$. Por lo tanto, el resultado se cumple para $\neg\varphi$.

Conjunción, disyunción, implicación, bicondicional: Si φ y ψ son fórmulas, entonces $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \to \psi$, y $\varphi \leftrightarrow \psi$ son fórmulas. Consideremos $\varphi \wedge \psi$ (los otros casos son análogos):

- ⋆ El número de conectivos en $\varphi \wedge \psi$ es $1 + nc(\varphi) + nc(\psi)$.
- ⋆ El número total de símbolos en $\varphi \wedge \psi$ es $1 + ns(\varphi) + ns(\psi)$.

Por la hipótesis de inducción, $nc(\varphi) \leq ns(\varphi)$ y $nc(\psi) \leq ns(\psi)$. Al sumar estas desigualdades y añadir 1, obtenemos:

$$1 + nc(\varphi) + nc(\psi) \leq 1 + ns(\varphi) + ns(\psi).$$

Por lo tanto, el resultado se cumple para $\varphi \wedge \psi$.

Conclusión: Por inducción estructural, el número de conectivos en una fórmula proposicional es menor que el número total de símbolos en la fórmula.

Ejercicio 5

1. Define recursivamente la función que obtiene el número de presencias de fórmulas atómicas en una fórmula `at`.
2. Define recursivamente la función que obtiene la lista de subfórmulas de una fórmula proposicional `sf`.
3. Demuestra usando inducción estructural:
   Proposición. *Si φ es una fórmula, entonces la longitud de la lista de subfórmulas de φ es igual a la suma del número de presencias de variables proposicionales de φ con el número de conectivos que figuran en φ. Es decir,*

$$\texttt{length}\,(\texttt{sf}\,(\varphi)) = \texttt{at}\,(\varphi) + \texttt{nc}\,(\varphi)$$

Nota: `length` calcula la longitud de una lista.

Sustitución

Una operación sintáctica fundamental en lógica proposicional es la sustitución. En una fórmula φ se reemplaza una variable proposicional p por una fórmula ψ, generando así una nueva fórmula, denotada como $\varphi[p := \psi]$. Esta fórmula se obtiene al sustituir todas las ocurrencias de p en φ por ψ.

Esta operación, conocida como sustitución textual, se define de manera recursiva de la siguiente forma:

$$\begin{array}{rcl}
p[p := \psi] & = & \psi \\
q[p := \psi] & = & q, \text{ si } p \neq q \\
\top[p := \psi] & = & \top \\
\bot[p := \psi] & = & \bot \\
(\neg\varphi)[p := \psi] & = & \neg(\varphi[p := \psi]) \\
(\varphi \wedge \chi)[p := \psi] & = & (\varphi[p := \psi] \wedge \chi[p := \psi]) \\
(\varphi \vee \chi)[p := \psi] & = & (\varphi[p := \psi] \vee \chi[p := \psi]) \\
(\varphi \to \chi)[p := \psi] & = & (\varphi[p := \psi] \to \chi[p := \psi]) \\
(\varphi \leftrightarrow \chi)[p := \psi] & = & (\varphi[p := \psi] \leftrightarrow \chi[p := \psi])
\end{array}$$

Ejemplo 8

$$(p \to p \vee q)\,[p := p \vee q] = ((p \vee q) \to (p \vee q) \vee q)$$

Se define también la sustitución simultánea de n variables proposicionales por n fórmulas. Este proceso se denota como $\varphi[p_1, \ldots, p_n := \psi_1, \ldots, \psi_n]$, o también como $\varphi[\vec{p} := \vec{\psi}]$, donde $\vec{p}$ representa el conjunto de variables proposicionales $(p_1, \ldots, p_n)$, y $\vec{\psi}$ representa el conjunto de fórmulas $(\psi_1, \ldots, \psi_n)$. Esta operación se define recursivamente de forma similar a la sustitución textual, considerando cada variable y su correspondiente fórmula simultáneamente.

$$\begin{array}{rcl}
p_i[\vec{p} := \vec{\psi}] & = & \psi_i, \text{ si } p_i \in \vec{p} \\
q[\vec{p} := \vec{\psi}] & = & q, \text{ si } q \notin \vec{p} \\
\top[\vec{p} := \vec{\psi}] & = & \top \\
\bot[\vec{p} := \vec{\psi}] & = & \bot \\
(\neg\varphi)[\vec{p} := \vec{\psi}] & = & \neg(\varphi[\vec{p} := \vec{\psi}]) \\
(\varphi \wedge \chi)[\vec{p} := \vec{\psi}] & = & (\varphi[\vec{p} := \vec{\psi}] \wedge \chi[\vec{p} := \vec{\psi}]) \\
(\varphi \vee \chi)[\vec{p} := \vec{\psi}] & = & (\varphi[\vec{p} := \vec{\psi}] \vee \chi[\vec{p} := \vec{\psi}]) \\
(\varphi \to \chi)[\vec{p} := \vec{\psi}] & = & (\varphi[\vec{p} := \vec{\psi}] \to \chi[\vec{p} := \vec{\psi}]) \\
(\varphi \leftrightarrow \chi)[\vec{p} := \vec{\psi}] & = & (\varphi[\vec{p} := \vec{\psi}] \leftrightarrow \chi[\vec{p} := \vec{\psi}])
\end{array}$$

Ejemplo 9

$$(p \to p \vee q)\,[p, q := p \vee q, r] = ((p \vee q) \to (p \vee q) \vee r)$$

Ejercicio 6

Realiza las siguientes sustituciones:

1. $(p \wedge (\neg q \vee r)) \to (p \vee \neg r)[p := (q \vee s)]$
2. $(p \wedge q) \to (\neg p \vee r)[p, q, r := (s \vee t), \neg r, p \wedge t]$

7. Automatización

La automatización de la sintaxis en lógica computacional es fundamental para formalizar y manipular expresiones lógicas de manera precisa y sin ambigüedades. Mientras que en lógica matemática la sintaxis se estudia desde un punto de vista teórico, en lógica computacional se implementa en lenguajes como Haskell para definir estructuras inductivas que permitan representar y procesar fórmulas de manera automática. Esto facilita la construcción de analizadores sintácticos, la validación de expresiones bien formadas y la generación de árboles de sintaxis, elementos importantes para la interpretación y transformación de lenguajes formales.

Haskell es una opción ideal para esta tarea debido a su naturaleza declarativa y su soporte nativo para estructuras de datos recursivas. Su sistema de tipos estático permite definir con precisión la estructura de las fórmulas lógicas, asegurando que las operaciones sobre ellas sean correctas desde el punto de vista sintáctico. Además, su capacidad para manejar reconocimiento de patrones facilita la implementación de funciones que operan sobre fórmulas lógicas, como analizadores sintácticos, funciones de sustitución y generadores de árboles de derivación. Esto hace que el desarrollo de herramientas para la manipulación de lenguajes formales sea más intuitivo y seguro en comparación con lenguajes imperativos.

Revisemos un breve esquema sobre cómo integrar estas técnicas de automatización con los conceptos estudiados hasta ahora. El laboratorio explora estos puntos con mayor profundidad.

El Tipo de Dato LProp

La gramática que especifica el lenguaje de la lógica proposicional puede ser descrita en Haskell mediante el siguiente tipo de dato:

```
data LProp = Var String
           | Cons Bool
           | Not LProp
           | And LProp LProp
           | Or LProp LProp
           | Impl LProp LProp
           | Equiv LProp LProp
           deriving (Eq, Show)
```

Observación 2

Puesto que toda fórmula atómica es fórmula de LProp, se incluyen los constructores de éstas fórmulas en LProp para simplificar la definición del tipo de dato.

Habiendo definido el tipo de dato LProp, ahora es posible crear expresiones de la lógica proposicional, por ejemplo, la fórmula $(p \wedge q) \vee r \to p$ se ve como:

```
let p = Var "p" ; q = Var "q" ; r = Var "r" in
    Impl ( Or ( And p q ) r ) p
```

Precedencia y Asociatividad de Operadores

Se pueden definir reglas de asociatividad y precedencia para los constructores de nuestro nuevo tipo de dato:

```
infixl 5 `And `,`Or `
infixl 4 `Equiv `
infixr 3 `Impl `
```

Esto nos permite definir fórmulas sin usar paréntesis:

```
> let p = Var "p" ; q = Var "q" ; r = Var "r" in  p `Impl ` q `Or` r
Impl ( Var "p" ) ( Or ( Var "q" ) ( Var "r" ) ) -- p -> (q \/ r)
```

Sustitución

En Haskell se puede definir una sustitución por medio de un sinónimo de tipo:

```
type Sustitucion = (String, LProp)
```

De esta manera, el tipo de la función sustitución (textual) quedaría como:

```
sustituye :: LProp -> Sustitucion -> LProp
```

Continuaremos explorando algunos de estos conceptos relacionados con la implementación en Haskell en futuras notas. Para mayor profundización se recomienda consultar [6].

8. Conclusiones

El estudio de la sintaxis en lógica proposicional es fundamental para la construcción de lenguajes formales precisos y sin ambigüedades. A través de la definición de un alfabeto, reglas de formación y gramáticas, es posible establecer expresiones bien formadas que constituyen la base del razonamiento lógico y computacional. El uso de árboles de sintaxis y reglas de precedencia permite una interpretación estructurada de las fórmulas, asegurando una única representación sintáctica. Además, las operaciones como la sustitución y la aplicación de inducción estructural facilitan el análisis y transformación de expresiones, proporcionando herramientas esenciales para el desarrollo de software, la verificación formal y la implementación de solucionadores SAT.

La automatización de estos conceptos mediante lenguajes funcionales como Haskell resulta especialmente útil, ya que permite representar estructuras inductivas de manera declarativa y operar sobre ellas con un sistema de tipos robusto. La implementación de la lógica proposicional en Haskell, a través de definiciones recursivas y reconocimiento de patrones, no solo facilita la manipulación de expresiones lógicas, sino que también abre la puerta a aplicaciones en inteligencia artificial, verificación de modelos y teoría de lenguajes de programación. Este enfoque computacional refuerza la importancia de la lógica en las ciencias de la computación y destaca su papel en la resolución de problemas mediante técnicas formales y automatizadas.

Referencias

[1] Enderton, H. B. (2001). *A Mathematical Introduction to Logic* (2nd ed.). Elsevier.

[2] Huth, M., & Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning about Systems* (2nd ed.). Cambridge University Press.

[3] Mendelson, E. (2015). *Introduction to Mathematical Logic* (6th ed.). CRC Press.

[4] van Benthem, J. (2010). *Logic in Games*. The MIT Press.

[5] Sipser, M. (2012). *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.

[6] Loyola Cruz, L. F. (2023). *Manual de Prácticas para la Asignatura de Lógica Computacional* (Proyecto de Apoyo a la Docencia). Universidad Nacional Autónoma de México.

[7] Miranda Perea, F. E., Reyes, A. L., González, L. del C., & Linares, S. (2018). *Lógica Computacional: Notas de clase.* Facultad de Ciencias, Universidad Nacional Autónoma de México.