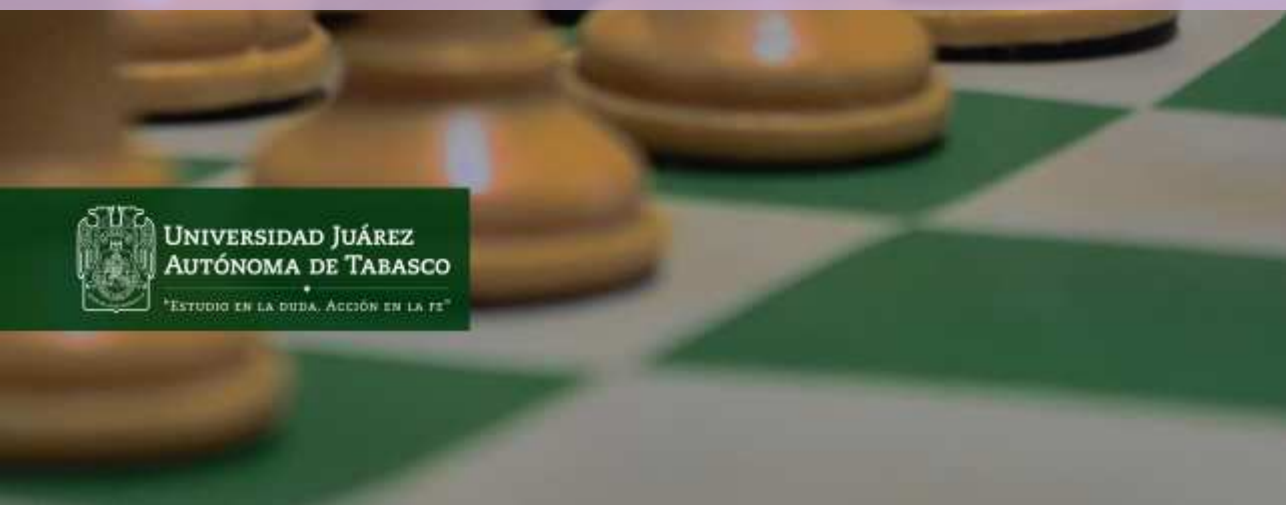
Lógica Computacional

Una perspectiva funcional con Python

Abdiel E. Cáceres González



Universidad Juárez Autónoma de Tabasco

"Estudio en la duda. Acción en la fe"

Lógica Computacional

Una perspectiva funcional con Python

COLECCIÓN
HÉCTOR OCHOA BACELIS
Textos de enseñanza de ciencias básicas

Guillermo Narváez Osorio

Rector

Hermicenda Pérez Vidal

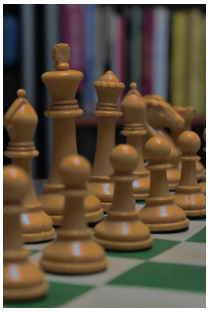
Directora de la División Académica de Ciencias Básicas

Universidad Juárez
Autónoma de Tabasco
"Estudio en la duda. Acción en la fe"

Lógica Computacional

Una perspectiva funcional con Python

Abdiel E. Cáceres González



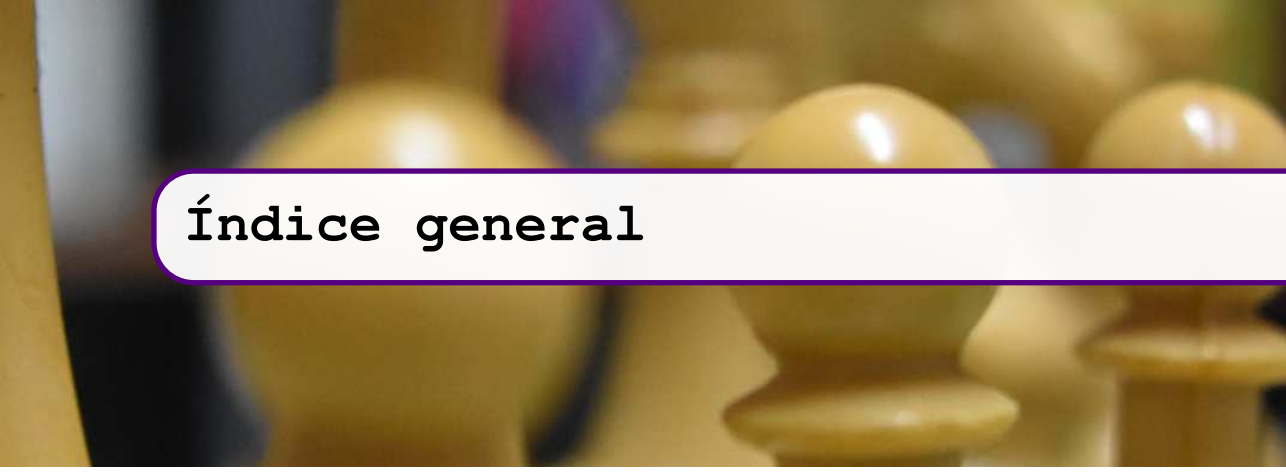
Fotografía en portada y título de capítulos:
por: Abdiel E. Cáceres González,

Primera edición, 2025

www.ujat.mx
ISBN 978-607-2628-56-4
DOI: https://doi.org/10.52501/ujat.002
Para su publicación esta obra ha sido dictaminada por el sistema académico de pares ciegos. Los juicios expresados son responsabilidad del autor y fue aprobada para su publicación.

Hecho en Cunduacán, Tabasco, México

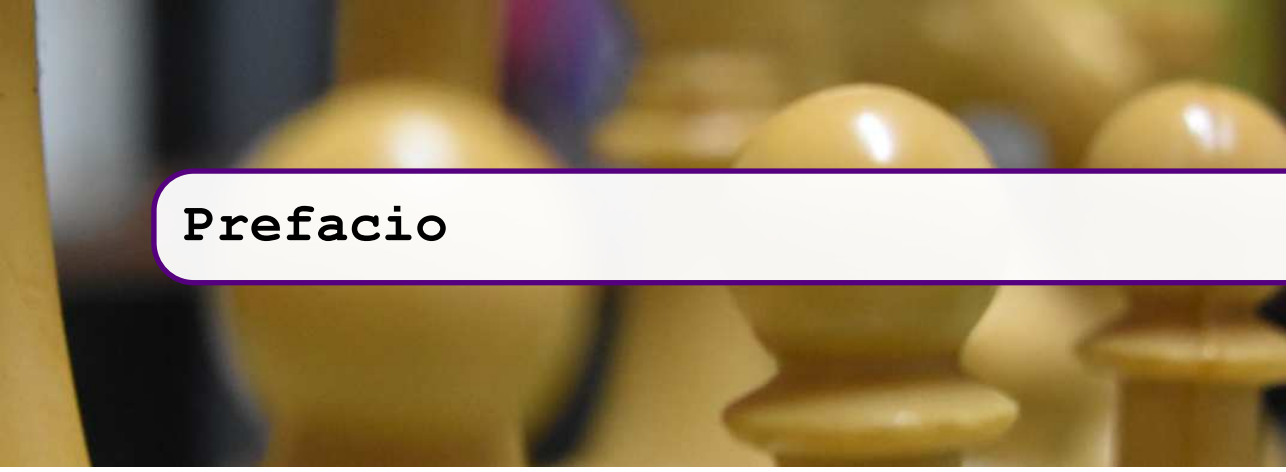
Para Amparo y Sara Camila



Índice general

IV

Apéndices



Prefacio

Este libro cubre el material de estudio para un curso de pregrado de Lógica Computacional. El curso tiene el propósito de introducir al lector en el campo de estudio de las Ciencias Computacionales desde dos frentes, el de los fundamentos teóricos de la computación y el de la programación funcional.

Este documento está dirigido principalmente a las personas de habla hispana que participan en un primer curso de lógica computacional, por lo que una primera meta es expresar con sentido lógico una o más sentencias escritas, de modo que se logre una secuencia de expresiones con sentido lógico. Una segunda meta es escribir funciones en `Python` que describan formalmente los conceptos del cómputo de predicados; finalmente se incluye un par de áreas de aplicación de la lógica computacional, la lógica de Hoare para la verificación de programas; y los circuitos digitales.

Se usa `Python` para implementar los conceptos de lógica. La decisión se tomó considerando que es un lenguaje rápido de aprender [RND17] ya que no utiliza muchas palabras reservadas, lo que es muy útil a la hora de expresar los conceptos y procedimientos de lógica.

El texto está presentado en tres diferentes notaciones: en el lenguaje natural en español; en el lenguaje formal de matemáticas; y en el lenguaje formal de `Python`, esto con el fin de comprender la correspondencia simbólica entre estos lenguajes.

En este libro se definen los conceptos desde tres perspectivas: de manera natural, efectiva y formal; sin embargo, aún en los últimos temas se mantiene un nivel de abstracción lo suficientemente alto, como para comprender claramente tanto el concepto como el significado de cada expresión.

Organización del libro

El contenido de este libro se ha dividido en tres partes:

1. **Cálculo proposicional.** Se tratan principalmente los temas de valores de verdad, sintaxis y semántica de las proposiciones lógicas; se estudian las reglas de inferencia para interactuar con proposiciones y llegar a conclusiones válidas.
2. **Cómputo de predicados.** Se incluyen temas como expresiones atómicas, operadores; predicados y funciones, y finalmente cuantificadores.
3. **Lógica aplicada.** Se muestran dos campos de aplicación de la lógica en las ciencias computacionales: la verificación de software y los circuitos digitales.

Al final del texto se muestra la colección completa de las definiciones creadas en `Python`. El lector puede copiarlas y probarlas, sin embargo, es aconsejable que las escriba por sí mismo, con lo que se logrará un mejor aprendizaje.

Tipografía

Para beneficio del lector, el texto se ha optimizado para una lectura basada en diferencias tipográficas, es decir, se ha escrito con diferentes tipos de letra para distinguir el texto en lenguaje natural, en el lenguaje simbólico de matemáticas y en el lenguaje formal de la programación.

Dado que a lo largo del texto se utilizan términos de diferentes clases, aquí se presenta un resumen de las clases de palabras y la forma en cómo se presentan en el texto. Podemos hacer la siguiente clasificación:

Clase I: Palabras en el lenguaje natural. Es el lenguaje base con el que se escribe la mayor parte de este texto. Se escribe con una tipografía de tipo `Latin Modern Roman`. Por ejemplo en la sentencia «el seno de equis es mayor o igual a cinco».

Clase II: Palabras y símbolos matemáticos. Usa tipografía llamada `Computer Modern`, aunque también se utilizan otros conjuntos de tipografías como `amsfont` o `amsmath`. Por ejemplo «$sen(x) \geq 0.5$». Aunque hay quien prefiere escribir los nombres de las funciones con letra `roman` como en «$\mathrm{sen}(x) \geq 0.5$», yo prefiero usar la primera, principalmente para hacer diferencia entre el texto normal y el matemático.

Clase III: Palabras y símbolos en código fuente. Aquí se incluyen las descripciones algorítmicas de los procedimientos, palabras reservadas en el lenguaje de programación y símbolos típicos en el lenguaje de programación. La tipografía utilizada es `TrueType`, que parece como una impresión hecha por una máquina de escribir antigua. Por ejemplo «`sen(x) >= 5`».

Los recursos tipográficos para enfatizar palabras son las comillas españolas «·», que se prefieren usar en el texto normal, ya que las comillas inglesas `"·"` se utilizan en el código fuente de `Python` para delimitar cadenas de caracteres. Por la misma razón, es decir, para no causar confusión, he preferido el uso de corchetes [·] en lugar de los paréntesis (·), porque los paréntesis se usan frecuentemente en la notación matemática y también en la notación del código fuente.

El símbolo '⊳', usado en el texto escrito en lenguaje natural, indica que inicia un comentario y termina hasta el final de la línea.

Para obtener un mayor beneficio, es aconsejable estudiar este texto teniendo a un lado una computadora con un entorno de desarrollo integrado de `Python` para escribir y comprobar cada definición.

Las referencias bibliográficas incluyen textos desde nivel medio superior en adelante. Se han seleccionado textos clásicos y recientes para abarcar un amplio rango de líneas de pensamiento.

En el índice alfabético se enlistan algunas palabras reservadas de `Python` como `if`, las funciones creadas en el texto, por ejemplo `neg`, y algunas palabras importantes. El lector puede referirse a este cuando lo considere oportuno.

Código fuente

En relación al código fuente, este se emplea en cuatro diferentes situaciones:

1. **Dentro de un párrafo.** La tipografía empleada hace la diferencia entre un texto alusivo a código fuente y texto que no lo es, por ejemplo:

   > La sentencia «El perro es bravo» se puede traducir en código fuente como un identificador `elPerroEsBravo`.

2. **Como ejemplo.** En ocasiones es necesario ejemplificar un caso mediante un algorítmo o en código fuente, también se puede presentar como una explicación de la sintaxis que se debe seguir al escribir una sentencia formal en el lenguaje de programación, por ejemplo:

   > La sintaxis de una expresión condicional `if` es la siguiente:
   >
   > ```
   > if ⟨expr-bool⟩:
   >     ⟨bloq-expr⟩
   > ```

3. **Como código fuente.** Los programas, scripts o funciones en `Python` se han escrito en apartados enmarcados, como el siguiente que contiene una definición para sumar dos números enteros no negativos:

   ```
   1  def suma(x,y):
   2      """
   3      Este es un ejemplo de la definición de una función en Python
   4      """
   5      if x == 0: # si el número x es 0 ...
   6          return y # devuelve el valor de y
   7      else:
   8          return suma(x-1,y+1) # de otro modo, decrementa x e incrementa y.
   ```

   El código escrito en este tipo de apartados se puede [y se debe] transcribir a la computadora, usando el IDE seleccionado por el lector. Para facilitar el estudio del código, se cuenta con números de línea al costado izquierdo; en las funciones se cuenta con comentarios de documentación, lo que permite conocer el propósito del código escrito.

4. **Como interacciones.** Al programar en el IDE de `Python`, se acostumbra escribir código y comprobar el correcto funcionamiento de la definición haciendo interacciones con el sistema. Las interacciones están escritas en un segmento apartado y bien delimitado con un contorno.
   Las interacciones se escriben para que el lector, al escribir su propio código, compare la salida que le ofrece su interacción con ofrecida en este texto. Tanto las interacciones como el código fuente se han tipografiado de manera similar a la que se presenta en el propio IDE de `Python`, para que la experiencia de aprendizaje del lenguaje sea más amable.

Todas las definiciones hechas en Python son probadas mediante ejemplos, que se han enmarcado en un recuadro para simular una interacción, como en el siguiente, en donde se prueba la función de suma [anterior]:

```
>>> suma(3,4)
7
>>> suma(5, suma(3,4))
12
>>>
```

Observaciones y notas sobre el lenguaje

En el texto se incluyen observaciones que por alguna razón son destacables, esto se hace mediante un ícono que llama la atención al párrafo que describe la nota, por ejemplo:

Aunque no es obligatorio, es buena idea incluir los tipos en los parámetros de entrada de las funciones y el tipo del valor de salida, por ejemplo:

```
def suma(a: int, b:int)-> int:
    pass
```

Se escribe una nota sobre el lenguaje de programación cuando se utiliza por primera vez alguna expresión o función primitiva de Python. Esto se hace mediante un breve comentario donde se explica la sintaxis y el uso de tal función. Este tipo de comentarios se señalan con el logotipo del lenguaje:

La función `enumerate(iterable, start=0)` recibe un objeto iterable como una lista, una tupla o un string y devuelve un objeto `enumerate` que es un empaquetado de un contador y los valores obtenidos al iterar sobre ese `iterable`.

```
>>> Grado = ['cabo', 'sargento', 'teniente', 'capitan']
>>> list(enumerate(Grado))
[(0, 'cabo'), (1, 'sargento'), (2, 'teniente'), (3, 'capitan')]
>>>
```

Herramientas de diseño y diagramación

Para la edición y diagramación de este libro he empleado algunas herramientas de libre uso, las herramientas más importantes son el entorno de edición integrado Kile [https://kile.sourceforge.io/] que trabaja con LaTeX[https://www.latex-project.org/].

Este texto se ha escrito en LaTeX con el paquete `listings` para escribir el código fuente. La versión de Python que se utilizó es la versión 3.10 de Python. Todos estos programas utilizados tienen licencia de libre uso.

Abdiel Emilio Cáceres González
Marzo 2025

I

Cálculo proposicional

Introducción a la lógica 1

1.1 Lógica en la historia

La lógica, en general, es una disciplina filosófica y matemática que estudia las reglas del razonamiento válido y la inferencia correcta. Se ocupa de la estructura del lenguaje, las formas de argumento y las reglas que permiten establecer la validez de los argumentos.

La lógica matemática estudia los principios y métodos usados para razonar de manera precisa y rigurosa, centrándose en el análisis formal de los argumentos y la validez de los mismos.

La lógica computacional es una aplicación de la lógica matemática en la computación. Aunque coinciden en muchos aspectos, una diferencia substancial es el uso de lenguajes de programación para definir conceptos y procedimientos que son utilizados para encadenar razonamientos.

Otro uso de la lógica computacional se encuentra en el diseño y verificación de programas y sistemas de cómputo, permitiendo asegurar su corrección y fiabilidad. Esto se logra mediante métodos formales que comprueban exhaustivamente el cumplimiento de especificaciones y la ausencia de errores lógicos o inconsistencias, además de otras áreas que se revisarán más adelante con un poco más de detalle [página 26].

Sin embargo, me parece importante esbozar la historia de la lógica, con el fin de tener conciencia del largo camino que se ha seguido desde sus inicios hasta la actualidad y su impacto en el campo de estudio que nos ocupa.

1.1.1 Los principios de la lógica

El desarrollo de la lógica occidental, la lógica que más ha influenciado a las Ciencias Computacionales, tiene sus raíces en la antigua Grecia, donde se establecieron los primeros fundamentos de esta disciplina. Uno de los filósofos que abordaron el tema fue Parménides de Elea [figura 1.1.A], quien en el siglo V a. C. escribió el poema «Sobre la Naturaleza», en donde una parte habla sobre la *vía de la verdad*, sentando las bases sobre la no contradicción, es decir, algo que es, no puede no ser y no hay más posibilidades [ZM16]. Sin embargo, fue con los trabajos de Aristóteles [figura 1.1.B] que la lógica comenzó a tomar forma como una disciplina sistemática.

(A) (B)

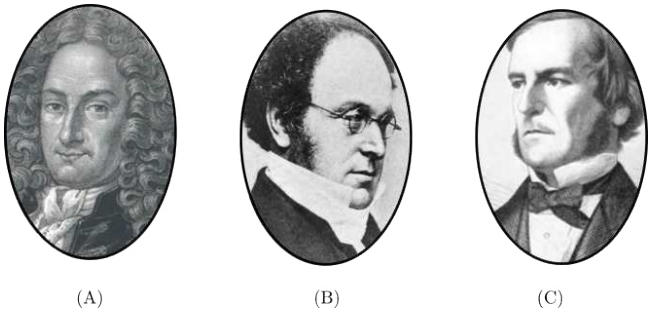
Figura 1.1: *(A) Parménides de Elea [Siglo V a. C.]. (B) Aristóteles [384 - 322 a. C.].*

Aristóteles, en el siglo IV a. C., desarrolló un sistema lógico formal conocido actualmente como lógica aristotélica [Boc68]. En su obra «Órganon», estableció las bases del razonamiento válido mediante el estudio del silogismo, formulando las reglas formales que permiten determinar su validez. Su trabajo estableció los fundamentos de la lógica aristotélica, influyendo en el desarrollo de la lógica y la argumentación.

Con el fundamento aristotélico, la lógica pasó un largo período durante la Edad Media sin avances considerables, ya que gran parte del pensamiento filosófico se dedicó casi por completo a la teología. Sin embargo, en el siglo XII, el filósofo y lógico inglés Guillermo de Ockham introdujo el principio de la navaja de Ockham, que postulaba que las explicaciones más simples son generalmente las más plausibles [Boc68, p. 239]. Este principio tuvo un impacto significativo en la lógica, ya que las conclusiones derivadas de las premisas establecidas generalmente ofrecen varias rutas de razonamiento, por lo que siguiendo el principio de la navaja de Ockham se debe elegir el camino más simple.

1.1.2 Historia de la lógica matemática

En el siglo XVII, el matemático y filósofo alemán Gottfried Wilhelm Leibniz [figura 1.2.A] hizo contribuciones clave al campo de la Lógica. Leibniz desarrolló una notación simbólica para los conceptos lógicos y propuso la idea de una «lengua universal» en la que todos los pensamientos pudieran expresarse mediante símbolos [DO82, p. 165]. Sus ideas establecieron los cimientos para el desarrollo de la lógica algebraica y simbólica, que se convertirían en la base de la lógica moderna en los siglos posteriores.



(A) (B) (C)

Figura 1.2: *(A) Gottfried Wilhelm Leibniz. (B) Augustus De Morgan. (C) George Boole.*

En el siglo XIX, Augustus De Morgan [figura 1.2.B], matemático y lógico británico, publicó en 1847 la obra *Lógica formal: O el cálculo de inferencia, necesaria y probable* [DM47]. De Morgan fue uno de los primeros matemáticos en desarrollar una notación simbólica para representar las proposiciones lógicas y algunos operadores lógicos. También introdujo los «diagramas de Venn» como una forma gráfica de representar conjuntos.

Por su parte, George Boole [figura 1.2.C], un matemático inglés, sentó las bases de la lógica matemática. En 1847, publicó su obra *El análisis matemático de la lógica* [Boo48], donde introdujo el álgebra booleana. Boole estableció una correspondencia entre la lógica y las operaciones matemáticas, permitiendo así una representación simbólica de las proposiciones y las operaciones lógicas. Su trabajo fue fundamental para el desarrollo posterior de la lógica moderna y sentó las bases para el enfoque algebraico.

A finales del siglo XIX y principios del XX, Frege [figura 1.3.B], un matemático y filósofo alemán, publicó su obra *Fundamentos de la aritmética* en 1884, donde introdujo el cálculo lógico de segundo orden y desarrolló una notación simbólica más sofisticada para la lógica. En el mismo período David Hilbert [figura 1.3.A] presentó su famoso conjunto de 23 problemas conocido como los *Problemas de Hilbert*.

Los problemas de Hilbert se presentaron en 1899, fueron expuestos en el Congreso Internacional de Matemáticos en París en 1900, y se publicaron en 1902 [Hil02]. En particular, el décimo problema de Hilbert, que cuestiona si existe un algoritmo que determine si una ecuación diofántica [polinomio con coeficientes enteros] tiene soluciones enteras [Dav73, Mat93], dio inicio a las Ciencias Computacionales, porque años más tarde sería retomado y estudiado para encontrar un método general [algoritmo] para resolver un tipo específico de problema matemático, las ecuaciones diofánticas.

A principios de 1900, Bertrand Russell [figura 1.3.C] trabajó en la formulación de una teoría de tipos para evitar las paradojas en la lógica, y publicó junto a Alfred North Whitehead la monumental obra *Principia Mathematica* en 1910. En este trabajo, establecieron los fundamentos de la lógica matemática y desarrollaron una forma de razonamiento lógico basada en la teoría de conjuntos.

(A) (B) (C)

Figura 1.3: *(A) David Hilbert. (B) Friedrich Ludwig Gottlob Frege. (C) Bertrand Russell. Nota: Las imágenes fueron seleccionadas del período en que realizaron su aportación, a inicios de la década de 1900.*

1.1.3 Historia de la lógica en computación

En 1921, se publicó la obra *Introducción a una teoría general de las proposiciones elementales*, basada en la tesis doctoral de Emil Leon Post [figura 1.4.A] realizada años antes. Post demostró que el cálculo proposicional descrito en *Principia Mathematica* estaba completo, es decir, que todas las tautologías son teoremas, dados los axiomas y reglas establecidas en la obra.

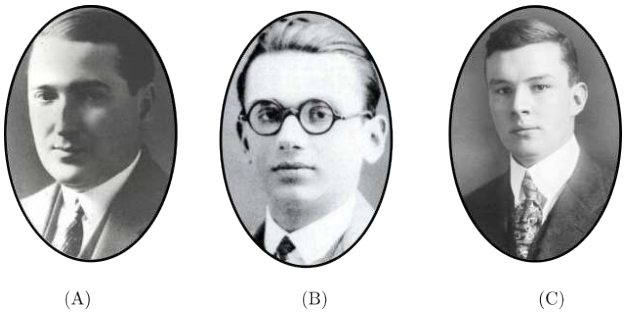
Mas tarde, en la década de 1930, se dieron a conocer importantes avances en el desarrollo de la Lógica Matemática y también en las Ciencias Computacionales, que han servido como fundamento para la aplicación de la lógica en campos como la Inteligencia Artificial, la Computación y la Filosofía de la Ciencia.

Kurt Gödel [figura 1.4.B], un lógico austriaco, contribuyó a la lógica y a los fundamentos de las matemáticas. En 1931, publicó su famoso teorema de incompletitud, que demostró que ningún sistema lógico formal puede ser completo y consistente a la vez. Este resultado tuvo un impacto profundo en la comprensión de los límites de la lógica y estableció que siempre existirán proposiciones matemáticas que no pueden ser probadas ni refutadas dentro de un sistema lógico formal.

También Haskell Brooks Curry [figura 1.4.C] realizó diversas contribuciones a la lógica matemática en la década de 1930 [Cur29, Cur34]. Sus aportaciones en la lógica combinatoria, el cálculo lambda, la paradoja de Curry y la teoría de la demostración han sido fundamentales en el desarrollo de estos campos y han tenido un impacto duradero en la lógica y la computación. Actualmente, el lenguaje de programación funcional `Haskell` lleva su nombre.

El desarrollo de la lógica formal es esencial para comprender los fundamentos teóricos de la computación, ya que proporciona las bases para el análisis riguroso de algoritmos. Diversos lógicos de renombre, como Alonzo Church, Stephen Cole Kleene y Andrei Andreyevich Markov Jr., realizaron aportes significativos en este campo, estableciendo principios y métodos que continúan influyendo en las teorías modernas de la computabilidad y el diseño de lenguajes de programación.

En la década de 1930, el lógico y matemático estadounidense Alonzo Church [figura 1.5.A], formuló el cálculo lambda, un sistema formal que se consolidó como una herramienta esencial para el estudio de la computabilidad [Chu32, Chu41].

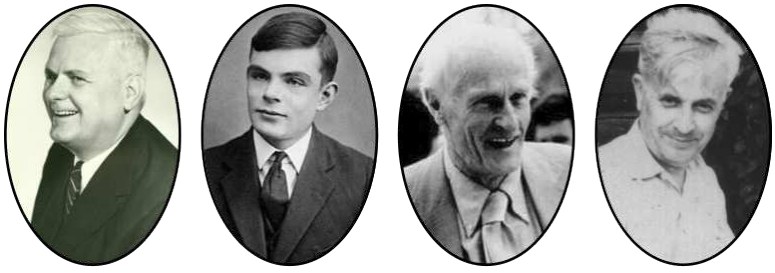


(A) (B) (C)

Figura 1.4: *(A) Emil Leon Post. (B) Kurt Gödel. (C) Haskell Brooks Curry.*

Church demostró que el cálculo lambda era equivalente al concepto de función recursiva, estableciendo así la base teórica para el campo de la computabilidad. Su trabajo sentó las bases de la teoría de la computación y fue fundamental para el desarrollo posterior de los lenguajes de programación y la teoría de la recursión.

Alan Mathison Turing [figura 1.5.B], un matemático y científico británico, realizó estudios importantes en Lógica y la Teoría de la Computación. En 1936, Turing propuso el concepto de la máquina de Turing [Tur36], un modelo teórico concebido para dar respuesta al décimo problema de Hilbert, que consise en un dispositivo capaz de realizar cualquier cálculo computacional, incluyendo las ecuaciones diofánticas. Esta idea sentó las bases para el estudio de la computabilidad y fue esencial en el desarrollo de las primeras computadoras electrónicas.



(A) (B) (C) (D)

Figura 1.5: *(A) Alonzo Church. (B) Alan Mathison Turing. (C) Stephen Cole Kleene. (D) Andrei Andreyevich Markov Jr.*

En la misma época, Stephen Kleene [figura 1.5.C], un matemático y lógico estadounidense, contribuyó significativamente a la Lógica Matemática y la Teoría de la Computación. Kleene trabajó en las funciones recursivas parciales, el cálculo lambda y en la teoría de la recursión generalizada, que proporcionó una base formal para el estudio de las funciones computables [Kle36, Kle81].

Andrei Andreyevich Markov Jr. [figura 1.5.D], un matemático soviético, también hizo contribuciones importantes al campo de la lógica y las ciencias computacionales, entre las que destaca un estudio teórico de algoritmos computacionales [MN88].

Estos lógicos de la primera mitad del siglo XX, Alonzo Church, Alan M. Turing, Stephen C. Kleene, Andrei A. Markov Jr., entre otros, fueron pioneros en el desarrollo de las Ciencias Computacionales y la Lógica Formal. Sus contribuciones fueron fundamentales para la comprensión de la computabilidad, la teoría de la recursión y el análisis de algoritmos. Además, sentaron las bases para el desarrollo de lenguajes de programación, la teoría de la complejidad computacional y otras áreas relacionadas con las ciencias de la computación. Su trabajo continúa siendo una parte fundamental de la educación y la investigación en este campo en la actualidad.

1.1.4 Influencia de la lógica en computación

El tema de lógica computacional está inmerso en el desarrollo de los programas de computadora y en el diseño de circuitos digitales de cualquier aparato electrónico. Gracias a la lógica, es posible establecer una relación entre los datos de entrada, y el resultado de salida como consecuencia lógica de la entrada. Además, está involucrada en todas las áreas de la computación. En seguida se describen solamente unas pocas y cómo está relacionada la lógica:

Teoría de la computación Se usan conceptos de lógica para proporcionar un marco formal en el estudio de la computabilidad y la complejidad. La lógica matemática y la lógica computacional se utilizan para definir y analizar conceptos clave como las máquinas de Turing, los lenguajes formales, las gramáticas y los autómatas.

Inteligencia Artificial: En este campo se usa la lógica computacional para representar el conocimiento y el razonamiento. Los sistemas expertos y los lenguajes de programación lógica, como Prolog, se basan en la lógica de predicados para representar el conocimiento y realizar inferencias lógicas. La lógica también es fundamental en el razonamiento automatizado, la planificación y la toma de decisiones.

Bases de Datos: En el campo de las bases de datos, la lógica juega un papel crucial en la especificación y el diseño de sistemas de gestión de bases de datos. El lenguaje SQL [*Structured Query Language*] se basa en la lógica relacional para consultar y manipular datos en bases de datos relacionales. La lógica también se utiliza para definir las reglas de integridad y consistencia de los datos, así como para la optimización de consultas.

Verificación de software: La verificación de software se ocupa de garantizar que un programa cumpla con ciertas propiedades o especificaciones. La lógica temporal, por ejemplo, se utiliza para especificar y verificar propiedades dinámicas del software, como la corrección de concurrencia y la ausencia de interbloqueos. Además, las técnicas de modelado formal y la verificación de modelos se basan en lógicas temporales y lógicas de predicados para verificar propiedades de sistemas complejos.

Criptografía: En criptografía, la lógica se utiliza para el diseño y el análisis de protocolos seguros. Los formalismos de lógica se emplean para expresar propiedades como la confidencialidad, la integridad y la autenticidad. Además, la lógica desempeña un papel importante en la demostración de que los protocolos criptográficos son plenamente confiables y en el descubrimiento y resolución de vulnerabilidades en un sistema de cómputo.

Sistemas Operativos: En sistemas operativos se utiliza lógica para definir algoritmos de planificación, gestionar recursos, controlar dispositivos, manejar errores y organizar la información. Proporciona un enfoque formal y estructurado para el diseño y la implementación de los componentes esenciales de un sistema operativo, garantizando su funcionamiento correcto y eficiente.

En general, podemos enunciar dos maneras principales en que la lógica formal está relacionada con la computación:

1. Para emitir una conclusión acerca de algo relativo a una pieza de software, por ejemplo en las áreas de computabilidad, verificación formal de software [RCS+22], razonamiento automático [Har09], demostración automática de teoremas [New00], entre otros.
2. Para construir aplicaciones. En el sentido de construir una cadena de razonamientos y conclusiones que transformen los datos de entrada en una conclusión de salida.

1.2 Lógica y sus divisiones

En esta sección se presenta una clasificación de las diferentes ramas de la lógica, destacando su relevancia y aplicaciones tanto en el ámbito matemático como en la computación. Se ofrece una visión general de las distintas ramas de la lógica, incluyendo la lógica clásica y no clásica; luego, se describen los principios básicos que sustentan la lógica computacional, incluyendo el cálculo proposicional y el cómputo de predicados, proporcionando las bases necesarias para el estudio y desarrollo de algoritmos y programas de computadora.

1.2.1 Subáreas de la lógica

Dentro de la lógica, existen diferentes subáreas, como la lógica proposicional, la lógica de predicados, la lógica modal y la lógica temporal, entre otras. La lógica computacional se sitúa dentro de este espectro y se centra en la aplicación de esos principios lógicos en la computación. Una clasificación de la lógica y algunas de las diferentes subáreas es la siguiente:

- Lógica
  - L. Clásica
    - – Lógica matemática: Se usan definiciones declarativas.
    - – Lógica computacional: Se usan definiciones efectivas.
  - L. No clásica
    - – Lógica k-valente
    - – Lógica difusa
    - – Lógica modal
    - – Lógica temporal
    - – Lógica epistémica
    - – Lógica no monotónica

Por supuesto, esta clasificación no es precisa, y tampoco es completa [ni pretende serlo], sin embargo sirve para ubicar, al menos en principio, el campo de estudio de la lógica computacional dentro de la lógica.

La lógica clásica es la referida hasta ahora, la que comenzó en la antigua Grecia y se ha ido refinando a lo largo del tiempo. La lógica clásica se basa en el principio del tercero excluido, esto es que una afirmación es verdadera o falsa, pero no hay una tercera opción. Se rige por las leyes clásicas del pensamiento, como el principio de identidad, que establece que una cosa es idéntica a sí misma; el principio de no contradicción, según el cual una afirmación no puede ser verdadera y falsa al mismo tiempo, y el principio del tercero excluido.

Se puede estudiar la lógica clásica desde al menos dos perspectivas, desde la lógica matemática y desde la lógica computacional.

La lógica matemática combina la lógica y la matemática para estudiar el razonamiento y la inferencia dentro de un contexto matemático. Se ocupa de desarrollar sistemas formales y reglas precisas para analizar y demostrar la validez de los argumentos matemáticos.

La lógica computacional es el estudio de la lógica, pero utilizando todas las herramientas computacionales disponibles, o dicho de otro modo y en una forma muy simplista, se trata de crear programas de computadora que permitan acercarse al conocimiento de la lógica. En la lógica computacional se utiliza un lenguaje de programación para representar y manipular el conocimiento, mientras que la lógica matemática establece su propio lenguaje y sus reglas de uso, que pueden ser aprovechadas por la lógica computacional.

En este libro se utilizan ambos acercamientos a la lógica, la lógica matemática y la computacional. La primera para representar el conocimiento y la segunda para comunicarlo a la computadora y realizar programas que guarden una estructura lógica en su fundamento.

1.2.2 Fundamentos de lógica computacional

Otra clasificación de la lógica tiene que ver con las bases para el razonamiento y el estudio de las estructuras formales y los conjuntos. Esta clasificación establece los cimientos para el desarrollo de otras ramas de las ciencias computacionales y la lógica:

- Lógica computacional
  - Cálculo proposicional
    - Proposiciones
    - Símbolos
    - Certeza y validez
    - Tablas de verdad
  - Cómputo de predicados
    - L. primer orden
    - L. orden superior
  - Teoría de Conjuntos
    - Elementos
    - Conjuntos
    - Propiedades

Este libro cubre en su mayoría esta clasificación, primero la lógica proposicional, donde se muestra cómo construir proposiciones lógicas y cómo combinar proposiciones para expresar proposiciones más complejas sin perder su sentido de validez.

En un segundo nivel, se muestra cómo crear predicados, con los que es posible generar expresiones que potencialmente lleguen a ser proposiciones con algún valor

de verdad que dependa de otra información; y aumentaremos el nivel al trabajar con expresiones de orden superior.

Finalmente, en la teoría de conjuntos se aplica el cálculo proposicional y de predicados para clasificar elementos que cumplan cierto predicado.

1.3 Pensamiento y razonamiento

En esta sección se exploran las diferentes formas en que las personas procesan la información para tomar decisiones, llegar a conclusiones y resolver problemas cotidianos, tomando decisiones basadas en la información disponible.

1.3.1 Razonamiento cotidiano

Las personas, en condiciones normales, realizan procesos mentales en los que toman información, que proviene de sus sentidos como la vista y el tacto; o bien información memorizada para enlazar pensamientos y así alcanzar una conclusión, esta actividad de pensar y construir una secuencia de pensamientos para llegar a una decisión es lo que llamamos «razonamiento», que se realiza en tres partes:

1. Analizar la información disponible.
2. Procesar la información, enlazando pensamientos.
3. Emitir una conclusión.

Es notable que estas mismas tres características se consideraron para suponer que la computadora podría llegar en algún momento a igualar la actividad del cerebro humano [Wie61, Tur50, VN58].

La vida de cada persona transcurre tomando decisiones basadas en conclusiones alcanzadas por medio de razonamientos [figura 1.6]. Hay conclusiones que se toman rápidamente. Por ejemplo una persona que está frente a un producto en el estante de una tienda y observa algunas características en el producto que no son de su agrado, luego emite su conclusión «este producto está dañado», simplemente porque ha observado un defecto tan notorio y obvio que ha llegado a esa conclusión.

Figura 1.6: *Para decidir se requiere información y una manera de enlazar pensamientos y emitir una conclusión. Fotografía:* Woman choosing frozen product in supermarket *por Laura James, derechos de libre uso,* https://www.pexels.com/photo/woman-choosing-frozen-product-in-supermarket-6097890/.

Lo que va de acuerdo con la propia percepción de la realidad es convertido en «cierto» y estamos de acuerdo. Otras cosas están en contra de la intuición o en contra del conocimiento y eso no es creído, por lo que no es parte del conocimiento verdadero, a menos que haya un convencimiento de lo contrario.

Hay diferentes maneras de razonar sin importar si la conclusión sea cierta o no: unas parten de lo general para obtener conclusiones en lo particular [deducción]; otra es lo contrario, parten de lo particular y generan conclusiones en lo general [inducción]; otra forma, el razonamiento lógico, es considerar una relación entre los datos de entrada y las conclusiones, siguiendo reglas preestablecidas.

1.3.2 El razonamiento lógico

La lógica es una manera de explicar cómo es el pensamiento racional y cómo se alcanzan las conclusiones a partir de un conocimiento previo. Siguiendo una serie de convenciones, una cadena de explicaciones racionales se convierte en un razonamiento lógico.

El razonamiento lógico se fundamenta en dos tipos de reglas:

1. Las reglas de escritura [sintaxis].
2. Las reglas de interpretación [semántica].

Para que un razonamiento pueda ser considerado «lógico», debe estar sintácticamente bien escrito, y ser semánticamente bien interpretado.

Ejemplo 1.1

Se sabe que «Todas las frutas son saludables», luego, un mercante dice que «las peras son frutas». Con lo que se construye el juicio: si «Las peras son frutas», y sé que «Todas las frutas son saludables», entonces podemos concluir o deducir que «las peras son saludables».

$$\frac{\begin{array}{r}\textit{Si}: \text{todas las frutas son saludables}\\ \textit{y}: \text{las peras son frutas,}\end{array}}{\textit{entonces}: \text{las peras son saludables.}}$$

Razonar lógicamente significa tomar como punto de partida algún conocimiento ya establecido, razonar aplicando las reglas conocidas y concluir algo. Eso que se ha concluido formará parte del conocimiento ya establecido. Luego, se pueden seguir encadenando razonamientos lógicos para obtener nuevos conocimientos que puedan ser utilizados en futuros razonamientos.

Si lo escribimos como una secuencia de pasos, el razonamiento lógico es:

Inicio: Define el problema o situación que requiere razonamiento lógico.

Paso 1: ***Recopilación de conocimiento.*** Reúne toda la información relevante disponible sobre el problema o situación en cuestión. Esto puede incluir datos, hechos, reglas, suposiciones, experiencias pasadas, etc.

Paso 2: ***Formulación de suposiciones.*** Genera posibles explicaciones o soluciones basadas en el conocimiento existente. Estas suposiciones deben ser coherentes con las reglas lógicas y los hechos establecidos previamente.

Paso 3: ***Prueba y evaluación.*** Evalúa cada suposición mediante razonamiento lógico y análisis crítico.

Paso 4: ***Generación de nuevo conocimiento.*** Si una suposición es lógicamente válida y está respaldada por pruebas y evidencias suficientes, se considera como nuevo conocimiento. Este nuevo conocimiento puede ser una conclusión, una regla

adicional o una solución al problema planteado inicialmente. Cada nuevo conocimiento es habilitado para ser utilizado posteriormente.

Paso 5: ***Iteración.*** Si ninguna suposición inicial se considera satisfactoria, regresa al paso 3 y genera nuevas suposiciones o ajusta las existentes. Repite el proceso de prueba y evaluación hasta que se encuentre una solución lógica aceptable.

Fin: Cuando se ha alcanzado una solución satisfactoria o se ha generado nuevo conocimiento válido, finaliza el proceso de razonamiento lógico.

Visto de un modo más adecuado para un computólogo, razonar es como programar, se inicia desde unos pocos conocimientos [los datos de entrada del programa], con los que se razona y se obtienen conclusiones [los procesos que efectúan las funciones y los valores que se obtienen de ellas]. De este modo es posible alcanzar una conclusión final, el nuevo conocimiento [el valor devuelto por el programa].

El razonamiento lógico es, entonces, una secuencia de juicios que mantienen entre sí relaciones lógicas, de modo que a partir de algunos juicios dados llamados *premisas* [datos de entrada], podemos alcanzar deductivamente un conocimiento que antes no teníamos y que se llama «conclusión» [información de salida]. Si se han observado las bases de la lógica, se asegura que la conclusión es válida.

La lógica es un área de estudio muy amplio, que es el fundamento de otras áreas de estudio que aparentemente son ajenas entre sí, como entre el derecho y las matemáticas. A pesar de las grandes diferencias entre estas áreas de estudio, se fundamentan en los mismos principios para llegar a conclusiones válidas.

1.3.3 Razonamiento algorítmico

Programar es vislumbrar el estado final de un proceso, aún antes de iniciado; lo que incluye repetir el mismo ejercicio por cada paso intermedio entre el inicio y el final de la tarea.

Imaginar el estado final de algo tiene que ver con la integración de conceptos; la comparación de resultados; la toma de decisiones para seleccionar la alternativa correcta; la memorización de palabras clave en el idioma computacional y por supuesto, lo que en ocasiones es más difícil, el ordenamiento secuencial de actividades a realizar.

En el razonamiento algorítmico se utiliza la lógica y la estructura de los algoritmos para analizar un problema, identificar los pasos necesarios para resolverlo y desarrollar un procedimiento lógico para llegar a la solución deseada.

Para desarrollar el pensamiento algorítmico, se puede practicar haciendo al menos las primeras tres actividades de la siguiente lista:

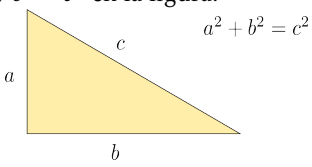
1. *Definir el problema:* Comienza por comprender claramente cuál es el problema que deseas resolver. Esto implica identificar los objetivos, las restricciones y los requisitos del problema.
2. *Analizar el problema:* Examina el problema en detalle y sepáralo en partes más pequeñas y manejables. Identifica las entradas necesarias para resolver el problema y las salidas que se esperan obtener.
3. *Diseñar el algoritmo:* Esta etapa implica desarrollar un plan lógico y detallado para resolver el problema. Puedes utilizar diferentes técnicas de diseño de algoritmos, como diagramas de flujo, pseudocódigo o estructuras de control, para representar la lógica del proceso paso a paso.

4. *Verificar* que el algoritmo siempre responde adecuadamente cuando se le dan las entradas adecuadas.
5. *Optimizar* el algoritmo.

Las actividades 4 y 5 requieren una mayor experiencia, en especial la número 5, pues no hay garantía de que se pueda realizar en cualquier caso. Se puede empezar con problemas computacionales simples, siempre que se consideren pasos de entrada de datos, procesamiento y salida de información. Analiza el siguiente ejemplo.

Ejemplo 1.2

El problema es calcular el valor de la hipotenusa de un triángulo rectángulo, de acuerdo con $a^2 + b^2 = c^2$ en la figura:



El primer paso es definir el problema: En este ejemplo se requiere el valor de c, que claramente es el objetivo que se debe alcanzar. Para esto contamos con los valores a y b.

El segundo paso es analizar el problema: Para alcanzar el objetivo se tienen que elevar los valores a y b al cuadrado cada uno, luego sumar esas cantidades. El resultado de eso es c^2, por lo que habrá que obtener la raíz cuadrada para llegar a c.

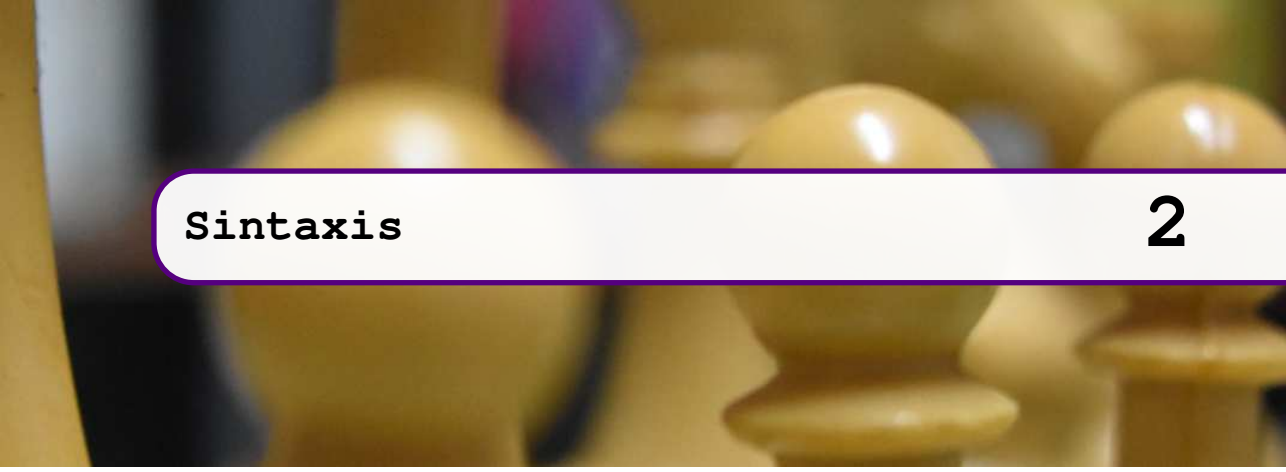
El tercer paso es diseñar el algoritmo:

Algoritmo HIPOTENUSA:
Requiere: a, b **dos números.**
1.- Obtener a^2, **el cuadrado de** a
2.- Obtener b^2, **el cuadrado de** b
3.- Obtener c^2, **la suma** $a^2 + b^2$
4.- Obtener c **como** $\sqrt{c^2}$
5.- Devolver el resultado c

Este algoritmo es simple, por lo que no se requiere optimización, pero si aumenta la dificultad del problema, es posible que sí se necesite optimizar algunas partes. Claramente hay pasos que podrían sugerir la creación de nuevos algoritmos, como obtener la raíz cuadrada, o incluso elevar un número a cualquier potencia.

El razonamiento algorítmico es esencial en áreas como la programación, la computación y la resolución de problemas. Permite descomponer un problema en pasos más pequeños y manejables, lo que facilita su resolución y eventual optimización. Además, también se utiliza en la verificación y demostración de la corrección de algoritmos, asegurando que sigan las reglas lógicas y produzcan los resultados esperados.

Este razonamiento implica utilizar la lógica y la estructura de los algoritmos para resolver problemas y tomar decisiones. Es una habilidad clave en la programación y la resolución de problemas complejos, permitiendo desarrollar estrategias lógicas y eficientes para alcanzar soluciones deseadas.



Sintaxis 2

2.1 Proposiciones

Una **proposición** es una expresión declarativa, es decir, una oración, con sujeto y predicado, que afirma o niega algo; también se les conoce como «expresiones enunciativas» [RAE10, p. 18]. De una proposición se puede decir que es cierto o falso aquello que se afirma o se niega, que es el sentido lógico de la proposición.

En el lenguaje español hay varios tipos de oraciones, la mayoría no son proposiciones, solo las oraciones declarativas son proposiciones.

Tipos de oraciones
- Declarativas [«En este momento es tarde»].
- Interrogativas [«¿Jacobo llegó a tiempo?»].
- Exclamativas [«¡Quita el florero de allí!»].
- Imperativas [«Quédate quieto»].

2.1.1 Proposiciones en el lenguaje cotidiano

Al enunciar una proposición, queremos declarar que algo es, por ejemplo «El sol es una estrella»; o bien que algo existe. Una proposición está relacionada con un conocimiento verdadero que ya ha sido demostrado o que se quiere demostrar, por ejemplo «hay vida en otros planetas» o «hay arena en el desierto».

☞ Una «proposición» no es lo mismo que una «preposición». Una preposición, que de acuerdo con la gramática en español, es una clase de palabras que establecen una relación, ya sea espacial, temporal o lógica, entre dos elementos de una oración, por ejemplo en la frase «El florero está sobre la mesa», la palabra «sobre» es una preposición. Hay muchas otras preposiciones, como las palabras «a», «ante», «con», «contra», «de», «desde», por mencionar algunas.

Cuando se enuncia una proposición, quien la recibe tiene la facultad de interpretar la oración y determinar si cree o no tal enunciado, todo de acuerdo a su conocimiento previo. Cuando no se cree en la proposición, aún se tiene la facultad de olvidarlo o recordarlo.

Ejemplo 2.1

Alicia y Roberto se encuentran después de algún tiempo de no verse. Luego de un breve saludo, Roberto le comenta a Alicia: – «Ayer llovió todo el día». Alicia, quien vive en la misma ciudad que Roberto, sabía que no había llovido tanto tiempo, por lo que quedó un poco confusa. Al principio no le creyó, pero luego pensó que era posible que en un lugar lloviera más que en otro, por eso terminó por creerlo.

Imagen generada por bluewillow.com [https://www.bluewillow.ai/]. Esta imagen fue creada utilizando tecnología de inteligencia artificial y representa una interpretación artística de dos personas conversando.

Después de haberlo pensado un poco, Alicia aceptó la proposición «ayer llovió todo el día» y ahora ella puede tener ese conocimiento listo para utilizarlo en otra oportunidad.

Al hablar o escribir, en ocasiones se utilizan expresiones declarativas. Se usan proposiciones para declarar una verdad relativa acerca de un objeto [Arn89].

Ejemplo 2.2

Las siguientes declaraciones son expresiones proposicionales:

- «Internet es un medio de comunicación».
- «Leer mejora la escritura».
- «El café produce enfermedades en el riñón». ▹ *Aunque no está plenamente comprobado, hay relación entre el consumo de café y enfermedades renales [WTT+17].*
- «Veinte es mayor que quince».
- «El sushi es una comida tradicional china». – *Es falso, porque es una comida japonesa.*
- «20 > 30». ▹ *También es posible incorporar expresiones en lenguaje de matemáticas.*
- «`a == b`». ▹ *Expresiones en código fuente.*
- «El botón primario del mouse está activado».
- «La variable contador tiene el valor 15».

Una característica de las proposiciones es que se pueden catalogar como cierto o falso. Esto es que, ante una proposición, se tiene la facultad de emitir un juicio y decir que la proposición es cierta o es falsa; dependerá de lo que se cree como verdad. Recuerda el ejemplo 2.1, y supón que Alicia no creyera lo que Roberto dijo, entonces la proposición sería falsa y eventualmente rechazada.

Hay otras expresiones que se pueden escribir [o decir] en el lenguaje cotidiano, pero que no son proposiciones, sino expresiones interrogativas, exclamativas o de cualquier otro tipo.

Ejemplo 2.3

Las siguientes expresiones no son proposiciones:

- «¿Cuándo empiezan las vacaciones?» ▹ *Esta una expresión interrogativa.*
- «¡Sírvete tu mismo!». ▹ *Es una orden, no una declaración.*
- «Todas las personas de esta universidad han estudiado lógica». ▹ *Actualmente no es posible dar un valor de verdad, antes de preguntar a cada persona implicada, aunque no es una proposición, podría llegar a serlo.*
- «$x + 5 = 28$». ▹ *También en matemáticas hay expresiones donde no es posible determinar si es cierto o falso, al menos antes de dar valor a las incógnitas.*

2.1.2 Proposiciones atómicas y moleculares

Es posible tener proposiciones como «Hoy tomé café y me siento feliz». Observa que esta proposición se puede subdividir y obtener de ella dos proposiciones más simples: «Hoy tomé café» y «Me siento feliz». De modo que podemos hacer una clasificación de las proposiciones:

Proposiciones
- Proposiciones atómicas — *Aquellas que no pueden ser subdivididas sin perder sentido lógico.*
- Proposiciones moleculares — *Aquellas que sí se pueden subdividir en una o más proposiciones.*

El nombre «atómica» o «molecular» se refiere a la posibilidad de que la expresión pueda o no ser subdividida [SH86]. Si una expresión atómica es dividida, cada parte pierde el sentido lógico y gramatical, a su vez pierde su calidad de proposición. Por otro lado, una expresión molecular se puede subdividir y obtener otras proposiciones sin que cada una de ellas pierda sentido.

Ejemplo 2.4

Las siguientes expresiones son proposiciones atómicas:

- «El gato es un andariego».
- «Alicia es buena amiga».
- «El pastel tiene muchas calorías».
- «El teclado tiene configuración de idioma ES-MX».

Las siguientes expresiones son proposiciones moleculares:

- «El café tiene leche y el agua está caliente».
- «Antonio está jugando o está en su cuarto».
- «Si mañana hay examen y no he estudiado, entonces no iré de vacaciones».
- «$8 \geq 5$ y $14 \leq 10$».

Las proposiciones atómicas también reciben los nombres de «simples» o «elementales» y las proposiciones moleculares también se llaman «compuestas» [Cur77, Arn89, BA09]; sin embargo, aquí se conservan los nombres «atómicas» y «moleculares» como un recordatorio del origen griego de la lógica y como tributo a los términos utilizados en *Principia Mathematica* [WR63].

2.1.3 Términos enlaces usados en lógica

Para crear proposiciones moleculares se requieren «términos enlaces» o «términos de enlace» [SH86]. Un enlace es un símbolo que se utiliza para unir proposiciones y generar nuevas proposiciones más complejas. Un enlace que puede ser un signo, un mnemónico, o palabras. Al usar diferentes enlaces es posible cambiar la interpretación y el significado de las proposiciones. Hay enlaces que se utilizan en una sola proposición y hay enlaces utilizados entre dos o más.

En el lenguaje común se utilizan diferentes enlaces gramaticales [FR09, DG06, MnR18], por ahora estaremos interesados en los siguientes cuatro, que son los de uso más frecuente:

- Enlaces lógicos
  - E. Opositivos
    - «*No*»
    - «*No es cierto que*»
    - «*Ni*»
    - «*Lo contrario de*»
  - E. Disyuntivos
    - «*o*»
    - «*O – o*»
  - E. Conjuntivos
    - «*y*»
    - «*A la vez – y*»
  - E. Implicativos
    - «*Si –, entonces*»
    - «*implica*»
    - «*es suficiente para*»
    - «*solo si*»
    - «*es necesario para*»

Ejemplo 2.5

Las siguientes proposiciones moleculares utilizan enlaces opositivos:

- «No es un delincuente». ▹ *El enlace aparece al inicio.*
- «$15 \neq 5$». ▹ *Se lee «15 no es igual que 5», el enlace es la palabra «no».*
- «Susana no tiene tiempo». ▹ *El enlace opositivo ocurre dentro de la frase.*
- «El vaso no está lleno». ▹ *No necesariamente debe estar vacío.*

Ejemplos de proposiciones moleculares que usan enlaces disyuntivos:

- «A Brenda le gusta el café con azúcar o con panela». ▹ *Se debe entender que Brenda gusta de tomar café con azúcar, pero también le gusta si está endulzado con panela.*
- «La taza o bien tiene café o bien tiene té». ▹ *La taza tiene uno u otro, pero no ambos.*

Estas proposiciones moleculares se han construido con enlaces conjuntivos:

- «A Brenda le gusta la ensalada con pepino y aguacate». ▹ *Si falta alguno de esos ingredientes, entonces no le gustará.*
- «El terreno es húmedo y el día es caluroso».
- «Los peces necesitan agua para vivir y el oro es un metal precioso».
- «$0 \leq 5 \leq 10$». ▹ *Esta expresión matemática se compone de dos expresiones que se juntan mediante una conjunción* $0 \leq 5$ *y* $5 \leq 10$.

Las siguientes proposiciones son implicativas:

- «Si mi equipo tiene más aciertos, entonces mi equipo gana el torneo».
- «Si llevas una dieta sana y haces ejercicio, entonces tendrás menos enfermedades».
- «Si el usuario ingresa la palabra clave correcta y responde correctamente el captcha».

☞ La palabra **captcha** proviene de las siglas en inglés *Completely Automated Public Turing test to tell Computers and Humans Apart*. Un captcha alude a una prueba de Turing pública y automatizada para diferenciar a las personas de las computadoras [https://definicion.de/captcha/], entonces tiene acceso al sistema.

2.2 Sintaxis de las proposiciones

La sintaxis es la manera o forma en que se escriben las proposiciones, particularmente las proposiciones moleculares, ya que pueden escribirse de diferente maneras, y es bueno establecer un patrón que ayude a reconocer las proposiciones que están involucradas, de modo que sea más fácil interpretarla. En el caso del lenguaje natural, la sintaxis es menos formal que en la lógica formal o en los lenguajes de programación, pero es bueno reconocer algunas formas principales para poder distinguir de otras formas que coloquialmente pudieran ser equivalentes.

2.2.1 Proposiciones opositivas

Las proposiciones formuladas con enlaces opositivos se pueden escribir de diferentes maneras, pero es fácil reconocer el tipo de una proposición con enlaces opositivos si se remueve la parte que hace negativa la proposición, se reescribe la frase como «no es verdad que...» y la frase sigue teniendo el mismo significado que la original: La proposición «El café no está caliente» es equivalente a «no es cierto que el café está caliente». También es común llamar a este enlace como la negación.

Ahora, utilizaremos el patrón ⟨_____⟩ para indicar el lugar en donde deben ser escritas las proposiciones, una proposición opositiva se puede escribir como:

$$\text{No } \langle \text{\_\_\_\_\_} \rangle$$

Ejemplo 2.6

- En la expresión «No hay agua» se puede distinguir la proposición opositiva escribiendo «no ⟨*hay agua*⟩ ».
- La expresión «Los zapatos no tienen agujetas» tiene el enlace dentro de la oración, pero se puede escribir como «no ⟨*los zapatos tienen agujetas*⟩ », aunque la lectura ya no sea tan clara ni directa en español.
- En términos matemáticos también podemos encontrar proposiciones opositivas como «$77 \neq 45$», que podemos traducir a nuestra forma como «no $\langle 77 = 45 \rangle$ ».

Hay ocasiones en que las oraciones que no son proposiciones parecen serlo. En esas ocasiones es útil tratar de escribirlas como si fuera una proposición opositiva, si al leer la nueva oración, esta tiene sentido gramatical, seguramente sí es una proposición, pero si al leerla carece de sentido gramatical, seguramente no es una proposición.

Ejemplo 2.7

En las siguientes oraciones que no son proposiciones, se usa el componente «No es verdad que» para ver claramente que no son tales:

1. «¡Antonio saca la basura!». La escribiremos como una proposición opositiva: «no es verdad que ¡Antonio saca la basura!». Notamos que gramaticalmente pierde sentido, por lo que la oración original no es una proposición.
2. «¿Mañana hay examen?». La escribiremos como una proposición opositiva: «no es cierto que ¿Mañana hay examen?». Notamos que gramaticalmente también pierde sentido, por lo que no es una proposición.

2.2.2 Proposiciones disyuntivas

Las proposiciones disyuntivas son proposiciones moleculares unidas por un enlace que genera la idea de optar por alguna de las proposiciones que la componen [atómicas o moleculares más simples]. Generalmente utilizamos la palabra «o» como enlace disyuntivo para las proposiciones, como en «Me gusta el café o el chocolate», a veces, por razones ortográficas se usa la «u» en lugar de «o», como en «hay setenta u ochenta espacios disponibles».

☞ Se cambia la «o» a la «u» si la palabra siguiente inicia con una letra con sonido /o/ u /ho/. https://www.rae.es/espanol-al-dia/cambio-de-la-o-disyuntiva-en-u-0.

Este siguiente ejemplo nos permite observar que en ocasiones las proposiciones no se encuentran directamente. Las siguientes tres proposiciones son equivalentes:

a_1. «hay setenta u ochenta espacios disponibles».
a_2. «hay setenta espacios disponibles u ochenta».
a_3. «hay setenta espacios disponibles o hay ochenta espacios disponibles».

Usando el patrón $\langle \_\_\_\_\_ \rangle$ para identificar las proposiciones, una proposición disyuntiva tiene la forma:

$$\langle \_\_\_\_\_ \rangle \text{ o } \langle \_\_\_\_\_ \rangle.$$

También es posible utilizar el términos de enlace disyuntivo «O – o»:

$$\text{«O } \langle \_\_\_\_\_ \rangle \text{ o } \langle \_\_\_\_\_ \rangle \text{ »}$$

Las palabras «o – o» forman parte del mismo enlace disyuntivo. Se puede utilizar indistintamente una u otra forma, aunque por simplicidad, escribiremos solamente «o» para unir dos proposiciones en forma disyuntiva.

Ejemplo 2.8

Las siguientes proposiciones son disyuntivas:

1. «$x = 100$ o $x = 200$». ▹ *Tiene dos proposiciones atómicas enlazadas con el término disyuntivo «o»: $\langle x = 100 \rangle$ o $\langle x = 200 \rangle$.*
2. «Arturo llega tarde o no participa». ▹ *Son dos proposiciones atómicas enlazadas con «o»: ⟨Arturo llega tarde⟩ o ⟨Arturo no participa⟩.*
3. «O caminar es buen ejercicio o nadar lo es». ▹ *Se enlazan dos proposiciones atómicas con el término disyuntivo «o – o»: O ⟨caminar es buen ejercicio⟩ o ⟨nadar es buen ejercicio⟩.*

2.2.3 Proposiciones conjuntivas

En una proposición compuesta como «El león es el rey de la selva y el búho es el animal más sabio», podemos distinguir las proposiciones encerrándolas entre corchetes triangulares como:

⟨*El león es el rey de la selva*⟩ y ⟨*el búho es el animal más sabio*⟩.

Es posible intercambiar de lugar las proposiciones respecto de la conjunción y aún la expresión está gramaticalmente bien escrita:

⟨*El búho es el animal más sabio*⟩ y ⟨*el león es el rey de la selva*⟩.

☞ Aunque al intercambiar de lugar las proposiciones, la interpretación de la proposición resultante no cambia, esto no sucede para todos los tipos de enlaces, especialmente los enlaces implicativos.

Podemos también intentar cambiando las proposiciones por otras, teniendo como resultado una proposición compuesta correctamente escrita:

⟨*Neptuno es un planeta*⟩ y ⟨*el aguacate es una fruta*⟩.

Este ejercicio nos hace ver que una proposición molecular conjuntiva tiene la forma:

⟨_____⟩ y ⟨_____⟩.

El lenguaje español nos permite utilizar otra palabras para «adornar» un término de enlace, por ejemplo las palabras «a la vez». Las palabras «a la vez» e «y», juntas forman el mismo enlace conjuntivo.

Ejemplo 2.9

- ⟨*Neptuno es un planeta o un meteorito*⟩ y ⟨*el aguacate es una fruta*⟩.
- ⟨*El usuario ingresó la clave correcta*⟩ y ⟨*resolvió el captcha*⟩.
- A la vez ⟨*llora*⟩ y ⟨*se ríe*⟩. Esta oración tiene un sujeto implícito.
- A la vez ⟨$0 < 5$⟩ y ⟨$5 < 10$⟩.

Hay otras formas gramaticales que transmiten la idea de que es obligatorio que cada proposición componente sea cierta para que la proposición completa también lo sea, he aquí un par de ejemplos: «La sopa contiene calabaza además de zanahoria», «Los recién ingresados están en el edificio A, lo mismo quienes ingresaron el año pasado».

2.2.4 Proposiciones implicativas

Son proposiciones moleculares que involucran dos proposiciones, una de ellas es tomada como un antecedente de la segunda, así la segunda es una consecuencia de la primera. Las proposiciones implicativas se reconocen porque se pueden escribir utilizando el enlace implicativo «si – entonces» con la siguiente forma:

«Si ⟨_____⟩, entonces ⟨_____⟩ ».

La proposición que se encuentra en la parte «si», se llama **antecedente**, también se conoce como «hipótesis» o «prótasis», y la proposición que se encuentra en la parte «entonces» es el **consecuente**, aunque también se nombra «tesis» o «apódosis». Son oraciones como:

1. Si ⟨*tiro el vaso*⟩, entonces ⟨*el vaso cae al piso*⟩.
2. Si ⟨*hoy amanece lloviendo*⟩, entonces ⟨*la ropa está mojada*⟩.
3. Si ⟨*come pasto*⟩, entonces ⟨*es herbívoro*⟩.
4. Si ⟨*tú te gradúas*⟩, entonces ⟨*tu conciencia está tranquila*⟩.

Otras formas en las que se puede apreciar las proposiciones condicionantes son:

- Omitiendo la palabra «entonces», como en:
  - «si ⟨*tiro el vaso*⟩, ⟨*el vaso cae al piso*⟩ ».
  - Si ⟨*hoy amanece lloviendo*⟩, ⟨*la ropa está mojada*⟩.
- Determinando la suficiencia de una acción, como en «Es suficiente si ⟨*corro 10 km*⟩ para ⟨*sudar copiosamente*⟩ »; a veces se puede escribir en sentido inverso, pero las acciones deben tener un sentido temporal como «Una condición suficiente para ⟨*sudar copiosamente*⟩ es ⟨*correr 10 km*⟩ ».
- Invirtiendo el antecedente con el consecuente, como en «⟨*Inicia la computadora*⟩, si ⟨*oprimo el botón de encendido*⟩ ».

2.3 Símbolos en expresiones lógicas

2.3.1 Símbolos en proposiciones

Las proposiciones moleculares pueden incluir tanto proposiciones moleculares más simples y proposiciones atómicas, tomemos como ejemplo la siguiente proposición implicativa que incluye todos los otros tipos:

⟨Si ⟨⟨no ⟨*come el paciente*⟩⟩ o ⟨*come comida chatarra*⟩⟩, entonces ⟨⟨*se enferma*⟩ y ⟨*muere*⟩⟩⟩.

Rápidamente es notoria la dificultad para distinguir las proposiciones, especialmente en expresiones largas. Otra situación que se presenta, es cuando hay que utilizar una misma proposición en diferentes partes de la expresión, esto requiere volver a escribir tal proposición tantas veces como se requiera.

El uso de modelos o plantillas ha sido un primer intento para utilizar símbolos en lugar de escribir las sentencias [página 37], principalmente con el fin de generalizar, pero el uso de ⟨_____⟩ no es la forma que comúnmente se utiliza. Para reducir estos problemas se sustituye cada proposición por un nombre corto que la identifique.

Para nombrar las proposiciones, se empieza por nombrar las proposiciones atómicas, generalmente utilizando los símbolos p, q, r, s, t aunque es posible utilizar cualquier otro símbolo que no se haya utilizado anteriormente, por ejemplo:

- Sea p la proposición ⟨*Ha llovido más de lo normal*⟩.
- Sea q la proposición ⟨*El día tiene más de 10 horas de luz*⟩.
- Sea r la proposición ⟨*hoy tuve problemas al llegar*⟩.

Al relacionar una proposición con un símbolo se crea una definición. Así en la frase «Sea p la proposición ⟨*Ha llovido más de lo normal*⟩ », se ha asignado el significado ⟨*Ha llovido más de lo normal*⟩ al término p. Representaremos la acción de asignar un significado a un término con el símbolo '↤'.

Así la expresión

$$p \leftarrow\!\!| \langle \textit{Ha llovido más de lo normal} \rangle,$$

debe leerse como: «El símbolo p significa $\langle \textit{Ha llovido más de lo normal} \rangle$ ».

Los símbolos empleados para representar proposiciones se llaman identificadores. Pueden ser de una sola letra, o bien utilizar una palabra corta o incluir subíndices, por ejemplo:

- $p_1 \leftarrow\!\!| \langle \textit{Ha llovido más de lo normal} \rangle$, y
- $p_2 \leftarrow\!\!| \langle \textit{El día tiene más de 10 horas de luz} \rangle$

o bien

- $prop_1 \leftarrow\!\!| \langle \textit{Ha llovido más de lo normal} \rangle$, y
- $prop_2 \leftarrow\!\!| \langle \textit{El día tiene más de 10 horas de luz} \rangle$.

Hay cuatro consideraciones importantes al utilizar el símbolo '$\leftarrow\!\!|$':

1. La expresión se debe leer e interpretar de izquierda a derecha, empezando por el identificador, esto con el fin de tener una lectura correcta.
2. Todos los identificadores al lado derecho de '$\leftarrow\!\!|$' ya deben tener una asignación previa o ser previamente conocidos.
3. Del lado izquierdo del símbolo, solamente puede haber identificadores, si hay más de uno, deben estar separados por coma; cuando hay más de un identificador, del lado derecho del símbolo debe haber la misma cantidad de proposiciones separadas por coma.
4. Solamente debe haber un símbolo de asignación en cada expresión.

Ejemplo 2.10

Los siguientes son usos válidos del símbolo de asignación.

1. $p \leftarrow\!\!| \langle \textit{El caballo es purasangre} \rangle$,
2. $p \leftarrow\!\!| \langle \textit{El caballo es digno de un rey} \rangle$.
3. $q, r \leftarrow\!\!| \langle \textit{El ajedrez es un juego interesante} \rangle$, $\langle$*'damas inglesas' es un juego de mesa*$\rangle$.

Aunque las dos asignaciones de p son válidas, utilizar el mismo símbolo una segunda vez tiene el efecto de «reescribir» el significado de la primera asignación, por lo que p tendrá el valor $\langle \textit{El caballo es digno de un rey} \rangle$, olvidando la primera.

En la tercera expresión, hay dos identificadores y dos proposiciones, de este modo q se define como $\langle \textit{El ajedrez es un juego interesante} \rangle$ y r se define como $\langle$*'damas inglesas' es un juego de mesa*$\rangle$.

Los siguientes ejemplos no son asignaciones válidas:

4. $p \leftarrow\!\!| p$.
5. p y $q \leftarrow\!\!| \langle \textit{El perro no ha comido hoy} \rangle$.
6. $p, q \leftarrow\!\!| \langle \textit{El camello toma mucha agua} \rangle$.
7. $p \leftarrow\!\!| \langle \textit{Antonio es abogado} \rangle \leftarrow\!\!| \langle \textit{Lucia es abogada} \rangle$.

El ejemplo 4 sintácticamente está bien escrito porque del lado izquierdo de '$\leftarrow\!\!|$' aparece un identificador y del lado derecho supuestamente aparece una proposición. Sin embargo, semánticamente hay un problema, porque al leer la expresión:

$$\text{«En lo sucesivo y hasta que no se indique otra cosa, } p \text{ significa } \langle p \rangle \text{ »},$$

la proposición p no tiene valor alguno, ocasionando que el identificador p tampoco tenga valor.

En el ejemplo 5, en el lado izquierdo aparece un término de enlace, lo que indica que es una proposición, esto es equivalente a tener un valor constante y no una variable, violando así la consideración 3.

El ejemplo 6 del lado izquierdo hay dos identificadores, pero del lado derecho solamente hay una proposición; esto viola la segunda parte de la consideración 3.

En el caso 7, hay más de una asignación en la misma expresión.

2.3.2 Símbolos para términos de enlace

Los enlaces lógicos también pueden ser reemplazados por símbolos. Los símbolos que comúnmente se utilizan se muestran en la siguiente tabla:

Para los ejemplos de la siguiente tabla, considera:

$p \leftarrow\!\!\shortmid \langle Hay\ internet \rangle$.
$q \leftarrow\!\!\shortmid \langle Escucho\ música\ en\ línea \rangle$.
$r \leftarrow\!\!\shortmid \langle Veo\ series\ en\ línea \rangle$.
$s \leftarrow\!\!\shortmid \langle Salgo\ a\ jugar \rangle$.

Conector lógico	Símbolo	Ejemplo
Opositivo	$\neg$	$\neg p$ se lee «No $\langle hay\ internet \rangle$ ».
Conjuntivo	$\wedge$	$p \wedge q$ se lee «$\langle hay\ internet \rangle$ y $\langle escucho\ música\ en\ línea \rangle$ ».
Disyuntivo	$\vee$	$p \vee s$ se lee «$\langle hay\ internet \rangle$ o $\langle salgo\ a\ jugar \rangle$ ».
Implicativo	$\rightarrow$	$\neg p \rightarrow r$ se lee «Si no $\langle hay\ internet \rangle$ implica que $\langle veo\ series\ en\ línea \rangle$ ».

Ejemplo 2.11

Considera la siguiente oración. Enseguida se enlistan las proposiciones atómicas o moleculares.

> «La lluvia cae suavemente sobre los campos verdes, mientras que el viento susurra entre los árboles. El aroma a tierra mojada llena el aire, indicando que ha sido una tormenta refrescante.»

- De la primera sentencia: «La lluvia cae suavemente sobre los campos verdes, mientras que el viento susurra entre los árboles», se obtienen dos proposiciones atómicas:
  $p_1 \leftarrow\!\!\shortmid \langle La\ lluvia\ cae\ suavemente\ sobre\ los\ campos\ verdes \rangle$
  $p_2 \leftarrow\!\!\shortmid \langle el\ viento\ susurra\ entre\ los\ árboles \rangle$.
  Sin embargo, el texto sugiere que al mismo tiempo que la lluvia cae, el viento susurra, esto permite tener una tercera proposición, esta vez una proposición molecular:
  $p_3 \leftarrow\!\!\shortmid p_1 \wedge p_2$.
- De la segunda sentencia: «El aroma a tierra mojada llena el aire, indicando que ha sido una tormenta refrescante.», se obtienen otras dos proposiciones atómicas:
  $p_4 \leftarrow\!\!\shortmid \langle El\ aroma\ a\ tierra\ mojada\ llena\ el\ aire \rangle$
  $p_5 \leftarrow\!\!\shortmid \langle ha\ sido\ una\ tormenta\ refrescante \rangle$.
  Esta parte del texto sugiere una proposición implicativa, pues «el aroma a tierra mojada» es una condición necesaria para determinar que «la tormenta ha sido refrescante», entonces podemos crear una nueva proposición molecular.
  $p_6 \leftarrow\!\!\shortmid p_4 \rightarrow p_5$.

Los símbolos usados matemáticas incluyen algunos que sirven para indicar una negación, usualmente se escribe un símbolo y su negación se escribe el mismo símbolo con una diagonal que lo atraviesa, observa los ejemplos en la siguiente tabla:

Proposición	Símbolo	Proposición negada	Símbolo
⟨*es igual*⟩	$=$	no ⟨*es igual*⟩	$\neq$
⟨*es mayor*⟩	$>$	no ⟨*es mayor*⟩	$\not>$
⟨*es mayor*⟩ $\vee$ ⟨*es igual*⟩	$\geq$	no ⟨⟨*es mayor*⟩ $\vee$⟨*es igual*⟩⟩	$\not\geq$

2.3.3 Símbolos de agrupamiento

En ocasiones es confuso interpretar proposiciones moleculares, especialmente cuando hay términos de enlace de diferentes tipos. Para ayudar a una adecuada interpretación, se usan símbolos de agrupamiento, principalmente de los diferentes tipos de paréntesis, '(' y ')'.

Ahora supongamos la expresión «no como carne», que es una proposición molecular opositiva de la forma «no ⟨_____⟩ » que contiene la proposición atómica $p \leftarrow\!\!|\langle$*como carne*$\rangle$, es decir hay dos proposiciones, una que contiene a la otra, lo que se puede escribir entre paréntesis:

$$(\neg(p))$$

de esta manera queda claro que hay una proposición molecular $(\neg(p))$ que es una proposición opositiva que afecta la proposición atómica p.

Consideremos ahora la proposición $p \leftarrow\!\!|\langle$*El agua está fría*$\rangle$. Observa que ahora p se ha redefinido, ya no significa ⟨*como carne*⟩ sino que ha cambiado su significado. Al obtener todas las proposiciones se logra la misma expresión $(\neg(p))$, y la lectura literal sería:

«No es cierto que el agua está fría»,

sin embargo, el uso del español permite reescribir la oración para que resulte más coloquial:

«El agua no está fría»,

pero otras interpretaciones como «el agua está caliente» o «el agua está tibia» son malas interpretaciones de la negación de la proposición atómica «el agua está fría».

El uso de los paréntesis que es descrito anteriormente es válido en todos los términos de enlace, supongamos la siguientes proposiciones atómicas:

$a \leftarrow\!\!| \langle 10 > 5\rangle$;
$b \leftarrow\!\!| \langle 10^2 > 10\rangle$;
$p \leftarrow\!\!| \langle$*Viajo en avión*$\rangle$;
$q \leftarrow\!\!| \langle$*Viajo en tren*$\rangle$;
$r \leftarrow\!\!| \langle$*Viajo en autobús*$\rangle$.

- La expresión «$10 > 5$ y $10^2 > 10$» es una expresión que es una proposición disyuntiva de la forma «⟨_____⟩ y ⟨_____⟩ » que involucra dos proposiciones atómicas y si hacemos $a \leftarrow\!\!|\langle 10 > 5\rangle$ y para la segunda hacemos $b \leftarrow\!\!|\langle 10^2 > 10\rangle$, podemos escribir

$$((a) \wedge (b)).$$

- La expresión «O viajo en avión o viajo en autobús» es una proposición molecular disyuntiva de la forma «o ⟨_____⟩ o ⟨_____⟩ », que incluye las proposiciones moleculares ⟨*viajo en avión*⟩ y ⟨*viajo en autobús*⟩. Si recordamos que $p \leftarrow\!\shortmid$⟨*viajo en avión*⟩ y $r \leftarrow\!\shortmid$⟨*viajo en autobús*⟩, tendremos:

$$((p) \vee (r)),$$

  porque hay una proposición disyuntiva que incluye dos proposiciones atómicas, primero (p) y luego (r) en ese orden.
- La expresión «no es cierto que viajo en autobús o que no viajo en tren» es una proposición molecular que incluye otras proposiciones moleculares y atómicas, veamos.
  1. «No es cierto que ⟨*viajo en autobús o que no viajo en tren*⟩ ». Es una proposición molecular opositiva que afecta a la proposición molecular ⟨*viajo en autobús o que no viajo en tren*⟩.
  2. «⟨*viajo en autobús*⟩ o ⟨*no viajo en tren*⟩ ». Es una proposición disyuntiva que incluye dos proposiciones, una es atómica y la otra es molecular opositiva.
  3. ⟨*viajo en autobús*⟩ es una proposición atómica.
  4. «No ⟨*viajo en tren*⟩ ». Es una proposición opositiva que incluye una proposición atómica.
  5. ⟨*viajo en tren*⟩ es una proposición atómica.

  Sabiendo que $p \leftarrow\!\shortmid$ ⟨*Viajo en avión*⟩ y $r \leftarrow\!\shortmid$ ⟨*Viajo en autobús*⟩, podemos escribir:

$$((\neg(r)) \vee ((\neg(p))))$$

Ejemplo 2.12

Las siguientes columnas ilustran el uso de los paréntesis en algunos ejemplos:

Ana está estudiando y no juega	(Ana está estudiando)$\wedge$($\neg$(Ana juega))
Si $x = 15$, entonces $x > 0$	$(x = 15) \rightarrow (x > 0)$.
No es cierto que $10 = 12$	$\neg(10 = 12)$, $10 \neq 12$.
No es cierto que si $x + 5 = 10$, entonces $x = 0$	$\neg((x + 5 = 10) \rightarrow (x = 0))$

2.3.4 Eliminación de algunos paréntesis

Es notable la cantidad de pares de paréntesis que se pueden llegar a escribir a la hora de representar alguna proposición molecular más o menos compleja. El uso de muchos pares de paréntesis tiene el efecto secundario de dificultar la lectura, por lo que se han tomado algunos acuerdos para reducir el número de paréntesis:

1. Los paréntesis más externos de una proposición se pueden omitir. Así la proposición (p) se puede escribir simplemente p. La proposición $((\neg(r)) \vee ((\neg(p))))$ se puede escribir $(\neg(r)) \vee ((\neg(p)))$ al omitir los paréntesis más externos.
2. Si el paréntesis involucra una solo identificador sin términos de enlace, se puede omitir. La proposición $((p))$ puede se pueden eliminar los paréntesis más internos un par de veces y terminar con p. Podemos reescribir $(\neg(r)) \vee ((\neg(p)))$ como $(\neg r) \vee ((\neg p))$, pero los paréntesis en $\neg(p \vee q)$ no se pueden omitir puesto que la negación involucra más de un identificador y un término de enlace, ocurre lo mismo con $\neg(p \wedge p)$.
3. Si hay juegos de paréntesis que encierran la misma expresión, se pueden omitir todos excepto uno. Por ejemplo en $\neg(((p \vee q)))$, para escribir $\neg(p \vee q)$.

4. Si una expresión con paréntesis involucra dos o más términos de enlace de diferente tipo, es posible eliminar algunos pares de paréntesis si es claro qué término de enlace tiene mayor jerarquía [ver la tabla 2.1 página 46].

Ejemplo 2.13

En la siguiente expresión se eliminan los paréntesis redundantes:

1. $((\neg(p)) \vee ((\neg(q)) \rightarrow (\neg(s))))$:
   a) Se puede eliminar los paréntesis más externos: $(\neg(p)) \vee ((\neg(q)) \rightarrow (\neg(s)))$.
   b) Se eliminan los paréntesis que involucran un solo identificador: $(\neg p) \vee ((\neg q) \rightarrow (\neg s))$.

2.3.5 Jerarquía de términos de enlace

La «jerarquía de los términos de enlace» se refiere al orden en que los términos de enlace deben ser interpretados para obtener una lectura estandarizada. Considera ahora la siguiente proposición molecular, que en principio se ha escrito sin paréntesis:

$$p \vee q \wedge r$$

Se pueden establecer dos posibles interpretaciones dependiendo de qué termino de enlace se interprete primero. Al colocar paréntesis alrededor de aquella que debe ser interpretada primero, tenemos las siguientes dos formas:

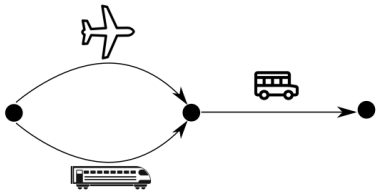
$$(p \vee q) \wedge r \quad \text{o bien} \quad p \vee (q \wedge r)$$

En general la interpretación no es la misma, para tener una idea mas clara de las distintas interpretaciones, observa cada figura y el contenido de las proposiciones:

Considera las proposiciones:

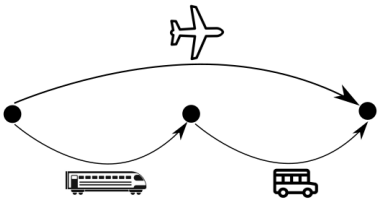
$p \leftarrow\!\!\shortmid \langle \textit{Viajo en avión} \rangle$.
$q \leftarrow\!\!\shortmid \langle \textit{Viajo en tren} \rangle$.
$r \leftarrow\!\!\shortmid \langle \textit{Viajo en autobús} \rangle$.

1. $(p \vee q) \wedge r$ se interpreta como:
   $(\langle \textit{Viajo en avión} \rangle$ o $\langle \textit{viajo en tren} \rangle)$ y $\langle \textit{viajo en autobús} \rangle$.



La interpretación gráfica es que la primera parte del trayecto se puede hacer de dos formas: por avión o por tren y la segunda parte se hace en autobús.

2. $p \vee (q \wedge r)$ se interpreta como:
   $\langle \textit{Viajo en avión} \rangle$ o $(\langle \textit{viajo en tren} \rangle$ y $\langle \textit{viajo en autobús} \rangle)$.



En esta segunda forma, el viaje se puede realizar de dos maneras: una por avión y la segunda en dos etapas, empezando por tren y terminando en autobús.

Debido a esta ambigüedad a la hora de interpretar las expresiones lógicas, se ha establecido una jerarquía de términos de enlace [Arn89, Ros04], que se muestra en la tabla 2.1.

Nivel	Tipo	Símbolo
0	Operadores de agrupamiento	(,)
1	Operadores opositivos	$\neg$
2	Operadores conjuntivos	$\wedge$
3	Operadores disyuntivos	$\vee$
4	Operadores condicionantes	$\rightarrow$

Tabla 2.1: *Jerarquía de términos de enlace.*

Siguiendo la norma de la tabla 2.1, la expresión $p \vee q \wedge r$, debe ser interpretada como $p \vee (q \wedge r)$ y ya no son necesarios los paréntesis. La jerarquía de operadores permite distinguir qué término de enlace debe ser interpretado primero.

Ejemplo 2.14

Digamos que $p \vee q \rightarrow \neg q \vee r \wedge \neg s$ es una proposición [el significado de cada literal por el momento no es relevante]. ¿Qué orden debe tener las operaciones para alcanzar una correcta interpretación? Colocaremos los paréntesis encerrando las proposiciones que se deben interpretar en primer lugar de acuerdo a la tabla 2.1:

1. Los enlaces opositivos tienen la prioridad más alta:
   $p \vee q \rightarrow \big(\neg q\big) \vee r \wedge \big(\neg s\big)$
2. Los enlaces disyuntivos tienen la siguiente prioridad más alta:
   $p \vee q \rightarrow (\neg q) \vee \big(r \wedge (\neg s)\big)$
3. Los enlaces disyuntivos tienen la siguiente prioridad:
   $\big(p \vee q\big) \rightarrow \big((\neg q) \vee (r \wedge (\neg s))\big)$
4. Los enlaces implicativos tienen la prioridad mas baja:
   $\big((p \vee q) \rightarrow ((\neg q) \vee (r \wedge (\neg s)))\big)$

Por regla general siempre se omiten los paréntesis más externos, así la expresión final será $(p \vee q) \rightarrow ((\neg q) \vee (r \wedge (\neg s)))$.

Los paréntesis en el ejemplo 2.14 se han colocado de acuerdo a la norma establecida en la tabla 2.1 porque en un inicio no tenía ninguno y hay ambigüedad en la prioridad de las operaciones, pero si una expresión ya cuenta con paréntesis, estos deben ser respetados y colocar solamente aquellos en los que aún exista ambigüedad.

2.4 Fórmulas bien formadas

Ya hemos utilizado palabras escritas con símbolos, como $((p \vee q) \rightarrow ((\neg(q \vee r)) \wedge (\neg s)))$ en el ejemplo anterior. Las palabras construidas de esta manera, las llamaremos fórmulas, o expresiones. A veces se pueden comenter errores al escribir las fórmulas o simplemente están mal escritas. Cuando una fórmula se ha escrito correctamente, siguiendo todas las reglas sintácticas y semánticas del lenguaje, decimos que la

fórmula es una fórmula bien formada (*fbf*). Afortunadamente hay manera para determinar si una fórmula es *fbf*.

2.4.1 Construcción de *fbf*

Las fórmulas bien formadas [*fbf*] son expresiones proposicionales que se escriben de acuerdo con las siguientes reglas [Chu32, CG15]:

1. **F**, **V** son *fbf*s
2. Un término literal es una *fbf*.
3. Si p es una *fbf*, entonces $\neg p$ es una *fbf*.
4. Si p y q son *fbf*, entonces $(p \vee q)$ es una *fbf*.
5. Una palabra es una *fbf* si resulta de un número finito de aplicaciones de las reglas 1, 2 y 3.

☞ Aunque el punto 4 indica que solamente $p \vee q$ es una *fbf*, cualquier otra expresión que incluya otros términos de enlace se pueden escribir utilizando solamente la negación ['¬'] y la disyunción ['∨'], por ejemplo $p \wedge q$ es equivalente a $(\neg(\neg p \vee \neg q))$. Así el punto 4 establece que $(p \wedge q)$, $(p \rightarrow q)$ y todas las demás operaciones booleanas de dos proposiciones [ver sección 5.3] son *fbf*s.

Es común que se omitan los paréntesis de acuerdo a los criterios descritos en la sección 2.3.5 [página 45].

Ejemplo 2.15

Se puede construir una *fbf* agregando términos proposicionales y de enlace:

1. p es una *fbf* por la regla 1.
2. q es una *fbf* por la regla 1.
3. $(p \vee q)$ es una *fbf* por la regla 3.
4. $\neg(p \vee q)$ es una *fbf* por la regla 2 usando la *fbf* anterior.
5. $(q \wedge p)$ es una *fbf*.
6. $((q \wedge p) \rightarrow \neg(p \vee q))$ es la *fbf* final. Los siguientes pasos reducen la expresión al omitir algunos paréntesis.
7. $(q \wedge p) \rightarrow \neg(p \vee q)$ al omitir los paréntesis más externos.
8. $q \wedge p \rightarrow \neg(p \vee q)$ es una *fbf* al omitir los paréntesis de la conjunción, ya que la jerarquía de términos de enlace obliga a interpretar primero la conjunción y después la implicación. No es posible eliminar los paréntesis de la disyunción, esto debido a que la negación afecta la toda expresión disyuntiva, $\neg p \vee q$ se debe interpretar $((\neg p) \vee q)$, que es diferente a lo que se tiene.

2.4.2 Estrategia de árbol para determinar una *fbf*

Determinar si una expresión simbólica es una *fbf* en ocasiones puede resultar difícil, principalmente cuando la expresión incluye muchos términos de enlace, muchos paréntesis o variables proposicionales. Se puede determinar si una expresión simbólica es una *fbf* siguiendo una estructura de tipo árbol que inicia con una *fbf* con los paréntesis balanceados.

Paréntesis balanceados

Para saber si los paréntesis están balanceados y además saber cuál es el enlace principal de una expresión proposicional simbólica, podemos seguir el siguiente

método, para lo que se requiere una expresión simbólica con todos los paréntesis, incluyendo el más externo.

1. Si es una literal, entonces no tiene enlace alguno.
2. Si la expresión empieza con una negación, este es el enlace principal.
3. Si la expresión empieza con paréntesis, se enumeran empezando con 1 los paréntesis, en orden creciente si abren y en orden decreciente si cierran. Si la cuenta de paréntesis termina en 0, entonces hay buen balance y el enlace principal se encuentra a la derecha del segundo paréntesis numerado con 1.

Ejemplo 2.16

Considera la expresión $(((q \wedge \neg p) \vee \neg r) \rightarrow \neg(q \wedge r))$. Siguiendo el criterio de paréntesis balanceados [ver página 48], se usa el inciso 3, porque la expresión inicia en paréntesis. Se enumeran los paréntesis en orden creciente si abren y en orden decreciente si cierran:

$$\underset{123}{(}\underset{2}{(}\underset{1}{(}q \wedge \neg p\underset{2}{)} \vee \neg r\underset{1}{)} \rightarrow \neg\underset{2}{(}q \wedge r\underset{10}{)}\underset{0}{)}$$

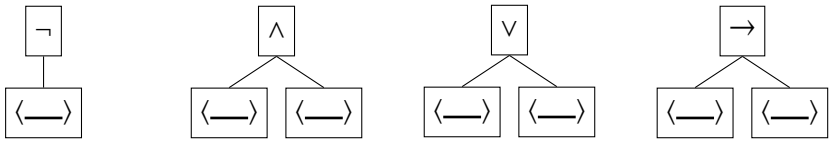
1. Como la cuenta termina en 0, los paréntesis están balanceados.
2. El enlace principal se encuentra a la derecha del segundo paréntesis con cuenta 1, este es un $\rightarrow$, por lo que la expresión es una proposición implicativa.

Construcción del árbol de una *fbf*

La representación de árbol se utiliza comúnmente en lógica booleana y programación para evaluar y simplificar expresiones lógicas de manera eficiente.

Se utiliza una estructura que organiza y muestra la relación jerárquica de los elementos de una expresión lógica, como los términos de enlace [y, o, si–entonces, no] y las proposiciones relacionadas. Esta estructura se asemeja a un árbol que crece de arriba hacia abajo, con el punto de inicio en la raíz, que representa la operación principal y cada rama que se deriva de la raíz representa una subexpresión escrita con los símbolos variables involucrados en la expresión lógica.

La estrategia es escribir gráficamente los patrones de construcción de las proposiciones moleculares de la forma «$\langle___\rangle$ENL$\langle___\rangle$ », donde ENL es un símbolo que representa un enlace, ya sea $\wedge$, $\vee$ o bien $\rightarrow$; o el patrón «$\neg\langle\_\_\_\rangle$ » si es una negación. La manera es la que se muestra en la siguiente figura:



(a) *Arbol de una expresión opositiva.* ***(b)*** *Estructura de árbol para una conjunción.* ***(c)*** *Árbol para una disyunción.* ***(d)*** *Estructura de árbol para una implicación.*

Figura 2.1: *Estructuras de árbol para los diferentes términos de enlace. Los patrones $\langle\_\_\_\rangle$ deben ser sustituidos por estructuras de tipo árbol.*

La manera de determinar si una expresión simbólica es una *fbf*, utilizando esta estructura de tipo árbol, es empezar con el paréntesis más externo que encierra toda

la expresión [si no hay, se debe colocar uno], y escribir la expresión en la forma de su estructura de tipo árbol que corresponde.

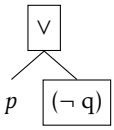
Continuar con los siguientes paréntesis, identificando qué proposiciones hay a cada lado del término de enlace. Seguir así hasta llegar a las proposiciones atómicas.

Si en algún momento no se puede continuar, antes de terminar con todas las proposiciones atómicas, entonces no es una *fbf*.

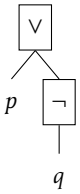
Ejemplo 2.17

Determinar si $p \vee \neg q$ es una *fbf*.

1. Se colocan los paréntesis de acuerdo a los criterios [sección 2.3.4, p. 44]: $(p \vee (\neg q))$.
2. Se escribe la estructura para el paréntesis más externo.



3. Se escribe la estructura para el siguiente paréntesis:



Es fácil observar que las siguientes expresiones simbólicas no son *fbf*, esto es porque son secuencias relativamente cortas y se debe notar que no cumplen con la sintaxis de los términos de enlace:

- $p\neg \vee q$.
  ▹ *El enlace $\neg$ está fuera de lugar. $p \vee q$ es correcto; $\neg p \vee q$ es correcto, incluso $p \vee \neg q$ es correcto.*
- $p(\rightarrow q)$.
  ▹ *Los paréntesis deben encerrar una fbf, pero $\rightarrow q$ no es una fbf porque falta una proposición a la izquierda del enlace.*
- $(r \vee t)\rightarrow(p \vee q))$.
  ▹ *Los paréntesis están desbalanceados.*

Ejercicios

1. Marca con una ✗ aquellas oraciones que sean proposiciones, ya sea una proposición simples o compuesta.
   a) (✗) *Ejemplo.* «Mi perro ladra».
   b) ◯ «¿Cómo te llamas?».
   c) ◯ «El respeto al derecho ajeno es la paz».
   d) ◯ «¡Qué caro está todo!»
   e) ◯ «No sé en dónde está tu bolso».
   f) ◯ «$30 \geq 9$».
   g) ◯ «Si estudias, entonces aprendes».
   h) ◯ «La edad de Antonio es mayor de 18».
   i) ◯ «¡Ponte bloqueador solar, o te vas a enfermar!».
   j) ◯ «Joaquín está en la biblioteca o jugando basquetbol».
   k) ◯ «Susana no quiere salir o tiene mucha tarea».
2. En las siguientes proposiciones moleculares, escribe las proposiciones atómicas entre corchetes angulares y encierra en círculo cada término de enlace.
   a) *Ejemplo.* «La computadora está mal configurada y no funciona».
   ⟨*La computadora está mal configurada*⟩ (y) (no) ⟨*la computadora funciona*⟩.
   b) «Ella canta bien y toca la guitarra».
   c) «Si pulsa la letra "n", entonces se cancela la operación».
   d) «A menos que la variable contador tenga el valor de 10, la suma se incrementa en 10».
   e) «Mi ropa preferida es de color azul, pero cuando el día es soleado».
   f) «Si $x = 20$, entonces $x >= 0$ y $x < 50$».
   g) «No es un triángulo y tampoco es un rectángulo».
   h) «$x^2 \geq 25$ o $x^2 > 100$, cuando $x \geq 5$».
   i) «Si las rectas no se intersectan, entonces no se forma ángulo alguno».
3. Clasifica las siguientes oraciones de acuerdo al tipo, ya sean oraciones declarativas, interrogativas, exclamativas o imperativas.
   a) *Ejemplo.* «¿Qué te hace sonreír incluso en los días más difíciles?»
   Interrogativa
   b) «El sol brilla intensamente en el cielo despejado.»
   c) «Mi hermana mayor estudia medicina en la universidad.»
   d) «¿Qué hora es ahora mismo?»
   e) «¿Cómo llegaste hasta aquí sin ayuda?»
   f) «¡Qué maravillosa es la naturaleza en primavera!»
   g) «¡No puedo creer lo rápido que creciste!»
   h) «Cierra la puerta antes de salir, por favor».
   i) «¡Prepara todo para la fiesta esta noche!»
4. En las siguientes oraciones declarativas, encuentra todos los términos de enlace:
   a) *Ejemplo.* «El éxito requiere tanto esfuerzo como persistencia.»
   El término de enlace es el conjuntivo, pues de acuerdo con la oración, para tener éxito se requiere esfuerzo y persistencia.
   b) «Juan no puede decidir si quedarse en casa o salir a divertirse.»

c) «El equipo de fútbol debe ganar o enfrentará la eliminación.»
d) «Puedes escoger entre chocolate o vainilla para el postre.»
e) «El concierto será el viernes o el sábado, tú decides a cuál asistir.»
f) «María estudia y trabaja para mantenerse independiente.»
g) «Para aprobar el examen, necesitas concentración y dedicación.»
h) «Si llueve, lleva un paraguas para no mojarte.»
i) «Si no estudias, es probable que no apruebes el curso.»

5. En las siguientes proposicionales indica con un círculo el término de enlace conjuntivo.
   a) *Ejemplo.* «$x = y$ y $y - 2 > y$»
   $x = y$ (y) $y - 2 > y$
   b) «Si $x > y$, entonces $y + 2 = x^2$ y $x \not> 0$»
   c) «$x > 3$ y $y < 10 + y$ y $y + z + x \geq y$»
   d) «Si $x \neq y$ y $y \neq x$, entonces $y + x > 0$ y $y - x = 0$».
6. Reescribe las siguientes declaraciones conjuntivas de modo que sea evidente el enlace conjuntivo.
   a) «El cielo despejado y las estrellas brillantes crean un paisaje nocturno hermoso.»
   Se puede reescribir como la conjunción de dos oraciones:
   1) «El cielo despejado crea un paisaje nocturno hermoso.»
   2) «Las estrellas brillantes crean un paisaje nocturno hermoso.»
   Haciendo: «El cielo despejado crea un paisaje nocturno hermoso (y) las estrellas brillantes crean un paisaje nocturno hermoso.»
   b) «Mi abuela cocina platos deliciosos y tradicionales para las celebraciones familiares.»
   c) «La música suave y el aroma a café llenan la atmósfera acogedora de este café.»
   d) «Los niños juegan felices en el parque bajo la luz del sol.»
   e) «Las olas rompen suavemente en la orilla, creando un sonido relajante en la playa.»
7. En los siguientes párrafos, descubre las proposiciones atómicas y asocia una variable con cada proposición diferente. Es posible que sea necesario reescribir algunas partes del texto.
   a) *Ejemplo.* «En el campo de la computación, la lógica es esencial para el diseño de algoritmos eficientes. Usando proposiciones lógicas, se construyen estructuras de control. Los ciclos y condicionales son estructuras de control que determinan el flujo de instrucciones.»
   $p_1 \leftrightarrow \langle$*La lógica es esencial para el diseño de algoritmos eficientes*$\rangle$.
   $p_2 \leftrightarrow \langle$*Usando proposiciones lógicas, se construyen estructuras de control*$\rangle$.
   $p_3 \leftrightarrow \langle$*Los ciclos son estructuras de control*$\rangle$.
   $p_4 \leftrightarrow \langle$*Las condicionales son estructuras de control*$\rangle$.
   $p_5 \leftrightarrow \langle$*Las estructuras de control determinan el flujo de instrucciones*$\rangle$.

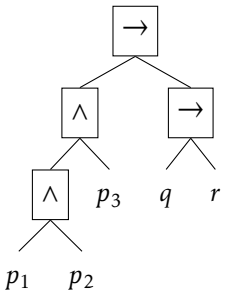
   b) «Las matemáticas son fundamentales en la lógica y el razonamiento deductivo. Mediante proposiciones lógicas, los matemáticos establecen teoremas y prueban afirmaciones basadas en axiomas y definiciones.»

c) «En la vida cotidiana, la lógica juega un papel crucial en la toma de decisiones. Al evaluar opciones, consideramos proposiciones lógicas como «si hago X, entonces Y ocurrirá». Además, razonamos con argumentos basados en premisas y conclusiones, lo que nos permite tomar decisiones informadas.»

8. Reescribe las siguientes proposiciones moleculares utilizando símbolos. Primero descubre las proposiciones asignándolas a una variable, luego construye la *fbf*. Considera colocar paréntesis si crees que ayuda a evitar confusiones.
   a) `Ejemplo.` «Si el participante ganó o es el único participante, entonces el premio es una computadora y además no paga la mensualidad por un año».
   Primero se obtienen las proposiciones:
   $p \longleftrightarrow \langle$*El participante ganó*$\rangle$.
   $q \longleftrightarrow \langle$*Es el único participante*$\rangle$.
   $r \longleftrightarrow \langle$*El premio es una computadora*$\rangle$.
   $s \longleftrightarrow \langle$*Paga la mensualidad por un año*$\rangle$.
   Así el párrafo anterior se reescribe simbólicamente como: $(p \vee q) \rightarrow (r \wedge \neg s)$.
   b) «Si el animal tiene 6 patas y brinca, entonces o es un saltamontes o es un grillo»
   c) «Si trabajas o estudias y trabajas, entonces tendrás dinero».
   d) «La sopa está sabrosa siempre que haya ingredientes frescos, si éstos se consiguen».
9. Convierte cada enunciado en una *fbf*.
   a) `Ejemplo.` «Si Luis, Fernanda y Ana tienen hambre, entonces si la comida está lista, entonces se sirve la cena.»
      1) Colocamos paréntesis en la proposición original para dividir la expresión en proposiciones menos complejas.
      «Si $\langle$*Luis, Fernanda y Ana tienen hambre*$\rangle$, entonces $\langle$*si* $\langle$*la comida está lista*$\rangle$, *entonces* $\langle$*se sirve la cena*$\rangle\rangle$.»
      2) Observamos que $\langle$*Luis, Fernanda y Ana tienen hambre*$\rangle$ es una conjunción de tres proposiciones:
      $\langle\langle$*Luis tiene hambre*$\rangle$ *y* $\langle$*Fernanda tiene hambre*$\rangle$ *y* $\langle$*Ana tiene hambre*$\rangle\rangle$
      3) Las proposiciones conjuntivas solamente tienen dos proposiciones, por lo que separamos la expresión:
      $\langle\langle\langle$*Luis tiene hambre*$\rangle$ *y* $\langle$*Fernanda tiene hambre*$\rangle\rangle$ *y* $\langle$*Ana tiene hambre*$\rangle\rangle$
      4) Descubrimos todas las proposiciones atómicas:
         $p_1 \longleftrightarrow \langle$*Luis tiene hambre*$\rangle$.
         $p_2 \longleftrightarrow \langle$*Fernanda tiene hambre*$\rangle$.
         $p_3 \longleftrightarrow \langle$*Ana tiene hambre*$\rangle$.
         $q \longleftrightarrow \langle$*La comida está lista*$\rangle$.
         $r \longleftrightarrow \langle$*Se sirve la cena*$\rangle$.
      5) Sustituimos las proposiciones moleculares por su respectiva variable proposicional:
      Si $((p_1$ y $p_2)$ y $p_3)$, entonces (si q, entonces r)
      6) Se sustituyen las palabras que se refieren a términos de enlace:

$$(((p_1 \wedge p_2) \wedge p_3) \rightarrow (q \rightarrow r))$$

7) Se elabora la estructura de tipo árbol de la expresión:



b) «Si estudias con dedicación y te enfocas en tus tareas, entonces aprobarás el examen, siempre y cuando te preparas con anticipación y no te distraes en otras actividades».

c) «Si como alimentos nutritivos, camino media hora al día y descanso lo suficiente son condiciones para tener una buena salud, para rendir en el trabajo y para ser feliz».

d) «Si optimizamos adecuadamente el algoritmo de búsqueda y utilizamos técnicas de indexación eficientes, conseguiremos reducir significativamente el tiempo de respuesta del motor de búsqueda, siempre y cuando la carga en el servidor no supere su capacidad máxima».

10. Las siguientes proposiciones involucran símbolos matemáticos y variables numéricas. Coloca los paréntesis en el lugar correcto de acuerdo a la jerarquía de operaciones de la tabla 2.1 en la página 46.

a) *Ejemplo.* $x+y=0 \wedge z>0 \vee z \neq w \rightarrow y+x<2z$

1) Los enlaces opositivos:
$x+y=0 \wedge z>0 \vee \neg(z=w) \rightarrow y+x<2z$

2) Los enlaces conjuntivos:
$(x+y=0 \wedge z>0) \vee \neg(z=w) \rightarrow y+x<2z$

3) Los enlaces disyuntivos:
$((x+y=0 \wedge z>0) \vee \neg(z=w)) \rightarrow y+x<2z$

4) Los enlaces implicativos:
$(((x+y=0 \wedge z>0) \vee \neg(z=w)) \rightarrow y+x<2z)$

Al omitir los paréntesis más externos se tiene:
$((x+y=0 \wedge z>0) \vee \neg(z=w)) \rightarrow y+x<2z$

b) $x=y \wedge z=w \vee w>x \rightarrow \neg x<z \vee y=w$

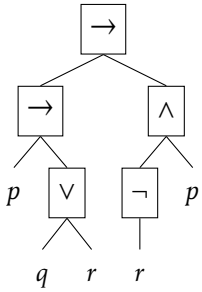
c) $x+y=0 \wedge x \neq 0 \rightarrow 2x>z \wedge \neg(x+y=z \vee x=0)$

d) $3x>12 \wedge y^2=100 \wedge 2x+z>y \rightarrow x<10 \wedge y>50$

11. Determina si las siguientes expresiones son *fbf*. Para esto, sigue el método de la página 2.4.2 y la estrategia de árbol [ver la página 48]:

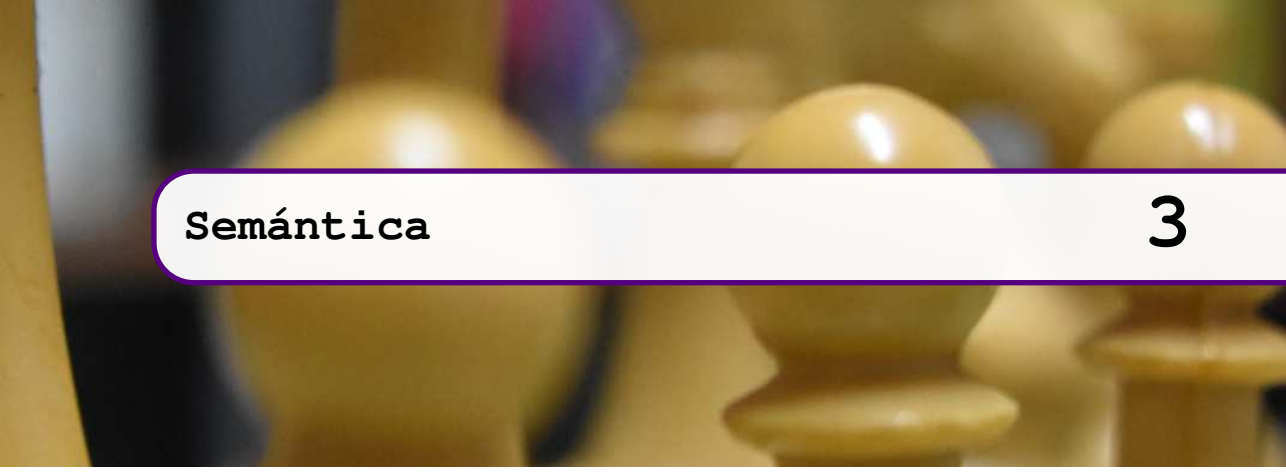
a) *Ejemplo.* $(p \rightarrow (q \vee r)) \rightarrow (\neg r \wedge p)$

- $\underset{12}{(}\underset{3}{(}p \rightarrow \underset{21}{(}q \vee r\underset{2}{)}\underset{10}{)} \rightarrow \underset{2}{(}\neg r \wedge p\underset{10}{)}$, tiene paréntesis balanceados.
- La estructura de árbol es:



- Sí, es una *fbf*.

b) $(\neg((p \wedge q) \vee (r \rightarrow s)))$
c) $\neg(p \rightarrow \vee\, q \wedge rs)$
d) $(((p \rightarrow q) \rightarrow (r \vee \neg s)) \wedge ((p \vee r) \rightarrow (q \wedge \neg s)))$
e) $(p \rightarrow q \rightarrow r \rightarrow s)$



Semántica 3

3.1 Principios fundamentales de la lógica

Hasta ahora conocemos cómo escribir las proposiciones, ya sean proposiciones atómicas o proposiciones moleculares. En este capítulo estudiaremos cómo interpretar las proposiciones, especialmente las proposiciones moleculares. Se ha mencionado que las proposiciones son oraciones que pueden ser ciertas o falsas [ver la sección 2.1.1 en la página 34] y el objetivo de la lógica formal es determinar la certeza o falsedad de una proposición correctamente escrita. Para lograrlo, es necesario establecer de antemano un conjunto de reglas de interpretación que deben ser utilizadas. En lo sucesivo, una «proposición» se referirá a una *fbf* ya sea atómica o molecular y el significado asociado a los símbolos proposicionales utilizados quedará en un segundo plano, ya que lo importante ahora es establecer un método para establecer el valor de verdad de la expresión.

3.1.1 Identidad

La ley de la identidad es el principio fundamental que establece que cualquier objeto es idéntico a sí mismo. Es una de las leyes de la lógica clásica y se formula como ⟨p es p⟩ o ⟨si p, entonces p⟩.

Establece que una proposición siempre es verdadera cuando se refiere a una sola cosa o a un individuo en particular. Por ejemplo:

- ⟨*El sol* es *el sol*⟩
- ⟨Si *llueve*, entonces *llueve*⟩
- ⟨*Gano la carrera* si *gano la carrera*⟩

Aunque pudiera parecer trivial, es esencial para el razonamiento lógico y la construcción de argumentos válidos. Sin ella, la coherencia en el pensamiento lógico se vería comprometida, ya que no podríamos afirmar con certeza la identidad de algo consigo mismo.

Sin embargo, computacionalmente y en particular en la programación de computadoras, hay tres tipos de identidad:

- **Identidad**
  - **Identidad nominal**: *Dos objetos son en realidad el mismo si tienen el mismo identificador p. ej. siendo* `a = 10`, `a == a` *significa que en este marco de creencia, es el mismo objeto.*
  - **Identidad esencial**: *Dos objetos se pueden considerar iguales si tienen asociado el mismo valor p. ej. si* `a = 10`, `b = 10`, *entonces* `a == b` *significa también que ambos objetos, aunque se llamen diferente, son el mismo.*
  - **Identidad material**: *Dos objetos son en realidad el mismo si se refieren al contenido de una misma localidad de memoria [figura 3.1], significa que aún cuando se llamen diferente, se ubiquen en lugares diferentes, pero se refieren al mismo objeto, entonces son iguales.*

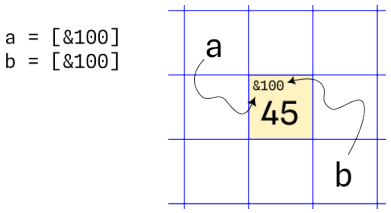


Figura 3.1: *Identidad material en programación de computadoras. Dos objetos se refieren a la misma localidad de memoria.*

Los tres conceptos de identidad se aplican en programación de computadoras de acuerdo a las necesidades y propósitos particulares de cada programa. Los primeros dos son más frecuentemente utilizados, y para los fines de este libro, se utilizarán solamente los primeros dos tipos de identidad, la nominal y la escencial.

El principio lógico de identidad es muy importante para determinar si se ha alcanzado un objetivo en particular. Sin este principio, un razonamiento pudiera dar lugar a una conclusión que no podríamos determinar si es el objetivo buscado.

3.1.2 Principio del tercero excluido

El principio del tercero excluido también se conoce como «el principio de la bivalencia»[TMRP00, p. 47] y afirma que una proposición solamente puede ser evaluada con exactamente uno de los dos posibles valores de verdad.

Los valores de verdad, también conocidos como valores lógicos o valores booleanos, son conceptos fundamentales en lógica y filosofía que se utilizan para evaluar la veracidad o falsedad de una proposición.

En la lógica computacional clásica existen únicamente dos valores de verdad:

Verdadero: Representa la situación en la que una proposición es cierta o está de acuerdo con la realidad construida. En computación, todas las proposiciones asignadas a variables son consideradas verdaderas, a menos que en el curso del programa cambien su valor. Que una proposición sea verdadera [computacionalmente], significa que existe en la lista de variables y ya se tiene un valor de verdad asociado, por ejemplo `el_perro_vuela = True` significa que la variable `el_perro_vuela` tiene asociado el valor `True`, esto no significa que en la realidad un perro vuele, sino que dentro del ámbito del programa, un identificador con ese nombre tiene el valor `True`. Los símbolos que usualmente se utilizan para referirse a esta categoría de verdad es $\top$, **1** y **V**.

Falso: Representa la situación contraria a la categoría «verdadero». En computación se puede crear una variable y asociarle el valor `False`, con esto se crea una nueva parte de la realidad construida, por ejemplo `el_perro_vuela = False` es una instrucción en donde se asocia `False` a la variable `el_perro_vuela`. Esta expresión tiene el mismo sentido que `p = False`, sin embargo, para que el programa funcione correctamente, se debe suponer que la variable `el_perro_vuela` es interpretada como `False`, a menos que se indique otra cosa. Los símbolos que usualmente se utilizan para referirse a esta categoría de verdad es $\bot$, **0** y **F**.

☞ Los valores de verdad «verdadero» y «falso» se llaman «booleanos» en honor a George Boole, un matemático y lógico del siglo XIX. Boole desarrolló un sistema algebraico en el que utilizó el número `1` para representar el valor verdadero y el número `0` para representar el valor falso [Boo48]. Así, el término «valores booleanos» se usa ampliamente en programación para referirse a los valores de verdad en lógica booleana.
En otros sistemas lógicos como en la lógica *k*-valente o también en la lógica difusa hay otros valores de verdad, pero son extensiones de este conjunto fundamental de los dos valores de verdad.

En algunos sistemas lógicos, también se pueden considerar valores adicionales, como «indefinido» o «desconocido», pero los valores de verdad clásicos son verdadero y falso.

Computacionalmente se dice que los valores booleanos forman un sistema binario porque se refieren a dos categorías excluyentes una de la otra, de modo que cualquier conjunto de categorías mutuamente excluyentes puede servir también como valores booleanos, por ejemplo los que se muestran en la tabla 3.1

Sin embargo, como lo menciona Bochensky en [Boc68, p. 62] «Lo verdadero y lo falso, en efecto, no se hallan en las cosas, algo así como si el bien fuese verdadero y el mal falso, sino en el pensamiento», las proposiciones que trataremos en este libro están sujetas al valor de verdad que se les asigna, no al que inherentemente podrían tener de acuerdo a cierto marco de creencias.

Verdadero	Falso
True	False
1	**0**
$\top$	$\bot$
V	**F**
Cierto	Falso
Arriba	Abajo
Encendido	Apagado
⎍	⎓
	

Tabla 3.1: *Sistemas binarios asociados con los valores booleanos.*

Verdad formal y verdad real

En la interpretación de las expresiones lógicas intervienen dos conceptos muy relacionados pero que en ocasiones pueden no coincidir.

La verdad «formal» se determina por el valor de verdad asignado a cada variable proposicional, junto con los enlaces que las unen para formar la *fbf*.

La verdad «real» se determina por el conocimiento común dentro de una realidad.

Ejemplo 3.1

La proposición p será **F** si $\neg p$ es **V**. Esta es la verdad formal, porque no importa qué proposición esté relacionada con p, sino que el valor se determina solamente por las reglas de inferencia de la lógica formal.

Por su parte, si $p \leftrightarrow \langle$*La constante de gravitación universal en la Tierra es de 18.67 m/s*$^2\rangle$, entonces p es **F**, y este es su valor de verdad real, puesto que obedece a un marco de creencias.

3.1.3 Principio de la no contradicción

El principio de la no contradicción establece que una proposición nunca puede ser **V** y **F** al mismo tiempo bajo las mismas circunstancias.

En su forma más simple, el principio de no contradicción afirma que una proposición y su negación no pueden ser ambas verdaderas simultáneamente. Por ejemplo, consideremos la siguiente proposición «El trabajador tiene ingreso menor a 10000 mensual», lo que podemos escribir:

$$\langle \texttt{ingreso\_mensual\_de\_trabajador} \leq 10000 \rangle$$

Si la aseveración es verdadera, entonces la negación, es decir, $\langle$*El trabajador no tiene ingreso menor a 10000 mensual*$\rangle$ sería falsa. Por el contrario, si la negación fuera verdadera, entonces la afirmación sería falsa. En ningún caso ambos enunciados pueden ser verdaderos al mismo tiempo.

Gráficamente podemos representar esta situación con el diagrama de la figura 3.2, donde se tiene que tomar una decisión considerando el monto de ingreso de un trabajador. Si es cierto que el monto es menor o igual a 10000, entonces se le aumenta el 50 %; pero si no es cierto, entonces el aumento será solo del 20 %. Sin el principio de

la no contradicción, no se podría tomar una decisión de esta clase. En la programación de computadoras en donde la toma de decisión es importante, el principio de la no contradicción es fundamental.

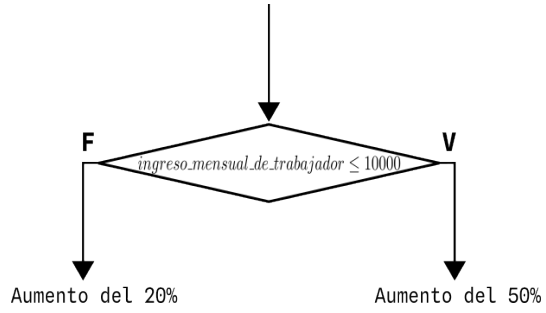


Figura 3.2: *Principio de la no contradicción: Si una proposición es* ***V****, no puede ser* ***F****, pero si es* ***F****, entonces no puede ser* ***V****.*

Este principio también es crucial en la lógica y el razonamiento porque es la base para la consistencia y coherencia del pensamiento. Sin el principio de no contradicción, no podríamos distinguir entre lo verdadero y lo falso, lo que llevaría a una situación en la que cualquier afirmación y su negación podrían ser igualmente válidas, lo que haría imposible llegar a conclusiones coherentes o tomar decisiones racionales.

3.1.4 El proceso de inferencia

En lógica computacional llamaremos «reglas de inferencia» a las instrucciones que permiten generar conclusiones válidas a partir de proposiciones preestablecidas. Estas instrucciones establecen las formas correctas de razonamiento que garantizan que, si las proposiciones son verdaderas, la conclusión también lo será.

Una proposición preestablecida es una proposición cuyo valor de verdad ya se conoce y no hay ni duda ni cuestionamiento acerca de su valor, se le conoce como «premisa». Computacionalmente representa un requerimiento de «entrada» a un procedimiento algorítmico.

La proposición obtenida por el uso de reglas de inferencia se llama «conclusión», y el proceso de convertir premisas en conclusiones se llama «deducción». Si a partir de un conjunto de premisas se deduce una conclusión, esa conclusión es una «consecuencia lógica» de las premisas. Computacionalmente la deducción es el «procesamiento», la conclusión es la «salida» y la deducción es el procedimiento que transforma las entradas en la salida del algoritmo.

El proceso de razonamiento formal es relativamente simple:

1. Se inicia con un conjunto de premisas.
2. Se elige una regla de inferencia que pueda ser satisfecha con algunas premisas actuales y se deduce una conclusión.
3. Si se alcanzó la conclusión deseada o no hay manera de alcanzarla, el proceso termina; de lo contrario, se agrega la conclusión al conjunto de premisas y se repite el proceso.

Si las premisas son verdaderas y se utilizan las reglas de inferencia correctamente, las conclusiones obtenidas también son verdaderas, porque las reglas de inferencia solamente producen proposiciones verdaderas.

Con inferencia se obtienen conclusiones basadas en las premisas con métodos que pueden ser inductivos o deductivos, por lo que tenemos ahora dos tipos de inferencia.

Inferencia
- I. inductiva: *Se alcanza una conclusión general con base en ejemplos o casos particulares. Este tipo de inferencia no garantiza la veracidad absoluta de la conclusión.*
- I. deductiva: *Se obtiene una conclusión específica a partir de premisas generalmente aceptadas como verdaderas y reglas de aplicación de premisas. Si las premisas son verdaderas y se aplican correctamente las reglas, la conclusión también debe ser verdadera.*

El proceso deductivo puede terminar cuando la conclusión que sea igual a una proposición que se desea demostrar, o bien cuando se determine que no es posible alcanzar tal proposición. Las conclusiones se pueden tomar como base para nuevos procesos deductivos y generar así nuevo conocimiento que sirva para otros procesos deductivos.

3.1.5 Notaciones para reglas de inferencia

Notación secuencial

Para escribir formalmente una regla de inferencia vamos a establecer el siguiente patrón de escritura:

$$\{p,\dots\} \vdash \{c,\dots\} \tag{3.1}$$

Como puedes observar, la notación se puede dividir en tres partes:

$$\underbrace{\boxed{\{p,\dots\}}}_{1} \quad \underbrace{\boxed{\vdash}}_{2} \quad \underbrace{\boxed{\{c,\dots\}}}_{3}$$

1. $\{p,\dots\}$, se puede leer como «con las premisas $p,\dots$». Es la sección de premisas. Las premisas constituyen el conocimiento actual, es la información con la que se dispone para construir los argumentos necesarios para alcanzar conclusiones. Todas las premisas tienen valor **V**.
2. $\vdash$ significa «es verdad». Sirve para decir que lo que sigue es cierto, es verdad.
3. $\{c,\dots\}$ es la sección de conclusiones. Se escriben después del símbolo '$\vdash$'. Pueden haber varias conclusiones separadas por coma. Cada conclusión se etiqueta por separado y se puede agregar al conocimiento adquirido para ser utilizada con alguna regla de inferencia y así poder alcanzar nuevas conclusiones.

Notación Gentzen

En 1933, Gerhard Karl Erich Gentzen [Gen33, Gen64] introdujo una notación para mostrar el proceso de inferencia, mostrando de manera ordenada las premisas y las conclusiones. En esta notación, las premisas se escriben arriba de una línea horizontal, que por razones de claridad, se pueden escribir apiladas. En la parte inferior de la línea se escribe cada una de las conclusiones seguida del símbolo '$\therefore$' que significa «se concluye».

$$\begin{array}{l} p_1 \\ \vdots \\ \hline \therefore \quad c_1 \ldots \end{array} \tag{3.2}$$

Ejemplo 3.2

La regla de inferencia llamada *modus ponendo ponens* [ver página 70], en la notación Gentzen se escribe:

$$\begin{array}{l} p \rightarrow q \\ p \\ \hline \therefore q \end{array}$$

3.2 Reglas de Inferencia

Ahora estableceremos algunas reglas para dar sentido correcto a las expresiones lógicas. En adelante trabajaremos más con símbolos que con proposiciones en lenguaje natural. Esto es por varios motivos:

1. La estructura de la fórmula es la misma sin importar lo que las variables proposicionales signifiquen en lenguaje natural.
2. El valor de verdad de una *fbf* se calcula a partir del valor de verdad que se le asigna a las premisas, a la construcción de la *fbf* y a las reglas de inferencia usadas.
3. Es mucho más cómodo, fácil de manipular y disminuye el riesgo de cometer errores.

Ejemplo 3.3

Digamos que $p \leftrightarrow$⟨*María tiene hambre y come mucho*⟩. La proposición p se puede modelar simbólicamente haciendo $q \leftrightarrow$⟨*María tiene hambre*⟩ y $r \leftrightarrow$⟨*María come mucho*⟩ ambas verdaderas como $r \leftrightarrow p \wedge q$.

Ahora, si p significa ⟨*El gato toma leche y la Luna es un satélite natural*⟩, la expresión $p \wedge q$ sigue teniendo la misma estructura y el mismo valor de verdad, porque las proposiciones atómicas también son **V**.

Una vez conocidas las estructuras de las proposiciones moleculares junto con los términos de enlace que permiten crear proposiciones más complejas, estudiaremos las principales reglas de inferencia.

3.2.1 Doble negación [DN]

La **doble negación**, por eso las iniciales DN, es una regla de inferencia que requiere una sola premisa y al aplicar dos veces la negación se obtiene nuevamente la premisa original. Por ejemplo la proposición:

«No es impuntual».

En principio, parece ser la negación de la proposición $\langle es\ impuntual\rangle$, pero la palabra «impuntual», en sí misma, es la negación de «puntual», por lo que en realidad se tiene la doble negación:

$$p \dashv (\neg(\neg\langle es\ puntual\rangle))$$

De acuerdo con la regla DN, se concluye que $\langle es\ puntual\rangle$, porque la primera negación cambia el valor de $\neg\langle es\ impuntual\rangle$, concluyendo $\langle es\ impuntual\rangle$, pero la segunda negación cambia nuevamente el valor y se concluye $\langle es\ puntual\rangle$.

La regla DN establece que $\neg\neg p$ significa exactamente lo mismo que p, esto es debido a los principios fundamentales de la lógica, los cuales establecen que solamente hay dos valores de verdad, **V** y **F** y que no hay posibilidad de tener otro valor diferente. De modo que al negar **V** se tiene **F** y al negar **F** se tiene **V**.

Esta regla se presenta en dos variantes:

1. $p \vdash \neg\neg p$ ▹ *que se lee:* con la premisa p se puede concluir $\neg\neg p$.
2. $\neg\neg p \vdash p$ ▹ *que se lee:* con la premisa $\neg\neg p$ se puede concluir p.

En la notación Gentzen se puede escribir:

1. $\dfrac{p}{\therefore \neg\neg p}$ 2. $\dfrac{\neg\neg p}{\therefore p}$

Ejemplo 3.4

La frase «No verás que no llego a tiempo» es una proposición doblemente negada, por lo que se concluye «llego a tiempo». En símbolos se establece que p es «llego a tiempo», que se puede traducir como $\neg\neg p$. De modo que

$$\frac{\neg\neg p}{\therefore p}$$

Así «llego a tiempo» es una conclusión lógica derivada de «no verás que no llego a tiempo».

En el uso de las reglas de inferencia pueden intervenir cualquier tipo de proposiciones lógicas, ya sea proposiciones atómicas o proposiciones moleculares.

Ejemplo 3.5

La frase «No ocurre que no, llueve o truena». Esta frase se puede simbolizar como $\neg\neg(p \vee q)$ y por la regla de la doble negación se concluye que $p \vee q$, es decir «llueve o truena».

Sabiendo que la negación de $\neg p$ es $\neg\neg p$, pero por la regla de la doble negación se tiene que $\neg\neg p$ equivale a p, podemos acortar la regla de la doble negación como

p es la negación de $\neg p$; ▹ *considerando que debería ser* $\neg(\neg p)$

lo que en sí, no es una nueva regla sino una manera más simple de la doble negación.

Ejemplo 3.6

Sea ahora ⟨*Antonio no está afiliado al club*⟩, al aplicar la regla DN, se tiene:

1. ¬¬⟨*Antonio no está afiliado al club*⟩.
2. ¬⟨*Antonio está afiliado al club*⟩.
3. ⟨*Antonio no está afiliado al club*⟩.

3.2.2 Adición [AD]

Una proposición disyuntiva tiene la forma «⟨___⟩ *o* ⟨___⟩ » en donde intervienen dos proposiciones. Por ejemplo la proposición «La naranja es dulce o está madura».

Observamos que se puede separar en sus dos proposiciones componentes:

$p \leftarrow\!\shortmid \langle$*La naranja es dulce*$\rangle$,

$q \leftarrow\!\shortmid \langle$*La naranja está madura*$\rangle$,

para formar la expresión simbólica $p \vee q$. Consideramos que $p \vee q$ es cierta cuando:

1. p es cierto, pero q no lo es. Por ejemplo «La naranja es dulce o está madura», es una expresión verdadera si se confirma que «la naranja es dulce».
2. q es cierto, pero p no lo es. En el ejemplo, la proposición disyuntiva será verdadera si solamente se confirma que «la naranja está madura».
3. ambas proposiciones, tanto p como q son ciertas.

De modo que una expresión disyuntiva $p \vee q$ es cierta cuando al menos una de las proposiciones que la componen es cierta. Por otro lado, el único caso cuando debemos considerar $p \vee q$ falsa, es cuando tanto p como q son falsas; en el ejemplo, «la naranja es dulce o está madura» será falso cuando se corrobora que la naranja no es dulce y que la naranja no está madura.

La regla de adición [AD] toma como base una proposición verdadera, digamos p. Por ahora no importa cuál es el significado real de p, porque lo importante en este momento es su verdad formal, que es **V** puesto que es una premisa.

De acuerdo con la interpretación de la disyunción, cualquier proposición disyuntiva será verdadera si tiene a p [de valor **V**] como uno de sus componentes.

En notación secuencial tenemos:

$$p \vdash (p \vee q), (q \vee p) \tag{3.3}$$

Donde q es una proposición atómica o molecular que puede ser **V** o bien **F**. Siguiendo la notación Gentzen, la regla de adición se escribe:

$$\frac{p}{\therefore p \vee q \quad \therefore q \vee p}$$

Ejemplo 3.7

Tomemos como ejemplo la premisa:

1. «Ella toma clases de francés». Simbólicamente asignamos esta proposición a una variable $p \leftarrow\!\shortmid \langle$*Marisela toma clases de francés*$\rangle$.

Se sabe que la proposición es cierta. Tomemos ahora $q \leftarrow\!\shortmid \langle$*Marisela toma clase de italiano*$\rangle$. De acuerdo con la regla de la adición, las siguientes dos proposiciones son verdaderas.

- $p \vee q$, que significa «Marisela toma clases de francés o de italiano»,
- $q \vee p$, que significa «Marisela toma clases de italiano o de francés».

La premisa que es base de esta regla de inferencia [y en general de todas ellas], es una expresión proposicional que puede ser atómica o molecular, por ejemplo las siguientes expresiones son aplicaciones de la regla de adición:

1. $\dfrac{p}{\begin{array}{l}\therefore p \vee (r \wedge q)\\ \therefore (r \wedge q) \vee p\end{array}}$

2. $\dfrac{(r \rightarrow (p \wedge \neg q))}{\begin{array}{l}\therefore (r \rightarrow (p \wedge \neg q)) \vee (r \vee q)\\ \therefore (r \vee q) \vee (r \rightarrow (p \wedge \neg q))\end{array}}$

Ejemplo 3.8

Tomemos ahora la premisa verdadera «Las tijeras están filosas y son caras». De manera simbólica se puede representar como $p \leftrightarrow$ ⟨*Las tijeras están filosas y son caras*⟩.

Si hacemos $q \leftrightarrow$ ⟨*La piedra de esmeril es de buena calidad*⟩, es válido suponer como verdadero cada una de las siguientes proposiciones:

1. $p \vee q$, que significa «Las tijeras están filosas y son caras, o la piedra de esmeril es de buena calidad».
2. $q \vee p$, que se puede interpretar como «La piedra de esmeril es de buena calidad, o las tijeras están filosas y son caras».

☞ Observa que $(p \vee q) \vdash p$ no es una regla válida. Esta propuesta dice que al tener una premisa de la forma $(p \vee q)$, se concluye que p es cierto, lo cual no es correcto, pues la veracidad de $(p \vee q)$ puede ser ocasionada por q, aunque p sea **F**.

Las disyunciones tienen algo especial, y es que en el lenguaje natural, una disyunción tiene un sentido excluyente, por ejemplo en la frase «Tomás está en la biblioteca o en la cancha», en este caso Tomás no puede estar en ambos lugares al mismo tiempo, hay una disyunción. En este sentido si es verdad que Tomás está en la biblioteca, entonces será falso que Tomás esté en la cancha, o al contrario.

En lógica se hace diferencia entre uno y otro caso, poniendo diferentes operadores lógicos, así el término de enlace '$\vee$' se aplica en situaciones donde se requiera un sentido incluyente, mientras que el término de enlace '$\oplus$' se utiliza cuando se quiera dar un sentido excluyente. La interpretación de una disyunción exclusiva $p \oplus q$ es entonces verdadera cuando exactamente una de las proposiciones componentes es verdadera y la otra es falsa. Por otro lado, $p \oplus q$ será falsa cuando ambas sean verdaderas, o ambas sean falsas.

En el lenguaje natural es un poco más difícil establecer una regla para especificar cuando se trata de una disyunción exclusiva, pero podemos llegar al acuerdo de escribir «o bien» `<prop1>` «o» `<prop2>`, por ejemplo «Tomás está, o bien en la biblioteca, o en la cancha», claro hay otras maneras, por ejemplo «Tomás está, una de dos, o en la biblioteca o en la cancha».

Ejemplo 3.9

Las siguientes expresiones deben ser tratadas como disyunciones en un sentido exclusivo:

- «Esta tarde, O voy al paseo o asistiré a la fiesta de cumpleaños».
  Se puede crear una proposición disyuntiva exclusiva con proposiciones atómicas:
  $p \leftrightarrow$ ⟨*Voy al paseo*⟩.
  $q \leftrightarrow$ ⟨*Asistiré a la fiesta de cumpleaños*⟩.
  $p \oplus q \leftrightarrow$ ⟨*Voy al paseo*⟩ $\oplus$ ⟨*Asistiré a la fiesta de cumpleaños*⟩.
  La disyunción es exclusiva porque en este ejemplo, no se pueden hacer ambas actividades a la vez.

- «O bien si está nublado, entonces no saldré al parque; o si hace sol, entonces iré a nadar».
  La proposición disyuntiva exclusiva se puede crear con dos proposiciones implicativas:
  ⟨*Si está nublado, entonces no saldré al parque*⟩ ⊕⟨*si hace sol, entonces iré a nadar*⟩.
  Cada una de las proposiciones se puede descomponer aún más:
  $p_1 \leftarrow\!\!\shortmid$⟨*Está nublado*⟩.
  $p_2 \leftarrow\!\!\shortmid$⟨*Saldré al parque*⟩.
  $q_1 \leftarrow\!\!\shortmid$⟨*Hace sol*⟩.
  $q_2 \leftarrow\!\!\shortmid$⟨*Iré a nadar*⟩.

Así la expresión original se puede escribir:

$$(p_1 \rightarrow p_2) \oplus (q_1 \rightarrow q_2)$$

La regla de adición en el sentido excluyente no es una regla de inferencia válida, porque si se toma una premisa p, no se puede agregar una proposición q [de valor **V**] para concluir ya sea $p \oplus q$ o $q \oplus p$, porque ambas proposiciones tienen el mismo valor de verdad y eso ocasiona que se generen conclusiones falsas, esto se puede escribir como: $p, q \nvdash (p \oplus q), (q \oplus p)$, donde el símbolo '$\nvdash$' significa «no concluye». Sin embargo, si se tiene como premisa una disyunción inclusiva que se define con dos conjunciones de la forma $(p \wedge \neg q) \vee (\neg p \wedge q)$, sí se puede concluir una disyunción exclusiva, la demostración de esta regla de inferencia se encuentra en la página 110:

$$(p \wedge \neg q) \vee (\neg p \wedge q) \vdash (p \oplus q), (q \oplus p). \tag{3.4}$$

3.2.3 Agregación [AG]

Para comprender esta regla de inferencia, es necesario establecer un criterio para tratar con las expresiones proposicionales conjuntivas de forma ⟨___⟩ ∧⟨___⟩, es decir, un par de proposiciones que pueden ser atómicas o moleculares unidas por el término de enlace conjuntivo '∧'.

Si p y q son proposiciones, la expresión «p y q» se denota $p \wedge q$, que es verdadera si ambas proposiciones componentes, tanto p como q, son verdaderas. Por otro lado, $p \wedge q$ será falsa en cualquier otro caso, es decir, si al menos una de las proposiciones, ya sea p o q es falsa.

Ejemplo 3.10

La proposición conjuntiva ⟨*El sombrero es de paja y el vestido es tradicional*⟩ es verdadera cuando se constata que el sombrero es de paja y que el vestido es tradicional.

Si al verificar ambas resulta que el sombrero no es de paja, será motivo suficiente para determinar que «el sombrero es de paja y el vestido es tradicional» es falso. Se llega a la misma conclusión cuando el vestido no es tradicional, aunque el sombrero sí sea de paja.

La regla de agregación [AG], también conocida como «adjunción»[SH86], «conjunción o combinación»[Ros04] requiere dos premisas, digamos p y q. Como ambas premisas son verdaderas [por el hecho de ser premisas], se podrán agregar o conjuntar en una proposición molecular conjuntiva $p \wedge q$ o bien $q \wedge p$ que claramente es verdadera de acuerdo al criterio de interpretación de la conjunción.

De manera simbólica, la regla de la agregación se denota:

$$p, q \vdash (p \wedge q), (q \wedge p). \tag{3.5}$$

La lectura es como sigue: «Al tener las premisas p y q, se puede concluir $p \wedge q$ o bien $q \wedge p$». En la notación Gentzen se puede escribir:

$$\begin{array}{l} p \\ q \\ \hline \therefore p \wedge q \\ \therefore q \wedge p \end{array}$$

Observa que las conclusiones difieren únicamente en el orden de las proposiciones que la componen, pero como ambas son verdaderas, la interpretación es la misma: La conjunción $p \wedge q$ es verdad, también $q \wedge p$ es verdad.

Ejemplo 3.11

Supongamos que p significa «Agustín aprobó historia», y que el símbolo q significa la proposición «Agustín aprobó matemáticas». Como tanto p como q son ciertas, podemos concluir que $p \wedge q$ es cierto. Lo que significa que «Agustín aprobó historia y matemáticas», aunque también se puede decir que «Agustín aprobó matemáticas e historia».

Nuevamente, tanto p como q representan las premisas, que son proposiciones atómicas o moleculares; además pueden ser tan complejas como sea. Enseguida muestro algunas formas en que pueden aparecer las proposiciones, pero en todas ellas se observa la regla de agregación.

1. $$\begin{array}{l} p \\ \neg q \\ \hline \therefore p \wedge \neg q \\ \therefore \neg q \wedge p \end{array}$$

2. $$\begin{array}{l} (p \wedge q) \\ \neg r \\ \hline \therefore (p \wedge q) \wedge \neg r \\ \therefore \neg r \wedge (p \wedge q) \end{array}$$

3. $$\begin{array}{l} (p \vee q) \\ (q \vee r) \\ \hline \therefore (p \vee q) \wedge (q \vee r) \\ \therefore (q \vee r) \wedge (p \vee q) \end{array}$$

Hay ocasiones en que no es posible alcanzar una conclusión en un solo paso, pero si se cuenta con las premisas necesarias, entonces se puede crear una sucesión de razonamientos.

Ahora identificaremos las premisas con '$\mathtt{P}_n$' donde n es un número y '$\mathtt{C}_m$' las conclusiones, con m también un número entero positivo.

Ejemplo 3.12

Este es un ejemplo en donde se utiliza la regla de agregación junto con la doble negación [página 62], veamos qué se puede deducir de las expresiones «no soy desaseado» y «soy puntual y responsable».

Primero encontramos las proposiciones atómicas:

$p \leftrightarrow \langle$*soy aseado*$\rangle$,
$q \leftrightarrow \langle$*soy puntual*$\rangle$,
$r \leftrightarrow \langle$*soy responsable*$\rangle$.

Ahora, escribimos las premisas dadas en forma de expresiones simbólicas:

$\mathtt{P}_1$: $\neg\neg p$. ▹ no *es verdad que* no *soy aseado, también podemos decir* no *soy desaseado*.
$\mathtt{P}_2$: $q \wedge r$. ▹ *soy puntual* y *responsable*.

Paso	Regla usada		Conclusión:
①	DN($\mathtt{P}_1$)	$\mathtt{C}_1$:	p
②	AG($\mathtt{C}_1$, $\mathtt{P}_2$)	$\mathtt{C}_2$:	$p \wedge (q \wedge r)$

En el paso 2 se utilizó la regla de agregación con p que es la conclusión $\mathtt{C}_1$ y con $(q \wedge r)$ que es la premisa $\mathtt{P}_2$:

$$\begin{array}{l} p \\ (q \wedge r) \\ \hline \therefore p \wedge (q \wedge r) \\ \therefore (q \wedge r) \wedge p \end{array}$$

de lo que se puede concluir $p \wedge (q \wedge r)$ que significa «Soy aseado, soy puntual y responsable». Además se concluye $(q \wedge r) \wedge p$ que significa «Soy puntual y responsable, y soy aseado».

3.2.4 Disgregación [DI]

La regla de disgregación [DI], trabaja en forma contraria a la de agregación. Toma una proposición conjuntiva como premisa, con la que se concluyen dos proposiciones, precisamente las que conforman la conjunción original.

En notación secuencial la regla DI se escribe:

$$(p \wedge q) \vdash p, q \tag{3.6}$$

que puede leerse como: «Sabiendo que $p \wedge q$ es verdadero, podemos concluir que p es **V**, además q también es **V**.» En la notación Gentzen se escribe:

$$\begin{array}{l} p \wedge q \\ \hline \therefore p \\ \therefore q \end{array}$$

Ejemplo 3.13

Un pintor, a quien le gusta hacer sus propios colores, sabe que al mezclar azul y amarillo se obtiene verde. Cuando el pintor entra a su estudio encuentra un frasco con pintura verde, por lo que inmediatamente deduce que se ha usado un poco de pintura azul; también deduce que se ha usado pintura color amarillo.

El razonamiento del pintor ha seguido la regla de disgregación con la expresión «El verde tiene azul y amarillo», que convierte en una proposición conjuntiva $\langle El\ verde\ tiene\ azul \rangle \wedge \langle el\ verde\ tiene\ amarillo \rangle$; luego utilizando la regla de disgregación con la premisa, ha podido obtener ambas conclusiones:

$$\begin{array}{l} \langle El\ verde\ tiene\ azul \rangle \wedge \langle El\ verde\ tiene\ amarillo \rangle \\ \hline \therefore \langle El\ verde\ tiene\ azul \rangle \\ \therefore \langle El\ verde\ tiene\ amarillo \rangle \end{array}$$

Es importante asegurar que la proposición $p \wedge q$ sea **V**, pues de lo contrario no se puede utilizar como premisa, pues no hay certeza del valor de verdad de cada proposición.

Hay expresiones que deben ser analizadas muy cuidadosamente antes de poder aplicar esta regla, por ejemplo $p \wedge q \rightarrow r$. La regla DI no se aplica en $(p \wedge q) \rightarrow r$ porque la premisa es una implicación, pero sí se puede aplicar en $p \wedge (q \rightarrow r)$ porque es una conjunción.

Enseguida muestro varios ejemplos en donde se han empleado premisas escritas en diferentes maneras:

1. $\dfrac{(p \vee q) \wedge r}{\therefore (p \vee q)\ \therefore r}$ 2. $\dfrac{(q \to r) \wedge (p \to q)}{\therefore (q \to r)\ \therefore (p \to q)}$ 3. $\dfrac{(p \wedge r) \wedge q}{\therefore (p \wedge r)\ \therefore q}$

Ejemplo 3.14

Analicemos la siguiente premisa:

P_1: «Si llueve entonces el piso está húmedo, y, si el piso está húmedo, entonces me resbalo».

De manera simbólica tenemos $p \to q$ donde:

$p \leftrightarrow \langle$*Si llueve entonces el piso esta húmedo*$\rangle$
$p_1 \leftrightarrow \langle$*Llueve*$\rangle$
$p_2 \leftrightarrow \langle$*El piso está húmedo*$\rangle$
$q \leftrightarrow \langle$*Si el piso está húmedo, entonces me resbalo*$\rangle$
$p_2 \leftrightarrow \langle$*El piso está húmedo*$\rangle$
$p_3 \leftrightarrow \langle$*Me resbalo*$\rangle$

Así P_1 escrita de manera simbólica es $(p_1 \to p_2) \wedge (p_2 \to p_3)$. Usando la regla DI con la premisa P_1 obtenemos las siguientes conclusiones:

1. $p_1 \to p_2$ ▷ *Si llueve, entonces el piso está húmedo.*
2. $p_2 \to p_3$ ▷ *Si el piso está húmedo, entonces me resbalo.*

3.2.5 *Modus Tollendo Ponens* [TP]

Esta nueva regla se puede entender como «negando una se afirma la otra», aunque su nombre original en latín es *modus tollendo ponens* que literalmente significa «el modo que al negar se afirma», y la interpretación está basada principalmente en una disyunción que sabemos verdadera.

Además de la premisa disyuntiva $p \vee q$, se requiere la negación de una de las proposiciones que componen la disyunción, así se requiere ya sea $\neg p$ o $\neg q$, con lo que es suficiente para afirmar la otra componente. Simbólicamente tenemos:

$$\begin{array}{l}(p \vee q), \neg p \vdash q \\ (p \vee q), \neg q \vdash p\end{array} \tag{3.7}$$

En la notación Gentzen se pueden escribir estas mismas reglas como:

1. $\dfrac{p \vee q\ \ \neg p}{\therefore q}$ 2. $\dfrac{p \vee q\ \ \neg q}{\therefore p}$

Para explicar estas reglas seguiremos un razonamiento en donde supondremos lo contrario de la conclusión, con el fin de alcanzar un estado que contradiga los supuestos dados como verdaderos. Consideremos solamente la regla $(p \vee q), \neg q \vdash p$, ya que para la otra regla se sigue un razonamiento similar.

En palabras esta regla dice que, si sabemos que la disyunción de p con q es verdadera, y además sabemos que q es falsa, entonces con seguridad, p es verdadera.

Supongamos por ahora que p es falsa, que es lo contrario de la conclusión en la regla de inferencia. Como p es falsa [supuestamente] y $p \vee q$ es verdadera porque es una premisa; por los criterios de la disyunción [página 63] la única manera en que $p \vee q$ siga siendo verdadera es que q sea verdadera, pero sabemos que q es falsa, porque es una premisa. El principio de la bivalencia [página 57] exige que q o es

verdadera o es falsa, pero no puede tener ambos valores al mismo tiempo, entonces la suposición de que p es falsa estuvo equivocada y p debe ser verdadera.

Ejemplo 3.15

Se tienen las siguientes premisas:

P1: «El té tiene azúcar o miel».
P2: «El té no tiene miel».

Usando la regla TP se debe concluir que «el té tiene azúcar». Podemos reescribir las premisas con símbolos:

P1: $p \vee q \triangleright$ *El té tiene azúcar o miel*.
P2: $\neg q \triangleright$ *El té no tiene miel*.

La regla utilizada es $\dfrac{p \vee q \quad \neg q}{\therefore p}$, por lo que se deduce p, que es «El té tiene azúcar».

La regla TP se puede presentar en diversas formas, pero lo importante es observar dos características: que una premisa sea una disyunción y la segunda característica es que aparezca una proposición que compone la disyunción, con valor de verdad diferente. En los siguientes ejemplo se muestran algunas formas en que se puede utilizar TP:

1. $\dfrac{(r \wedge s) \vee q \quad \neg(r \wedge s)}{\therefore q}$

2. $\dfrac{\neg(p \wedge q) \vee s \quad \neg s}{\therefore \neg(p \wedge q)}$

3. $\dfrac{p \vee \neg(r \rightarrow q) \quad \neg\neg(r \rightarrow q)}{\therefore p}$

4. $\dfrac{\neg(p \wedge r) \vee (q \wedge t) \quad \neg(q \wedge t)}{\therefore \neg(p \wedge r)}$

5. $\dfrac{\neg p \vee q \quad p}{\therefore q}$

6. $\dfrac{p \vee \neg q \quad q}{\therefore p}$

En el uso de las reglas de inferencia, se suelen tener en consideración todas las premisas y la colección de reglas de inferencia conocidas. En el siguiente ejemplo se considera una lista de premisas y se debe llegar a una proposición que sea una consecuencia lógica de las premisas.

Ejemplo 3.16

Con las siguientes premisas:

P_1: $\neg\neg r$
P_2: $\neg r \vee (q \wedge t)$

Se debe concluir t. Para esto, se pueden seguir los siguientes pasos:

Paso	Regla usada		Conclusión:
(1)	DN(P_1)	C_1:	r
(2)	TP(C_1, P_2)	C_2:	$q \wedge t$
(3)	DI(C_2)	C_3:	q
(4)	DI(C_2)	C_4:	t

Al final, la conclusión C_4 establece que t es verdadero.

3.2.6 *Modus Ponendo Ponens* [PP]

El nombre de esta regla proviene del latín *modus ponendo ponens* que literalmente significa «el modo que afirmando se afirma». Aunque el nombre parezca un tanto confuso, podemos entenderlo un poco mejor sabiendo cómo funciona la regla. Esta regla consiste en tener una implicación y afirmar que el antecedente es verdadero. Sabiendo esto, se puede confirmar con seguridad que el consecuente de la implicación es verdadero.

☞ Una implicación es de la forma $p \rightarrow q$, donde la proposición p se le llama «antecedente» y a la proposición q se le llama «consecuente» [Sección 2.2.4, página 39].

La interpretación de una implicación se basa en la relación causa–efecto entre las proposiciones que componen la proposición implicativa. Suponiendo que la regla de una implicación es $p \rightarrow q$, la relación causal indica que una vez que ocurra p, ocurre q, en otras palabras, p es la causa y q es el efecto, queda claro entonces que si p ocurre, pero no se produce el efecto q, entonces $p \rightarrow q$ debe ser **F**.

Así, cualquiera que sea la relación implicativa donde p sea el antecedente y q sea el consecuente será **V**, excepto cuando p sea **V** y q sea **F**. Explícitamente se tienen los siguientes cuatro casos para $p \rightarrow q$:

1. $\mathbf{V} \rightarrow \mathbf{V} \vdash \mathbf{V}$. ▹ *Se lee: Si una proposición verdadera implica una proposición verdadera, se concluye que la proposición implicativa es verdadera.*
2. $\mathbf{V} \rightarrow \mathbf{F} \vdash \mathbf{F}$.
3. $\mathbf{F} \rightarrow \mathbf{V} \vdash \mathbf{V}$.
4. $\mathbf{F} \rightarrow \mathbf{F} \vdash \mathbf{V}$.

Simbólicamente la regla PP es

$$(p \rightarrow q), p \vdash q \tag{3.8}$$

La lectura es «Afirmando que $p \rightarrow q$ es cierto y confirmando que el antecedente p es cierto, se concluye que el consecuente q es cierto». En notación de Gentzen se escribe:

$$\frac{\begin{array}{l} p \rightarrow q \\ p \end{array}}{\therefore q}$$

Ejemplo 3.17

Ahora consideremos la expresión «Si no se sufre, entonces no se aprende». Aquí podemos escribir la misma frase con símbolos como: $\neg p \rightarrow \neg q$, donde $p \leftarrow\!\!\shortmid \langle \textit{se sufre} \rangle$, $q \leftarrow\!\!\shortmid \langle \textit{se aprende} \rangle$.

$$\frac{\begin{array}{l} \neg p \rightarrow \neg q \\ \neg p \end{array}}{\therefore \neg q} \qquad \frac{\begin{array}{l} \langle \textit{No se sufre} \rangle \rightarrow \langle \textit{No se aprende} \rangle \\ \langle \textit{No se sufre} \rangle \end{array}}{\therefore \langle \textit{No se aprende} \rangle}$$

La regla *modus ponendo ponens* garantiza que el consecuente de una implicación es una consecuencia lógica tanto de la implicación como del antecedente de esta. Esta regla, funciona por ejemplo, cuando hay una advertencia de avisa que si ocurre alguna cosa, otra cosa sucede con seguridad, luego se constata que efectivamente ocurre lo esperado y claro, sucede lo que la regla advierte.

Ejemplo 3.18

Hay un refrán que advierte «siembra vientos y cosecharás tempestades». Aunque el refrán parece una conjunción por tener el conjuntor «y», en realidad es una implicación, donde se establece que si se siembran vientos, en el futuro como consecuencia de esto, se cosecharán tempestades. Este refrán se puede escribir en forma de una proposición implicativa como:

$$\langle \textit{Si [él] siembra vientos, entonces [él] cosecha tempestades} \rangle$$

Supongamos ahora que nos referimos a un líder de equipo que trata mal a sus compañeros, es egoísta y manda sin colaborar. Lo que sucede se puede interpretar como «siembra vientos». Luego los compañeros se pueden sentir incómodos e infravalorados; en el futuro se puede generar un ambiente de trabajo tóxico y los resultados son deficientes, lo que es interpretado como «cosecha tempestades».

Así:

- «Si el líder es egoísta y manda sin colaborar, entonces el ambiente es tóxico y los resultados son deficientes» es la premisa implicativa donde:
  $p_1 \leftrightarrow \langle \textit{El líder es egoísta} \rangle$,
  $p_2 \leftrightarrow \langle \textit{El líder manda sin colaborar} \rangle$,
  $q_1 \leftrightarrow \langle \textit{El ambiente es tóxico} \rangle$,
  $q_2 \leftrightarrow \langle \textit{Los resultados son deficientes} \rangle$,
  $p_1 \wedge p_2 \rightarrow q_1 \wedge q_2$.
- Se observa que ocurre $p_1 \wedge p_2$, que es otra premisa y coincide con el antecedente de la implicación.
- Se aplica la regla PP $(p_1 \wedge p_2 \rightarrow q_1 \wedge q_2), (p_1 \wedge p_2) \vdash (q_1 \wedge q_2)$.
- En la notación Gentzen:
$$\dfrac{p_1 \wedge p_2 \rightarrow q_2 \wedge q_2 \quad p_1 \wedge p_2}{\therefore q_2 \wedge q_2}$$

Por lo que se concluye $q_1 \wedge q_2$, esto es que «el ambiente es tóxico y los resultados son deficientes» es una consecuencia lógica de las premisas dadas.

Se puede encontrar muchas situaciones donde se aplica PP, observa que en todos los casos el antecedente de la implicación se confirma como una premisa:

1. $\dfrac{p \rightarrow q \quad p}{\therefore q}$

2. $\dfrac{p \rightarrow \neg q \quad p}{\therefore \neg q}$

3. $\dfrac{p \wedge q \rightarrow r \quad p \wedge q}{\therefore r}$

4. $\dfrac{\neg p \rightarrow q \quad \neg p}{\therefore q}$

5. $\dfrac{p \rightarrow q \wedge r \quad p}{\therefore q \wedge r}$

La situación 2 establece la condicional en segundo lugar, pero el orden de las premisas no es relevante. Cuando hay expresiones que pudieran ser confusas se pueden colocar paréntesis. Por ejemplo $p \rightarrow q \rightarrow r$ se interpreta $(p \rightarrow q) \rightarrow r$ y no como $p \rightarrow (q \rightarrow r)$, a menos que los paréntesis indiquen un orden específico.

Tomemos como ejemplo las siguientes premisas:

P_1: $\langle \textit{Si llueve, se moja la ropa} \rangle$.
P_2: $\langle \textit{Si se moja la ropa, entonces no puedo salir de casa} \rangle$.
P_3: $\langle \textit{Si no puedo salir de casa, entonces me aburro} \rangle$.
P_4: $\langle \textit{Llueve} \rangle$.

Es importante observar que esta lista de premisas se puede simbolizar como:

P_1: $p \rightarrow q$
P_2: $q \rightarrow r$
P_3: $r \rightarrow s$
P_4: p

Es necesario comprender cada parte de las premisas. Ahora se quiere demostrar s, es decir ⟨*me aburro*⟩; por lo que se sigue el siguiente razonamiento:

① Usando PP con las premisas 1 y 4 [PP(P_1,P_4)] se concluye C_1: q

$$\begin{array}{l} p \rightarrow q \\ \underline{p} \\ \therefore q \end{array}$$

② Usando PP con la premisa P_2 y la conclusión C_1 [PP(P_2,C_1)] se concluye C_2: r

$$\begin{array}{l} q \rightarrow r \\ \underline{q} \\ \therefore r \end{array}$$

③ Usando PP(P_3,C_2)] se concluye C_3: s

$$\begin{array}{l} r \rightarrow s \\ \underline{r} \\ \therefore s \end{array}$$

Para concluir s se necesitaron tres pasos, cada uno de ellos aplicando la regla de inferencia PP. En el primer paso podemos concluir q, luego utilizando la primera conclusión q como premisa junto con $q \rightarrow r$, podemos concluir r; ahora tenemos a r como una premisa y junto con $r \rightarrow s$ podemos utilizar nuevamente PP para finalmente concluir s.

Normalmente se utiliza más de una regla, pues los procesos deductivos generalmente son de más de un paso e involucran premisas en diversas formas.

Ejemplo 3.19

Considera las siguientes premisas:

P_1: «Si no mantengo sana distancia, entonces no ocurre que no me enfermo».
P_2: «No mantengo sana distancia».

¿Qué se puede deducir de esta colección de premisas? Primero se traduce el enunciado en premisas en el lenguaje simbólico:

P_1: $\neg p \rightarrow \neg\neg q$.
P_2: $\neg p$.

Usamos PP: $\dfrac{\neg p \rightarrow \neg\neg q \quad \neg p}{\therefore \neg\neg q}$, ahora usamos DN: $\dfrac{\neg\neg q}{\therefore q}$. Siendo q la proposición «me enfermo».

Podemos decir que «me enfermo» es una consecuencia lógica de las premisas dadas.

3.2.7 *Modus Tollendo Tollens* [TT]

Esta nueva regla se llama «al negar se contradice». En latín *modus tollendo tollens* significa literalmente «el modo que al negar se niega».

Se requieren entonces dos premisas, una expresión condicional y otra proposición que niega el consecuente de la condicional:

$$(p \rightarrow q), (\neg q) \vdash \neg p \tag{3.9}$$

En la notación Gentzen se observa con claridad las dos premisas y debe notarse que el consecuente de la implicación aparece negado como segunda premisa, aunque el orden en las premisas no es importante:

$$\begin{array}{l} p \rightarrow q \\ \neg q \\ \hline \therefore \neg p \end{array}$$

Ejemplo 3.20

Se sabe que «Si la ropa está limpia, entonces la ropa huele bien», además resulta que la ropa no huele bien.

Podemos reescribir las premisas como:

P_1: $p \rightarrow q$ significa ⟨*Si la ropa está limpia, entonces la ropa huele bien*⟩, donde:
$p \leftrightarrow$⟨*La ropa está limpia*⟩, ▹ *El antecedente.*
$q \leftrightarrow$⟨*La ropa huele bien*⟩. ▹ *El consecuente.*

P_2: $\neg q$, se observa que ⟨*La ropa no huele bien*⟩. ▹ *La negación del consecuente.*

Por la regla TT:

$$\begin{array}{l} p \rightarrow q \\ \neg q \\ \hline \therefore \neg p \end{array}$$

de lo cual se concluye $\neg$⟨*la ropa está limpia*⟩, que se interpreta como «la ropa no está limpia».

Ejemplos de otras maneras en las que se puede utilizar la regla TT.

1. $\begin{array}{l} p \rightarrow q \\ \neg q \\ \hline \therefore \neg p \end{array}$ 2. $\begin{array}{l} (p \wedge r) \rightarrow s \\ \neg s \\ \hline \therefore \neg (p \wedge r) \end{array}$ 3. $\begin{array}{l} p \rightarrow \neg q \\ \neg \neg q \\ \hline \therefore \neg p \end{array}$

Ejemplo 3.21

En este ejemplo se tienen tres premisas y se desea demostrar que r es una consecuencia lógica de las premisas.

P_1: $p \rightarrow q$
P_2: $\neg q$
P_3: $\neg p \rightarrow \neg \neg r$

Paso	Regla usada	Explicación		Conclusión:
①	TT(P_1, P_2)	$(p \rightarrow q), \neg q \vdash \neg p$	C_1:	$\neg p$
②	PP(P_3, C_1)	$\neg p, (\neg p \rightarrow \neg \neg r) \vdash \neg \neg r$	C_2:	$\neg \neg r$
③	DN(C_2)	$\neg \neg r \vdash r$	C_3:	r ■

El símbolo '■' se agrega al final de una demostración, y se lee «lo que queda demostrado».

Ejemplo 3.22

Con las siguientes premisas se desea demostrar r.

P_1: $s \vee q$
P_2: $(q \vee p) \rightarrow r$
P_3: $s \rightarrow t$
P_4: $p \wedge \neg t$

Paso	Regla usada	Explicación		Conclusión:
①	DI(P_4) [p. 67]	$p \wedge \neg t \vdash \neg t$	C_1:	$\neg t$
②	TT(C_1, P_3) [p. 72]	$\neg t, (s \rightarrow t) \vdash \neg s$	C_2:	$\neg s$
③	TP(C_2,P_1) [p. 68]	$\neg s, (s \vee q) \vdash q$	C_3:	q
④	AD(C_3, q) [p. 63]	$q, p \vdash (q \vee p)$	C_4:	$(q \vee p)$
⑤	PP(C_4) [p. 70]	$(q \vee p) \rightarrow r \vdash r$	C_5:	r ■

Ejemplo 3.23

Con las siguientes premisas se desea demostrar $\alpha = 9 \vee \beta \neq 7$.

P_1: $\alpha \neq \beta$
P_2: $\beta = \alpha + \kappa$
P_3: $\alpha < \beta \vee \alpha = \beta$
P_4: $(\alpha > 7 \vee \alpha < 2) \wedge \beta = \alpha + \kappa \rightarrow \beta \neq 7$
P_5: $\alpha < \beta \vee \beta = 12 \rightarrow \alpha > 7$

Paso	Regla usada	Explicación		Conclusión:
①	TP(P_1,P_3) [p. 68]	$\alpha \neq \beta, (\alpha < \beta \vee \alpha = \beta) \vdash \alpha < \beta$	C_1:	$\alpha < \beta$
②	AD(C_1,$\beta = 12$) [p. 63]	$(\alpha < \beta), (\beta = 12) \vdash \alpha < \beta \vee \beta = 12$	C_2:	$\alpha < \beta \vee \beta = 12$
③	PP(C_2,P_5) [p. 70]	$\alpha < \beta \vee \beta = 12$, $\alpha < \beta \vee \beta = 12 \rightarrow \alpha > 7$ $\vdash \alpha > 7$	C_3:	$\alpha > 7$
④	AD(C_3, $\alpha < 2$) [p. 63]	$\alpha > 7, \alpha < 2 \vdash (\alpha > 7 \vee \alpha < 2)$	C_4:	$(\alpha > 7 \vee \alpha < 2)$
⑤	AG(C_4, P_2) [p. 65]	$(\alpha > 7 \vee \alpha < 2), \beta = \alpha + \kappa$ $\vdash (\alpha > 7 \vee \alpha < 2) \wedge (\beta = \alpha + \kappa)$	C_5:	$(\alpha > 7 \vee \alpha < 2) \wedge (\beta = \alpha + \kappa)$
⑥	PP(C_5,P_4) [p. 70]	$(\alpha > 7 \vee \alpha < 2) \wedge (\beta = \alpha + \kappa)$, $(\alpha > 7 \vee \alpha < 2) \wedge (\beta = \alpha + \kappa) \rightarrow \beta \neq 7$ $\vdash \beta \neq 7$	C_6	$\beta \neq 7$
⑦	AD(C_5, $\alpha = 9$) [p. 63]	$\beta \neq 7, \alpha = 9 \vdash \alpha = 9 \vee \beta \neq 7$	C_2:	$\alpha = 9 \vee \beta \neq 7$ ■

3.3 Silogismo hipotético [SH]

Un silogismo es una manera de razonar en la que se toman dos proposiciones como premisas y se obtiene una tercera proposición, que es el resultado deductivo de las premisas.

☞ Silogismo : [M. FIL.] Argumento que consta de tres proposiciones, la última de las cuales se deduce necesariamente de las otras dos.

Este silogismo requiere dos premisas en la forma de proposiciones condicionales, una es de la forma $p \rightarrow q$, y la otra es de la forma $q \rightarrow r$. Observa que el consecuente de la primera implicación es el antecedente de la segunda. De estas premisas se puede concluir la proposición implicativa $p \rightarrow r$:

$$(p \rightarrow q), (q \rightarrow r) \vdash (p \rightarrow r) \tag{3.10}$$

y en la notación Gentzen podemos escribir:

$$\begin{array}{l} p \to q \\ q \to r \\ \hline \therefore\ p \to r \end{array}$$

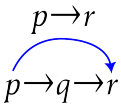
El silogismo hipotético tiene transitividad en el sentido de que la proposición que antecede en la primera implicación, también antecede en la nueva implicación; y el consecuente de la segunda implicación es también el consecuente de la implicación inferida.

☞ **Transitividad:** [F.] Cualidad de transitivo.
Transitivo: [ADJ.] Que pasa y se transfiere de uno a otro.

Con ambas implicaciones se puede formar un «encadenamiento implicativo» que inicia en el primer antecedente y termina con el segundo consecuente:

$$p \to q \to r$$

La transitividad se muestra al pasar directamente de p a r, omitiendo la proposición transitiva q.



Ejemplo 3.24

Supongamos las siguientes dos proposiciones dadas como premisas:

P1: «Si María llega a tiempo al teatro, entonces escucha toda la obra».
P2: «Si María escucha toda la obra, entonces es feliz».

Descubriendo las proposiciones dadas:

1. $p \leftarrow\!\shortmid \langle$*María llega a tiempo al teatro*$\rangle$.
2. $q \leftarrow\!\shortmid \langle$*María escucha toda la obra*$\rangle$.
3. $r \leftarrow\!\shortmid \langle$*María es feliz*$\rangle$.

Podemos reescribir las premisas en forma simbólica:

P1: $p \to q$
P2: $q \to r$

Se concluye $p \to r$, lo que significa:

«Si María llega a tiempo al teatro, entonces es feliz».

Las proposiciones que componen las implicaciones del silogismo hipotético, pueden, al igual que en todas las reglas de inferencia, ser moleculares o atómicas. Analiza el siguiente ejemplo.

Ejemplo 3.25

Las siguientes dos proposiciones son dadas como premisas:

P_1: «Si hierve el agua y pasan al menos 5 minutos, entonces el agua es potable».
P_2: «Puedo tomar agua o lavar las verduras porque el agua es potable».

En términos simbólicos, las premisas se pueden descomponer en sus proposiciones atómicas:

$p_1 \leftarrow\!\shortmid \langle$*Hierve el agua*$\rangle$.
$p_2 \leftarrow\!\shortmid \langle$*Pasan al menos 5 minutos*$\rangle$.

$q \leftarrow$ ⟨*El agua es potable*⟩.
$r_1 \leftarrow$ ⟨*Puedo tomar agua*⟩.
$r_1 \leftarrow$ ⟨*Puedo lavar verduras*⟩.
Ahora, reescribiendo las proposiciones de manera simbólica:
P_1: $(p_1 \wedge p_2) \rightarrow q$
P_2: $q \rightarrow (r_1 \vee r_2)$
Observamos que hay dos implicaciones y que el consecuente de la primera es el antecedente de la segunda, por lo que podemos utilizar el silogismo hipotético:

$$\begin{array}{l} (p_1 \wedge p_2) \rightarrow q \\ q \rightarrow (r_1 \vee r_2) \\ \hline \therefore (p_1 \wedge p_2) \rightarrow (r_1 \vee r_2) \end{array}$$

Con lo que podemos concluir que «Si hierve el agua y pasan al menos 5 minutos, entonces puedo tomar agua o lavar verduras».

3.4 Dilema constructivo [DC]

Esta regla es muy interesante, toma tres premisas: Una disyunción de la forma $p \vee q$ y dos condicionales, una de ellas de la forma $p \rightarrow r$ y la otra de la forma $q \rightarrow s$. Observa que las dos proposiciones que participan en la disyunción son los antecedentes de las condicionales. Lo que se concluye es una nueva disyunción de la forma $r \vee s$, formada con los dos consecuentes de las implicaciones.

En términos simbólicos se tiene:

$$(p \vee q), (p \rightarrow r), (q \rightarrow s) \vdash (r \vee s), (s \vee r) \tag{3.11}$$

Y en la notación Gentzen se puede escribir:

$$\begin{array}{l} p \vee q \\ p \rightarrow r \\ q \rightarrow s \\ \hline \therefore r \vee s \\ \therefore s \vee r \end{array}$$

Ejemplo 3.26
Tenemos las premisas:
P_1: «La manzana es saludable o la calabaza es saludable».
P_2: «Si la manzana es saludable, entonces se puede comer».
P_3: «Si la calabaza es saludable, entonces se usa en la preparación de comida».
Podemos simbolizar cada premisa haciendo:
$p \leftarrow$ ⟨*La manzana es saludable*⟩.
$q \leftarrow$ ⟨*La manzana se puede comer*⟩.
$r \leftarrow$ ⟨*La calabaza es saludable*⟩.
$s \leftarrow$ ⟨*La calabaza se usa en la preparación de comida*⟩.
Con lo que podemos utilizar el dilema constructivo para concluir: $\langle r \vee s \rangle$, que se interpreta como «La manzana se puede comer o la calabaza se usa en la preparación de comida».

Cuando se usa el criterio inclusivo de la disyunción, en la premisa $p \vee q$, al menos una de las proposiciones es **V**, pero cuando se usa la disyunción exclusiva [página 64], exactamente una de las proposiciones, ya sea p o bien q, es **V**, pero no ambas.

Ejemplo 3.27

Supongamos las premisas:

P1: «O bien paso la materia o repruebo la materia».
P2: «Si paso la materia, entonces viajo de vacaciones».
P3: «Si repruebo la materia, entonces me quedo en casa en vacaciones».

Reescribiendo las premisas y la conclusión en forma simbólica:

P1: $p \oplus q$
P2: $p \rightarrow r$
P3: $q \rightarrow s$
C1: $r \vee s$
C2: $s \vee r$

De las premisas, se puede concluir «Viajo de vacaciones o me quedo en casa en vacaciones».

En la premisa disyuntiva en modo exclusivo $p \oplus q$, solamente una de las proposiciones es **V**, de modo que podemos analizar cada uno de esos casos:

Caso 1: p es **V** y q es **F**. En este caso se puede utilizar PP:

$$\begin{array}{l} p \\ p \rightarrow r \\ \hline \therefore r \end{array}$$

con lo que se puede garantizar que r es verdad, luego utilizando la regla AD [página 63] se puede concluir que $r \vee s$ es verdad.

Caso 2: q es **V** y p es **F**. En este caso se puede utilizar PP:

$$\begin{array}{l} q \\ q \rightarrow s \\ \hline \therefore s \end{array}$$

con lo que se puede garantizar que s es verdad y consecuentemente $s \vee r$ es verdad.

El hecho de que, o bien r es verdad o s es verdad, garantiza que tanto $s \vee r$ como $r \vee s$ es verdad, pero no es posible concluir que ambas proposiciones r y s sean falsas o verdaderas.

3.5 Leyes de De Morgan

Las leyes De Morgan son dos reglas de inferencia que describen cómo se relacionan las operaciones de negación, conjunción y disyunción cuando se aplican a *fbf*. En diferentes fuentes se puede encontrar como «leyes De Morgan» [Cur77, DP17] o «leyes de De Morgan» [Arn89, BA09] o «Teoremas de De Morgan» [CN02], pero se refieren a las mismas leyes en el ámbito de la lógica y el álgebra booleana.

Son relaciones que establecen una equivalencia entre diferentes formas de pueden presentar las *fbf*.

3.5.1 Caso disyuntivo [LMD]

El caso disyuntivo de estas reglas toma como premisa la negación de una disyunción y se garantiza que se puede reescribir como una conjunción, como lo establece la siguiente expresión:

$$\begin{aligned} &\neg(p \vee q) \vdash (\neg p \wedge \neg q) \\ &(\neg p \wedge \neg q) \vdash \neg(p \vee q) \end{aligned} \tag{3.12}$$

En la notación Gentzen se tiene:

$$\frac{\neg(p \vee q)}{\therefore \neg p \wedge \neg q}, \qquad \frac{\neg p \wedge \neg q}{\therefore \neg(p \vee q)}$$

Se observa que la premisa de una expresión es la conclusión de la otra, y viceversa. Esto es así porque ambas expresiones son equivalentes.

Ejemplo 3.28

Sean $p \leftarrow\!\!\shortmid\langle Como\ fruta\rangle$ y $q \leftarrow\!\!\shortmid\langle Como\ carne\rangle$. La premisa $\neg(p \vee q)$ se puede interpretar como «No es verdad que, como fruta o como carne». La ley De Morgan en el caso disyuntivo asegura que de esta expresión se puede concluir $\neg p \wedge \neg q$, que se interpreta como «ni como fruta ni como carne», esto es:

$$\frac{\neg(\langle como\ fruta\rangle \vee \langle como\ carne\rangle)}{\therefore \neg\langle como\ fruta\rangle \wedge \neg\langle como\ carne\rangle}$$

Ahora, si se toma como premisa «ni como fruta ni como carne», se puede concluir que «no es verdad que no como fruta o no como carne».

$$\frac{\neg\langle como\ fruta\rangle \wedge \neg\langle como\ carne\rangle}{\therefore \neg(\langle como\ fruta\rangle \vee \langle como\ carne\rangle)}$$

Las proposiciones p y q representan *fbf* que pueden tener cualquier complejidad, sin embargo deben conservar la forma adecuada para poder aplicar la regla.

Ejemplo 3.29

En los siguientes ejemplos se utiliza la ley de Morgan en su versión disyuntiva:

- $\neg((p \rightarrow q) \vee (r \rightarrow s)) \vdash \neg(p \rightarrow q) \wedge \neg(r \rightarrow s)$
- $\neg((q \wedge \neg r) \vee (\neg p \rightarrow s)) \vdash \neg(q \wedge \neg r) \wedge \neg(\neg p \rightarrow s)$
- $\neg c \wedge \neg(b \wedge a) \vdash \neg(c \vee (b \wedge a))$

3.5.2 Caso conjuntivo [LMC]

El caso conjuntivo de las leyes de De Morgan es muy similar al caso disyuntivo, también se presenta en dos formas:

$$\begin{aligned} &\neg(p \wedge q) \vdash \neg p \vee \neg q \\ &\neg p \vee \neg q \vdash \neg(p \wedge q) \end{aligned} \tag{3.13}$$

y en la notación Gentzen:

$$\frac{\neg(p \wedge q)}{\therefore \neg p \vee \neg q}, \qquad \frac{\neg p \vee \neg q}{\therefore \neg(p \wedge q)}$$

Debe notarse que en una forma la premisa es la conclusión en la otra, y la conclusión de una es la premisa en la otra forma, es por eso que se pueden ver como sustituciones. Donde aparece una expresión que tiene la estructura de la premisa, se puede escribir la conclusión.

Ejemplo 3.30

Sea «No llueve y no hay arcoíris» una expresión proposicional, de donde se pueden obtener las siguientes proposiciones:

$p \longleftrightarrow \langle Llueve \rangle$.

$q \longleftrightarrow \langle Hay\ arcoíris \rangle$.

Al construir la premisa $\neg(p \wedge q)$ se interpreta «no es verdad que llueve y hay arcoíris». La interpretación indica que no ocurre que, en el momento de llover, también esté presente el arcoíris; las dos aseveraciones deben ser ciertas en el mismo momento. Usando la regla de sustitución de De Morgan en su forma conjuntiva se tiene:

$$\frac{\neg(p \wedge q)}{\therefore \neg p \vee \neg q}$$

Recordando el significado de cada literal:

$$\frac{\neg(\langle llueve \rangle \wedge \langle hay\ arcoíris \rangle)}{\therefore \neg\langle llueve \rangle \vee \neg\langle hay\ arcoíris \rangle}$$

Finalmente la conclusión $\neg p \vee \neg q$ se interpreta como «no llueve o no hay arcoíris», indicando que al menos una de las dos proposiciones opositivas ocurra, incluso ambas.

Hay expresiones que permiten utilizar las leyes de De Morgan en más de una ocasión en la misma expresión. Por ejemplo en la premisa

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

se observa que la negación se conforma de una negación, cuyos componentes son expresiones que permiten el uso de alguna ley de De Morgan:

$$\underset{LMC}{\boxed{\neg(\underset{LMC}{\boxed{\neg(p \wedge q)}} \wedge \underset{LMD}{\boxed{\neg(r \vee s)}})}},$$

Aunque se puede aplicar cualquier regla de inferencia en cualquier momento, no existe un método definido para determinar el orden de aplicación cuando hay múltiples opciones. Sin embargo, una buena estrategia es simplificar la expresión reduciendo el número o la complejidad de sus componentes.

Aplicando LMC en la expresión completa:

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

podemos aplicar DN en cada componente de la disyunción:

$$(p \wedge q) \vee (r \vee s).$$

Otro modo de trabajar la expresión es utilizar las reglas de De Morgan en las subexpresiones más internas:

$$\neg((\neg p \vee \neg q) \wedge (\neg r \wedge \neg s)),$$
$$\neg(\neg p \vee \neg q) \vee \neg(\neg r \wedge \neg s).$$

En general, las leyes de De Morgan se aplican a una disyunción o a una conjunción siguiendo los siguientes pasos:

① Cambiar las disyunción por conjunción, o bien la conjunción por disyunción; esto es sustituir $\vee$ por $\wedge$ y $\wedge$ por $\vee$.
② Negar cada componente de la disyunción o conjunción.
③ Negar toda la expresión.

Con este método, es interesante observar que las leyes de De Morgan se pueden aplicar a disyunciones o conjunciones, sin importar si sus componentes están o no negados.

Ejemplo 3.31

Utilizar las leyes de De Morgan en las siguientes expresiones:

- $r \wedge s$
  ① $r \vee s$ ▹ *Cambiar $\wedge$ por $\vee$.*
  ② $\neg r \vee \neg s$ ▹ *Negar cada componente.*
  ③ $\neg(\neg r \vee \neg s)$ ▹ *Negar toda la expresión.*
- $\neg r \vee s$
  ① $\neg r \wedge s$ ▹ *Cambiar $\vee$ por $\wedge$.*
  ② $\neg\neg r \wedge \neg s$ ▹ *Negar cada componente.*
  ③ $\neg(r \wedge \neg s)$ ▹ *Negar toda la expresión.*
- $\neg(\neg q \wedge \neg s)$
  ① $\neg(\neg q \vee \neg s)$ ▹ *Cambiar $\wedge$ por $\vee$.*
  ② $\neg(\neg\neg q \vee \neg\neg s)$ ▹ *Negar cada componente.*
  ③ $q \vee s$ ▹ *Negar toda la expresión.*

3.6 Otras equivalencias notables

Una equivalencia es un sistema de dos proposiciones moleculares construidas con las mismas variables lógicas y que tienen el mismo valor de verdad. Se utiliza el símbolo '$\equiv$' para establecer que la expresión a la izquierda es equivalente a la expresión de la derecha.

Por ejemplo, se puede escribir $p \equiv \neg\neg p$ para expresar que p tendrá el mismo valor [*equivalente*] de verdad que $\neg\neg p$, claramente $\neg\neg p \equiv p$ también es válido.

3.6.1 Leyes de idempotencia

Idempotencia disyuntiva [LID]

La idempotencia disyuntiva es una disyunción de la forma $p \vee p$, es decir, que ambos componentes son la misma proposición. Cuando ocurre de esta manera, se puede reducir la expresión para tener una sola proposición p.

$$p \vee p \equiv p \tag{3.14}$$

Idempotencia conjuntiva [LIC]

De manera similar que el caso disyuntivo, al hacer una conjunción con la misma proposición, el valor de verdad que resulta es idéntico al valor de verdad de la proposición.

$$p \wedge p \equiv p \tag{3.15}$$

3.6.2 Leyes conmutativas

La conmutatividad en las expresiones lógicas de conjunción o disyunción se refiere a que el resultado de ciertas operaciones no varía al cambiar el orden de sus términos o elementos.

La ley conmutativa basa su interpretación en la misma definición de la conjunción y la disyunción. Si se tiene una proposición verdadera de la forma $p \wedge q$, se concluye $q \wedge p$. Por otro lado, si se tiene una proposición de la forma $p \vee q$, se concluye $q \vee p$.

En el caso de la disyunción se tiene:

$$p \vee q \equiv q \vee p \tag{3.16}$$

Para el caso conjuntivo se tiene:

$$p \wedge q \equiv q \wedge p \tag{3.17}$$

3.6.3 Leyes asociativas

La ley asociativa es un principio o propiedad que se aplica en lógica y en diferentes áreas de las matemáticas, que en esencia establece que el agrupamiento de elementos en una operación binaria (como la disyunción o la conjunción) no cambia el resultado final, independientemente de cómo se agrupen esos elementos.

Para el caso de la disyunción, la ley asociativa se expresa:

$$(p \vee q \vee r) \equiv (p \vee (q \vee r)) \equiv ((p \vee q) \vee r) \tag{3.18}$$

Para la conjunción, se expresa:

$$(p \wedge q \wedge r) \equiv (p \wedge (q \wedge r)) \equiv ((p \wedge q) \wedge r) \tag{3.19}$$

El caso general, es similar tanto para la disyunción como para la conjunción. Analizaremos aquí el caso conjuntivo. La generalización considera cero, una o incluso más proposiciones. En el caso de la conjunción tenemos:

$$\bigwedge_{i=0}^{n} p_i = (\mathbf{V} \wedge (p_1 \wedge (\ldots \wedge (p_{n-1} \wedge p_n)\ldots))) \tag{3.20}$$

Al observar detenidamente la expresión, notamos que cuando $i = 0$ significa que no hay proposiciones con las que hacer la conjunción, en este caso el valor de verdad de la generalización es **V**. Por otro lado, cuando $i > 0$, es decir, hay al menos una proposición, ya se puede utilizar la conjunción habitual para calcular el valor de verdad, que será **F** cuando se presente alguna proposición de valor **F**, pero cuando todas son **V**, también lo es la conjunción generalizada.

Para entender con detalle cómo funciona esta conjunción generalizada, veamos lo que sucede en los primeros casos:

- Cuando no hay proposiciones el valor es **V**. Nada más que hacer.

- Cuando se tiene una proposición, la conjunción generalizada es:

$$\bigwedge_{i=0}^{1} p_i = \mathbf{V} \wedge p_1 = p_1,$$

que es de valor **V** cuando p_1 es **V**, y **F** cuando p_1 es **F**.
- Cuando $i = 2$ hay dos proposiciones, por lo que:

$$\bigwedge_{i=0}^{2} p_i = ((\mathbf{V} \wedge p_1) \wedge p_2) = ((\bigwedge_{j=0}^{1} p_j) \wedge p_2),$$

Si suponemos que p_1 es **F**, no hay necesidad de verificar el resto, pues $(\mathbf{V} \wedge p_1) \vdash \mathbf{F}$. Ahora, suponiendo que p_1 es **V**, con el fin de averiguar lo que sucede con p_2, vemos que el valor de $\bigwedge_{i=0}^{2} p_i$ es el mismo que el valor de p_2, si es **F**, la conjunción múltiple también lo es; si el **V**, la conjunción múltiple también lo es.
- Cuando tenemos 3 proposiciones:

$$\bigwedge_{i=0}^{3} p_i = (((\mathbf{V} \wedge p_1) \wedge p_2) \wedge p_3) = ((\bigwedge_{j=0}^{2} p_j) \wedge p_3).$$

Utilizando la regla de agregación:

$$\bigwedge_{i=0}^{3} p_i = p_3 \wedge \left(\bigwedge_{j=0}^{2} p_j \right).$$

En general, notaremos que dada una secuencia no vacía $p_1, p_2, \ldots, p_k$ de proposiciones, el valor de $\bigwedge_{i=0}^{2} p_i$ es **V** si todas ellas son **V**, y será **F** cuando exista al menos una proposición p_i, con $1 \leq k$, que sea **F**. Así

$$\bigwedge_{i=0}^{k} p_i = \begin{cases} \mathbf{V} & \text{si } i = 0 \\ p_k \wedge \left(\bigwedge_{i=0}^{i-1} p_i \right) & \text{si } i > 0 \end{cases} \tag{3.21}$$

3.6.4 Leyes distributivas

La ley distributiva es un principio fundamental en lógica y en general en matemáticas, que describe cómo se relacionan dos operaciones, generalmente la conjunción y la disyunción, en relación con la combinación de proposiciones en una expresión molecular. Esta ley establece cómo se distribuye un término de enlace sobre dos o más términos dentro de un paréntesis.

Una proposición molecular en la que se puede aplicar la ley distributiva tiene la forma:

$$\langle \_\_ \rangle_{\text{TED}} (\langle \_\_ \rangle_{\text{ENL}} \langle \_\_ \rangle)$$

donde:

TED: Es el término de enlace que se distribuye entre las expresiones dentro del paréntesis.

ENL: En un término de enlace.

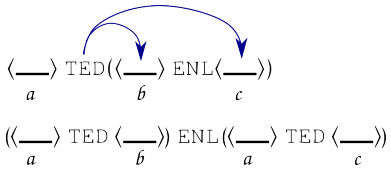


Figura 3.3: La propiedad distributiva permite que se forme una proposición molecular con una proposición a y con cada una de las proposiciones dentro del paréntesis b y c usando el enlace TED, unidas todas ellas con el término de enlace ENL.

La distribución de la disyunción sobre una conjunción se expresa:

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r) \tag{3.22}$$

La conjunción se puede distribuir sobre la disyunción de la siguiente manera:

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r) \tag{3.23}$$

La implicación también es una expresión con propiedad distributiva sobre la disyunción y la conjunción. El caso de la implicación sobre la disyunción se expresa:

$$p \rightarrow (q \vee r) \equiv (p \rightarrow q) \vee (p \rightarrow r) \tag{3.24}$$

El caso de la implicación sobre la conjunción se puede escribir como:

$$p \rightarrow (q \wedge r) \equiv (p \rightarrow q) \wedge (p \rightarrow r) \tag{3.25}$$

3.6.5 Bicondicional [BI]

La bicondicional es una proposición molecular que relaciona dos proposiciones, también se conoce como «si y sólo si» y se utiliza para expresar que dos proposiciones tienen el mismo valor de verdad, usualmente se utiliza el símbolo '↔' con el patrón ⟨___⟩ ↔⟨___⟩.

Ejemplo 3.32

La expresión «La figura es un triángulo si y sólo si tiene tres ángulos» se puede simbolizar como:

$p \leftarrow \langle$*La figura es un triángulo*$\rangle$.

$q \leftarrow \langle$*La figura tiene tres ángulos*$\rangle$.

Con lo que se puede escribir $p \leftrightarrow q$.

La bicondicional es equivalente a otras expresiones construidas con implicaciones, conjunciones y negaciones:

$$p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p) \tag{3.26}$$

$$p \leftrightarrow q \equiv \neg((p \vee q) \wedge \neg(p \wedge q)) \tag{3.27}$$

3.6.6 Ley del complemento

Las leyes del complemento describen cómo las negaciones de las proposiciones se relacionan entre sí. Estas leyes se emplean frecuentemente en la reducción de expresiones complejas.

Ya anteriormente se han enunciado algunas expresiones que relacionan los complementos, como las leyes de De Morgan [páginas 77 y 78], aquí se expresan dos nuevas leyes.

La disyunción de una proposición con su negación es equivalente a **V**:

$$p \vee \neg p \equiv \mathbf{V}. \tag{3.28}$$

La conjunción de una proposición con su negación es **F**:

$$p \wedge \neg p \equiv \mathbf{F}. \tag{3.29}$$

Ejemplo 3.33

Escribe una versión reducida de la expresión $(p \wedge \neg q) \vee (p \wedge q) \vee \neg(p \wedge q)$.

1. Usamos la ley de De Morgan en el último término: $(p \wedge \neg q) \vee (p \wedge q) \vee (\neg p \vee \neg q)$.
2. Usamos la distribución en los primeros dos términos: $(p \wedge (\neg q \vee q)) \vee (\neg p \vee \neg q)$.
3. Aplicamos la ley del complemento en $(\neg q \vee q)$: $(p \wedge \mathbf{V}) \vee (\neg p \vee \neg q)$.
4. Aplicamos el primer caso de la asociatividad en la conjunción: $(p \vee (\neg p \vee \neg q))$.
5. Aplicamos la asociatividad: $((p \vee \neg p) \vee \neg q)$.
6. Aplicamos la ley del complemento: $(\mathbf{V} \vee \neg q)$.
7. Por la disyunción se tiene **V**.

Así $(p \wedge \neg q) \vee (p \wedge q) \vee \neg(p \wedge q)$ es lógicamente equivalente a **V**.

Ejercicios

1. Determina qué principio fundamental de la lógica se emplea en cada situación. Marca con ✗ dentro del círculo que corresponda.
   a) *Ejemplo.* «Un objeto no puede ser idéntico a otro objeto y ser diferente de ese mismo objeto al mismo tiempo»
      ◯ Principio de Identidad;
      ◯ Principio del tercero excluido
      (✗) Principio de la no contradicción
   b) «Un número dado es o par o impar, no puede ser ambos a la vez.»
      ◯ Principio de Identidad;
      ◯ Principio del tercero excluido
      ◯ Principio de la no contradicción
   c) «Si un triángulo tiene tres lados, entonces tiene tres lados. Si tiene cuatro lados, ya no es un triángulo.»
      ◯ Principio de Identidad;
      ◯ Principio del tercero excluido
      ◯ Principio de la no contradicción
   d) «Un círculo es un conjunto de puntos equidistantes de su centro. Un círculo no puede ser, al mismo tiempo, un conjunto de puntos equidistantes de su centro y no serlo.»
      ◯ Principio de Identidad;
      ◯ Principio del tercero excluido
      ◯ Principio de la no contradicción
   e) «Un objeto en movimiento continuará en movimiento a menos que una fuerza actúe sobre él. Un objeto en reposo permanecerá en reposo a menos que una fuerza actúe sobre él. No existe una tercera opción entre movimiento y reposo para un objeto.»
      ◯ Principio de Identidad;
      ◯ Principio del tercero excluido
      ◯ Principio de la no contradicción
2. Aplica la regla de doble negación [p. 62] a cada una de las siguientes oraciones, luego da una interpretación adecuada en español.
   a) *Ejemplo.* No sucede que el estudiante no se preparó para el examen.
   Se puede escribir $\neg\neg\langle$*El estudiante se preparó para el examen*$\rangle$, luego es equivalente a $\langle$*El estudiante se preparó para el examen*$\rangle$.
   b) No pasa que el coche está estacionado en el garaje.
   c) No ocurre que el niño no está desatendido.
   d) El restaurante no está abierto los fines de semana.
   e) No acontece que el equipo no haya ganado el partido.
   f) No es cierto que sea inhumano.
3. Las siguientes expresiones pueden resultar un poco confusas, en el sentido de no saber si se puede aplicar en ella cierta regla de inferencia. Decide si la expresión puede ser utilizada en la regla propuesta. En caso de que no se pueda, explicar el motivo.

a) *Ejemplo.* ¿Se puede aplicar la regla DI [p. 67] con la premisa $p \to q \wedge r$ para demostrar $p \to q$?
No, debido a que la jerarquía en los términos de enlace indica que la expresión debe interpretarse como $p \to (q \wedge r)$, que es una expresión implicativa, y la regla DI exige una conjunción.

b) ¿Es correcta la conclusión ⟨*como carne o como verduras*⟩ a partir de la premisa ⟨*Como carne*⟩?

c) ¿A partir de la premisa ⟨*el león es canino o felino*⟩ se puede concluir que ⟨*El león es felino*⟩?

d) Si se tienen como premisas ⟨*Ana lava ropa el domingo*⟩ y ⟨*Ana trabaja el lunes*⟩, ¿se puede concluir ⟨*Ana lava ropa el domingo o trabaja el lunes*⟩?

e) ¿Es correcta la conclusión ⟨$x+2>5$⟩ cuando se sabe ⟨$x+2 \leq 5 \vee 2x > 15$⟩?

4. Enlista las premisas que se pueden obtener en cada párrafo, descubre cuál es la conclusión. Traduce las premisas en expresiones simbólicas y encuentra una demostración paso a paso.

a) *Ejemplo.* «Si no hay nuevos platillos, entonces no se modifica el menú. Si no se modifica el menú, entonces no se contratan más cocineros. O se contratan más cocineros o se remodela el restaurante. Pero no se remodela el restaurante. Por lo tanto si hay nuevos platillos».

$p \leftarrow\!\shortmid$ ⟨*Hay nuevos platillos*⟩
$q \leftarrow\!\shortmid$ ⟨*Se modifica el menú*⟩
$r \leftarrow\!\shortmid$ ⟨*Se contratan más cocineros*⟩
$s \leftarrow\!\shortmid$ ⟨*Se remodela el restaurante*⟩

Premisas:

$\mathtt{P}_1$: $\neg p \to \neg q$
$\mathtt{P}_2$: $\neg q \to \neg r$
$\mathtt{P}_3$: $r \vee s$
$\mathtt{P}_4$: $\neg s$

Se desea concluir p.

Paso	Regla usada	Explicación		Conclusión:
①	TP($\mathtt{P}_3$,$\mathtt{P}_4$)	$r \vee s, \neg s \vdash r$	$\mathtt{C}_1$:	r
②	TT($\mathtt{P}_2$,$\mathtt{C}_1$)	$r, \neg q \to \neg r \vdash s$	$\mathtt{C}_2$:	q
③	TT($\mathtt{C}_2$, $\mathtt{P}_1$)	$q, \neg p \to \neg q \vdash p$	$\mathtt{C}_3$:	p ■

b) «Si mastica al nacer, entonces tiene dientes desde que nace. El perro toma leche materna si es mamífero. Si toma leche materna, entonces no tiene dientes desde que nace. El perro es un mamífero».

c) «Si no tengo hambre, entonces quiero dormir. Si tengo hambre, entonces estoy nervioso y de mal humor. Pro no es cierto que estoy nervioso y de mal humor. Por lo tanto no tengo hambre y quiero dormir».

5. En los siguientes enunciados, determina la conclusión es una consecuencia lógica de las premisas, o no lo es.

a) *Ejemplo.* «Antonio estudia o trabaja. Por lo tanto trabaja».
◯ Sí es (✗) No es .

b) «Si es un número par, entonces es divisible por dos. No es divisible por dos. Por lo tanto no es un número par»
◯ Sí es ◯ No es .

c) Si camina entonces tiene pies. Camina. Por lo tanto no tiene pies.
◯ Sí es ◯ No es .

d) Si ha hecho mucho calor entonces llueve. Hoy es cumpleaños de Tito y no llueve. Por lo tanto no ha hecho mucho calor.
◯ Sí es ◯ No es .

e) Si tiene cuatro lados, entonces es un cuadrado. Es un cuadrado o un rectángulo. No es un rectángulo. Por lo tanto no tiene cuatro lados.
◯ Sí es ◯ No es .

6. Escribe una demostración paso a paso, usando las reglas de inferencia de esta lección para llegar a la demostración solicitada en cada caso:

a) *Ejemplo.* Se desea demostrar $\neg\neg s$, teniendo las premisas:

$\mathtt{P}_1$: q
$\mathtt{P}_2$: $\neg r \to s$
$\mathtt{P}_3$: $q \to \neg r$

Paso	Regla usada	Explicación		Conclusión:
①	PP($\mathtt{P}_1$,$\mathtt{P}_3$)	$q, q\to\neg r \vdash \neg r$	$\mathtt{C}_1$:	$\neg r$
②	PP($\mathtt{C}_1$,$\mathtt{P}_2$)	$\neg r, \neg r\to s \vdash s$	$\mathtt{C}_2$:	s
③	DN($\mathtt{C}_2$)	$s \vdash \neg\neg s$	$\mathtt{C}_3$:	$\neg\neg s$ ■

b) Se desea demostrar u, teniendo las premisas:

$\mathtt{P}_1$: $p \wedge \neg t$
$\mathtt{P}_2$: $s \to t$
$\mathtt{P}_3$: $s \vee q$
$\mathtt{P}_4$: $q \vee p \to u$

c) Se desea demostrar p, teniendo las premisas:

$\mathtt{P}_1$: s
$\mathtt{P}_2$: $s \to (t \wedge r)$
$\mathtt{P}_3$: $q \vee p$
$\mathtt{P}_4$: $q \to \neg t$

d) Se desea demostrar p, teniendo las premisas:

$\mathtt{P}_1$: $q \to s \vee p$
$\mathtt{P}_2$: $\neg r \wedge \neg t \to q$
$\mathtt{P}_3$: $t \to \neg u$
$\mathtt{P}_4$: $u \wedge \neg s$
$\mathtt{P}_5$: $u \to \neg r$

7. En cada ejercicio, determina cuál es la conclusión correcta a partir de las premisas dadas.

a) *Ejemplo.*
$\mathtt{P}_1$: ⟨*Si descanso, entonces no me duelen los pies*⟩.
$\mathtt{P}_2$: ⟨*Si juego al futbol o beisbol, entonces me duelen los pies*⟩.
$\mathtt{P}_3$: ⟨*Juego basquetbol y beisbol*⟩

b) $\mathtt{P}_1$: ⟨*Si Rita es feliz, entonces hace fiesta*⟩.
$\mathtt{P}_2$: ⟨*Si llueve, entonces Rita no hace fiesta*⟩.
$\mathtt{P}_3$: ⟨*Llueve y hace frio*⟩.

c) $\mathtt{P}_1$: ⟨*Si Antonio llegó antes, entonces se desespera*⟩.
$\mathtt{P}_2$: ⟨*Antonio se desespera y se aburre*⟩.
$\mathtt{P}_3$: ⟨*Antonio llegó antes*⟩.
$\mathtt{P}_4$: ⟨*Si Antonio se aburre, entonces juega con la tierra*⟩.

d) En cada ejercicio, decide cuál regla de inferencia puede ser aplicada ante las premisas dadas:

1) *Ejemplo.* Con las premisas:
$\mathtt{P}_1$: $p \to q$
$\mathtt{P}_2$: p
◯ *M. Tollendo tollens* (✗) *M. Ponendo ponens* ◯ *M. Ponendo tollens*

2) Con las premisas:
$\mathtt{P}_1$: $\neg p \to (q \wedge r)$
$\mathtt{P}_2$: $\neg\neg(q \wedge r)$
◯ *M. Tollendo tollens* ◯ *M. Ponendo ponens* ◯ *M. Ponendo tollens*

3) Con las premisas:
$\mathtt{P}_1$: ⟨*Si se cierra la compuerta, entonces no entra el agua*⟩
$\mathtt{P}_2$: ⟨*Entra el agua*⟩
◯ *M. Tollendo tollens* ◯ *M. Ponendo ponens* ◯ *M. Ponendo tollens*

4) Con las premisas:
$\mathtt{P}_1$: ⟨*O juegas o te quedas sentado*⟩
$\mathtt{P}_2$: ⟨*No juegas*⟩
◯ *M. Tollendo tollens* ◯ *M. Ponendo ponens* ◯ *M. Ponendo tollens*

5) Con las premisas:
$\mathtt{P}_1$: ⟨*Si se acciona el botón izquierdo del mouse, entonces se abre una aplicación; o si se acciona el botón derecho del mouse, entonces se despliega el menú de opciones*⟩.
$\mathtt{P}_2$: ⟨*No es verdad que si se acciona el botón izquierdo del mouse, entonces se abre una aplicación*⟩.
◯ *M. Tollendo tollens* ◯ *M. Ponendo ponens* ◯ *M. Ponendo tollens*

8. Escribir una demostración a partir de las premisas dadas:

a) *Ejemplo.* Demostrar $\neg s$ a partir de las premisas:
$\mathtt{P}_1$: $(p \to s) \wedge q$
$\mathtt{P}_2$: $s \to r$
$\mathtt{P}_3$: $(p \to s) \to \neg r$

$\mathtt{C}_1$: $p \to s$	$(p \to s) \wedge q \vdash (p \to s), q$	▹ *Usando DI*($\mathtt{P}_1$)
$\mathtt{C}_2$: q	$(p \to s) \wedge q \vdash (p \to s), q$	▹ *Usando DI*($\mathtt{P}_1$)
$\mathtt{C}_3$: $\neg r$	$(p \to s), (p \to s) \to \neg r \vdash \neg r$	▹ *Usando PP*($\mathtt{C}_1$,$\mathtt{P}_3$)
$\mathtt{C}_4$:$\neg s$	$(s \to r), \neg r \vdash \neg s$ ■	▹ *Usando TT*($\mathtt{P}_2$,$\mathtt{C}_3$)

b) Con las siguientes premisas demostrar ⟨*hay vacaciones*⟩:
$\mathtt{P}_1$: Vamos a la playa.
$\mathtt{P}_2$: Si no llueve, entonces hace sol.
$\mathtt{P}_3$: Si llueve, entonces no vamos a la playa.
$\mathtt{P}_4$: hace sol y hay vacaciones.

c) Con las siguientes premisas demostrar que ⟨*hay descuento*⟩:
$\mathtt{P}_1$: si no ha salido el sol entonces es temprano.
$\mathtt{P}_2$: Si es temprano, entonces está abierto y hay descuento

P_3: Si salio el sol, entonces hay mucho calor.
P_4: No hay mucho calor.

d) Con las siguientes premisas demostrar que $b = 12$:
P_1: $c < b \rightarrow b = 12$.
P_2: $(b > 6 \vee c < 2) \rightarrow (c < b \vee a = 9)$.
P_3: $a = 9 \rightarrow \neg(c < 2 \vee b > 6)$.
P_4: $b = 3 \rightarrow b > 6$.
P_5: $b = 3 \vee b < a$.
P_6: $b < a \rightarrow c < 2$.

e) Demostrar $\neg(2\omega \neq 8 \wedge \omega \neq \beta)$ a partir de las premisas:
P_1: $5\omega = \alpha \wedge 4\omega = 12$.
P_2: $\omega = \beta \rightarrow \omega + 2\delta = 16$.
P_3: $5\omega = \alpha \leftrightarrow \omega = \beta$.

9. Utiliza el silogismo hipotético, el dilema constructivo y las leyes de De Morgan para resolver los siguientes ejercicios:

a) *Ejemplo.* ¿Qué conclusión puedes obtener de las siguientes premisas?
1) «Si cumplo a nos, entonces hay fiesta»
2) «Si hay fiesta, entonces hay invitados»

La declaración «Si cumplo años, entonces hay fiesta» se puede simbolizar como $(p \rightarrow q)$, haciendo:
1) $p \leftarrowtail \langle \textit{Cumplo años} \rangle$
2) $q \leftarrowtail \langle \textit{Hay fiesta} \rangle$.

La declaración «Si hay fiesta, entonces hay invitados» puede ser simbolizada como $q \rightarrow r$, ya que $q \leftarrowtail \langle \textit{Hay fiesta} \rangle$ y haciendo $r \leftarrowtail \langle \textit{Hay invitados} \rangle$. Ahora, si tenemos las premisas $(p \rightarrow q), (q \rightarrow q)$, por el silogismo hipotético se puede concluir $(p \rightarrow r)$, lo que significa que:

«Si cumplo años, entonces hay invitados».

b) ¿Qué conclusión puedes obtener de las siguientes premisas?
1) «Si estudias, entonces apruebas».
2) «Si apruebas, entonces consigues el certificado».

c) ¿Qué acciones puedes asegurar que realizarás en estas premisas?
1) «Tienes un perro o un gato».
2) «Si tienes un perro, entonces paseas y juegas a la pelota».
3) «Si tienes un gato, entonces paseas solo»

d) ¿Qué artículos vas a comprar según estas premisas?
1) «Te gusta el chocolate o la vainilla».
2) «Si te gusta el chocolate, entonces compras chocolate, pan y fresas».
3) «Si te gusta la vainilla, entonces compras canela, plátanos, leche y azúcar».

e) Usando las reglas de inferencia de De Morgan, ¿puedes demostrar que las siguientes afirmaciones son lógicamente equivalentes?
1) «No es cierto que María y Alfonso lleguen temprano».
2) «Es falso que ni María llegue temprano ni Juan llegue temprano».

10. Usa las leyes de De Morgan para transformar las siguientes expresiones:

a) *Ejemplo.* $(a \vee b)$
1) $(a \wedge b) \triangleright$ *Cambiar* $\wedge$ *por* $\vee$, *y* $\vee$ *por* $\wedge$.
2) $(\neg a \wedge \neg b) \triangleright$ *Negar cada componente*.

3) $\neg(\neg a \wedge \neg b)$ ▹ *Negar toda la expresión.*

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

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

d) $\neg(\neg(p \oplus q) \vee \neg(q \uparrow r))$

11. Escribe la tabla de verdad de las siguientes expresiones lógicas:

a) `Ejemplo.` $(p \downarrow q) \oplus \neg p$

p	q	$(p \downarrow q)$	$\neg p$	$(p \downarrow q) \oplus \neg p$
V	**V**	**F**	**F**	**F**
V	**F**	**F**	**F**	**F**
F	**V**	**F**	**V**	**V**
F	**F**	**V**	**V**	**F**

b) $(p \wedge (q \leftrightarrow r)) \oplus (\neg r \rightarrow q)$

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

d) $(p \leftarrow (q \wedge r)) \nleftrightarrow (\neg q \wedge (p \top q))$

12. Usar la tabla de Verdad para determinar la equivalencia entre los siguientes pares de expresiones:

a) `Ejemplo.` $((p \downarrow q) \oplus \neg p)$ con $(p \nleftarrow q)$.

p	q	$(p \nleftarrow q)$	$(p \downarrow q)$	$\neg p$	$(p \downarrow q) \oplus \neg p$
V	**V**	**F**	**F**	**F**	**F**
V	**F**	**F**	**F**	**F**	**F**
F	**V**	**V**	**F**	**V**	**V**
F	**F**	**F**	**V**	**V**	**F**

b) $\neg(\neg p \vee \neg q)$ con $(p \wedge q)$.

c) $(\neg p \leftrightarrow (q \leftarrow \neg(q \oplus p)))$ con $p \leftrightarrow (q \vee \neg p)$.

d) $((r \leftarrow \neg t) \wedge (t \leftrightarrow s)) \leftrightarrow (t \downarrow \neg s)$ con $(\neg r \leftrightarrow t) \vee \neg s$.

13. Mediante el uso de tablas de Verdad, determina si las siguientes expresiones son tautologías.

a) `Ejemplo.` $p \wedge (q \vee r) \leftrightarrow ((p \wedge q) \vee (p \wedge r))$

p	q	r	$p \wedge (q \vee r)$	$((p \wedge q) \vee (p \wedge r))$	$p \wedge (q \vee r) \leftrightarrow ((p \wedge q) \vee (p \wedge r))$
V	**V**	**V**	**F**	**F**	**V**
V	**V**	**F**	**F**	**F**	**V**
V	**F**	**V**	**F**	**F**	**V**
V	**F**	**F**	**F**	**F**	**V**
F	**V**	**V**	**F**	**F**	**V**
F	**V**	**F**	**V**	**V**	**V**
F	**F**	**V**	**V**	**V**	**V**
F	**F**	**F**	**V**	**V**	**V**

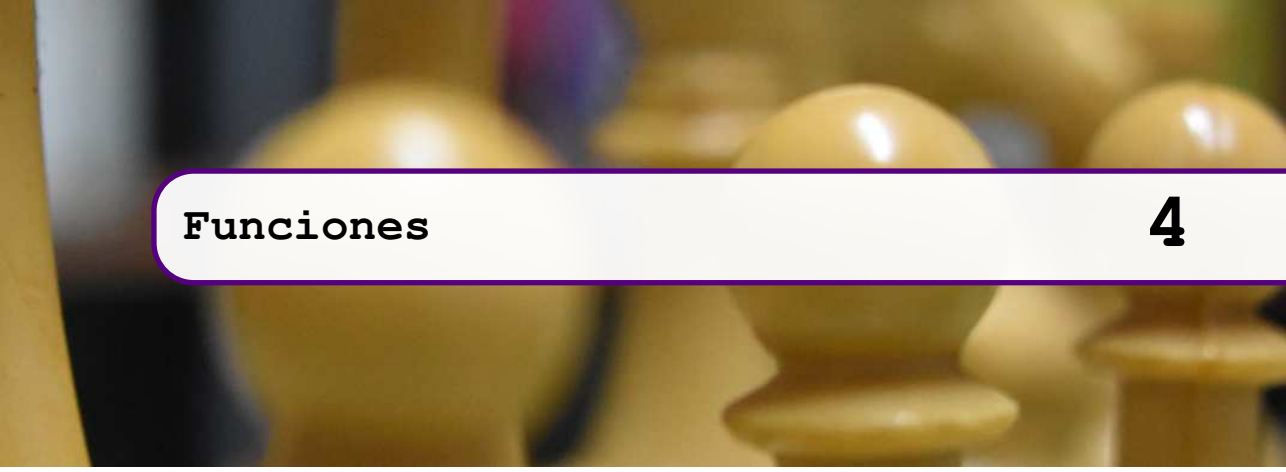
b) $(p \wedge q) \vee (\neg p \wedge q) \vee (p \wedge \neg q) \vee (\neg p \wedge \neg q)$

c) $(p \rightarrow q) \wedge (q \rightarrow r) \wedge (r \rightarrow p)$

d) $(p \leftrightarrow q) \wedge (q \leftrightarrow r) \wedge (r \leftrightarrow p)$

II

Cómputo de predicados



Funciones 4

4.1 Funciones

En el capítulo 2 se introdujo el concepto de «proposición» como una declaración con sujeto y predicado de la cual se puede decir **F** o **V** [ver página 33].

Se mencionó también que una proposición atómica es una expresión indivisible, porque al quitar algo de esa expresión pierde el sentido lógico y gramatical [ver página 35].

El cálculo de predicados, también llamado «lógica de predicados»[AA06] es una parte de la lógica matemática y la teoría de la computación que se utiliza para analizar y formalizar afirmaciones y proposiciones que involucran predicados y cuantificadores. En otras palabras, es un sistema formal que se utiliza para describir y analizar el razonamiento sobre declaraciones genéricas reutilizables que pueden estar sujetas a cuantificadores como «para todo» o «existe un».

«Cómputo de predicados» es un término que es sinónimo de «cálculo de predicados», pero tiene sin dudas una fuerte connotación de programación de computadoras. En síntesis, permite crear y utilizar funciones computacionales para generalizar proposiciones, expresando y manipulando expresiones lógicas complejas haciendo uso de los elementos de un lenguaje de programación, en este caso de `Python`.

☞ Para instrucciones de cómo instalar `Python` en una computadora personal y cómo utilizar un entorno de desarrollo integrado, vea el apéndice A en la página 223.

4.1.1 Variables y símbolos lógicos

Ya anteriormente se introdujeron lineamientos para tratar proposiciones utilizando variables lógicas [ver página 46], pero en esta sección se utilizará el lenguaje computacional para tratar los elementos del sistema lógico, de modo que las variables, en este sentido, siguen las reglas de escritura de `Python`, que es el lenguaje de programación usado en este libro. Sin embargo, si se utilizara otro lenguaje de programación, entonces se deben observar las reglas de ese lenguaje.

En programación, una variable es el nombre de un contenedor que se utiliza para almacenar y representar datos en la memoria de un programa. Ese nombre es una forma de llamar a un valor, lo que facilita la manipulación y el acceso a ese dato a lo largo del programa. Una variable pueden referenciar datos de diferentes tipos de información, como números, texto, valores booleanos y funciones, entre otros.

En el ámbito de la lógica, se utilizan variables para almacenar el valor de verdad asociado con alguna proposición.

4.1.2 Creación de variables

Computacionalmente, siguiendo las reglas de `Python`, una variable se crea siguiendo las siguientes normas:

1. Se deben escribir en una sola palabra, sin espacio en blanco.
2. Deben iniciar con un símbolo entre `a ... z|A...Z`.
3. Pueden incluir los símbolos `a ... z|A...Z |0...9|_`.
4. Deben ser nombres significativos para el propósito de la variable, pero no muy largos para que no sea engorroso escribirlos una y otra vez.
5. No se pueden elegir los nombres de las palabras clave en `Python`, como `def`, `if`, `return`, entre otras.

Para crear una variable, solamente se debe escribir el nombre de la variable e instanciarla con un valor. La variable ahora tendrá el mismo tipo que el tipo del valor asignado.

4.1.3 Instanciación de variables

Antes de utilizar las variables, deben ser instanciadas, esto significa que se debe asociar un valor a la variable, de no hacerlo, se activa un mensaje de error que indica que el nombre `p` no está definido:

```
>>> p
Traceback (most recent call last):
 File "<stdin>", line 1, in "<module>"
NameError: name 'p' is not defined
>>>
```

En `Python`, para asignar un valor a una variable se utiliza el operador de asignación '=', que en `Python` se puede leer como «significa» o «vale». Este símbolo puede al principio causar confusión e interpretarlo como «es igual a», pero para hacer esto, `Python` utiliza el símbolo '==='. Dos símbolos '=' seguidos significan comparación por igualdad, un solo símbolo significa asignación. Cuando se asigna un valor a una variable, el identificador siempre está en el lado izquierdo y en el lado derecho un valor, que puede ser una constante, una función o incluso el nombre de otra variable ya instanciada.

```
>>> p = True
>>>
```

Debido a que en cómputo de predicados estamos más interesados en calcular el valor de verdad de una expresión en lugar de una interpretación en el lenguaje natural, no es tan importante mantener la interpretación del símbolo. Así, `p = True` significa que una variable `p` está asociada con el valor de verdad **V**, sin importar qué interpretación tenga `p` en el lenguaje natural.

A pesar de lo anterior, siempre es posible crear variables con un significado más descriptivo como:

```
>>> la_casa_esta_ocupada = True
>>> la_casa_esta_en_venta = True
>>> p = la_casa_esta_ocupada
>>> q = la_casa_esta_en_venta
>>>
```

`Python`, así como todos los lenguajes de programación, utilizan la memoria de la computadora para almacenar datos, como estos datos son de diferente naturaleza, se requiere diferente cantidad de memoria para almacenar cada datos de diferente tipo. Para operaciones lógicas, `Python` tiene un tipo especial llamado `bool`, contracción de «booleano», que viene del apellido de George Boole [ver página 23].

Una forma en la que se puede consultar el tipo de dato de una variable es con el comando `type`, que al aplicarlo a una variable, devuelve el tipo de dato con el que ha sido relacionada en el momento de la última instanciación.

```
>>> p = True
>>> type(p)
<class 'bool'>
>>>
```

En programación, una función es un segmento de código que es independiente y reutilizable, y que tiene un propósito particular. Las funciones se utilizan para organizar el código en partes manejables, eso tiene dos efectos, uno es que facilita la comprensión del código y el propósito, lo que es benéfico para las personas que utilizan el código. El otro beneficio es que facilita el mantenimiento y reutilización del código.

El comportamiento de las funciones computacionales en general es el mismo: toman alguna cantidad de parámetros como datos de entrada, realizan algunas operaciones con ellos y según convenga, devuelven un resultado como información de salida.

En `Python`, las funciones se escriben siguiendo el formato

```
def ⟨nombreFuncion⟩(⟨listaParametros⟩):
    ⟨expresiones⟩
    return ⟨valorDevuelto⟩
```

Donde:

'def' es la palabra clave que indica que se está creando una función.

'nombreFuncion' es el nombre asignado a la función. Es un identificador iniciado con letra, no debe contener espacios ni símbolos especiales, excepto _.

'listaParametros' son los parámetros que la función recibe como entrada. Se puede tener cualquier número de parámetros o ninguno.

':' Se escribe ':' para indicar el inicio del cuerpo de la función, pero también el inicio de un bloque de instrucciones que pertenecen al mismo ámbito.

'*expresiones*' En el cuerpo de la función, se realizan las operaciones necesarias utilizando los parámetros y posiblemente otras variables.

'return' es opcional y se utiliza para devolver un valor como resultado de la función. Si no se incluye, la función devuelve `None` de manera implícita.

'valorDevuelto' si la palabra `return` está presente, el valor devuelto se presenta en forma de una constante o una variable ya instanciada, también puede ser la palabra reservada `None` que indica la ausencia de valor.

Aquí es un buen momento para aclarar la diferencia entre el nombre de la función y la aplicación de la función, porque al menos computacionalmente, hay una gran diferencia entre el nombre de la función y su aplicación. Tomemos por ejemplo la siguiente definición:

```
def foo(x: bool, y: bool) -> bool:
  """
  cuerpo de la función donde se crea e instancía
  la variable z
  """
  return z
```

4.1.4 Clasificación de funciones

Con base en el tipo de variables formales y el tipo del valor devuelto, las funciones computacionales pueden ser clasificadas desde lo más general a lo más particular en:

Funcional: Es una función que puede recibir como argumento una o más funciones, y también puede devolver como resultado una nueva función. Este tipo de funciones también se conocen como «funciones de orden superior» [Mey90]. En los capítulos 7 [página 149] y el capítulo 8 [página 167] hay ejemplos de este concepto y se analizan con mayor detalle.

Función: Es un segmento de código que es independiente y reutilizable, y realiza una tarea específica. Recibe cero o más parámetros de entrada y posiblemente devuelve como salida un valor que puede ser de cualquier tipo, pero asignado de acuerdo al cómputo dentro de la función.

Predicado: Es una función que se define con 0 o más variables formales [parámetros], no necesariamente de tipo booleano; pero el valor de salida siempre es de tipo booleano.

Función proposicional: También conocida como función lógica, es un tipo de predicado definido con variables formales de tipo booleano y como característica distintiva, siempre devuelve una proposición, es decir, un valor booleano. Se caracterizan porque en el procedimiento interno, se utiliza al menos un término de enlace de la misma aridad[1].

Función proposicional primitiva: También conocidas como funciones lógicas primiticas. Es el tipo de función proposicional que modela el comportamiento de un término de enlace. Hay 4 funciones lógicas primitivas de aridad 1; hay 16 funciones lógicas primitivas de aridad 2; en general hay 4^a funciones lógicas primitivas, de aridad a.

[1] Aridad : Número de variables formales con los que se define una función.

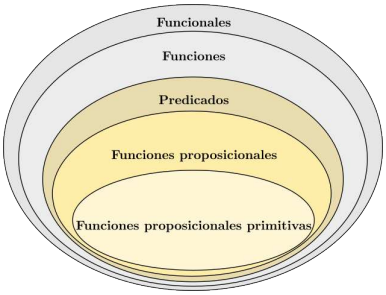


Figura 4.1: Clasificación de funciones por sus parámetros y tipo de valor de salida.

Ejemplo 4.1

- El funcional `aridad` recibe como entrada una función y devuelve como salida la cantidad de variables formales con las que se ha definido la función:

```
def aridad(F: callable) -> int:
    """
    Devuelve la cantidad de variables formales de la función
        .
    """
    return F.__code__.co_nlocals
```

En `Python` aún las funciones son objetos. Un método de toda función es `__code__`, que proporciona `co_nlocals`, el número de variables locales [parámetros] de la función.

- La función `suma` recibe enteros y devuelve un entero.

```
def suma(a: int, b:int) -> int:
    """
    Devuelve la suma de a con b.
    """
    return a + b
```

- El predicado `esMayor` recibe enteros y devuelve un booleano.

```
def esMayor(a: int, b:int) -> bool:
    """
    Devuelve True si a es mayor que b
        .
    """
    return a > b
```

- La función lógica `no_ayb` recibe dos booleanos y devuelve un booleano.

```
def no_ayb(a: bool, b:bool) -> bool:
    """
    Devuelve True si neg(a and b) es True.
    """
    return not(a and b)
```

- La función lógica primitiva «Verdad» recibe booleanos y devuelve un booleano.

```
def Verdad(a: bool, b:bool) -> bool:
    """
    Siempre devuelve True.
    """
    return True
```

No pasa nada si a un predicado le dicen función, o a una función lógica le dicen predicado, porque la clase «función» incluye a los predicados y a las funciones lógicas.

Cuando se escribe un texto como «la función `foo`», la palabra `foo` se refiere al nombre del procedimiento que está definido y nombrado como `foo`, mientras que en expresiones como «si `foo(p,q)` es **V**...» [para ciertos valores `p` y `q` válidos], la expresión `foo(p,q)` se refiere al resultado que presenta la función `foo` cuando se aplican los parámetros `p` y `q`, que en el ejemplo es un valor de tipo `bool`.

Debido a que `foo(p,q)` es de tipo `bool`, entonces se puede utilizar dentro de expresiones lógicas compuestas sin cometer errores de mezcla de tipos. Por su parte `foo` es de tipo `callable`, por lo que hacer la conjunción de `foo` con una proposición, sería un error ya que no se puede hacer una conjunción con un valor de tipo booleano y un valor de tipo función.

Para los propósitos de este libro, las funciones lógicas primitivas son especiales porque:

1. Reciben como entrada los valores booleanos que representan el valor dado a las proposiciones.
2. Modela el comportamiento de un término de enlace, utilizando únicamente funciones definidas por casos y términos de enlace de aridad menor.
3. Como respuesta, devuelven el valor booleano que es la interpretación de la molécula.

4.2 Funciones λ

La notación λ es una manera de representar el procedimiento que debe hacerse con ciertos datos para realizar ciertas acciones, la sintaxis es [SK95]:

$$\lambda v_0, \ldots, v_{a-1} \bullet expr$$

Donde:

$v_0, \ldots, v_{a-1}$**:** Es una colección de a identificadores [nombres para variables].

$expr$**:** Es una expresión que puede involucrar a esas variables.

Una expresión λ se refiere al procedimiento que realiza cierta función, pero no tiene nombre. Una función λ puede estar presente como:

1. **Procedimiento**. Es el algoritmo o secuencia de pasos a realizar, junto con las variables formales, también conocidas como «parámetros»:

«el procedimiento»	Parámetros	Separador	Algoritmo
λ	$v_0, \ldots, v_{a-1}$	$\bullet$	$expr$

   Al crear una definición donde se relaciona un concepto con su significado, solamente aparece el significado sin el concepto. El significado es la expresión $\lambda v_0 \ldots, v_{a-1} \bullet expr$, porque este tipo de funciones son anónimas, cada vez que se requiera, debe escribirse toda la expresión. La aridad de la función anónima es a, la cantidad de variables declaradas.
2. **Definición**. Es cuando la expresión λ es relacionada o ligada a un identificador. Tanto la expresión λ como el identificador asociado con esta expresión son de tipo `callable`, es decir, se pueden invocar. La asignación tiene la forma:

$$\texttt{funId} \leftarrow\!\shortmid \lambda v_0 \ldots, v_{a-1} \bullet expr$$

Una vez hecha la asignación, se crea la definición del concepto `funId`, con significado $\lambda v_0,\ldots,v_{a-1} \bullet expr$. Como cualquier otra definición, una función en formato λ tiene concepto y significado:

Concepto	«significa/vale»	Significado
`funId`	$\leftarrow\!\!\shortmid$	$\lambda v_0,\ldots,v_{a-1} \bullet expr$

También, como cualquier otra definición, en un momento determinado se puede utilizar ya sea el nuevo concepto o solo el significado.

3. **Aplicación**. Una función se aplica cuando se invoca ya sea el concepto o el significado junto con los argumentos. El «argumento» es el valor asociado a una variable formal. Una vez instanciados los parámetros, se realiza el cómputo definido para la función. En la aplicación de una función se puede utilizar la expresión λ o bien el concepto con el que ha relacionado el algoritmo. En ambos casos se produce ($\mapsto$) un valor de salida:
   a) $\texttt{funId}(k_1,\ldots,k_{a-1}) \mapsto val$,
   b) $\{\lambda v_0,\ldots,v_{a-1} \bullet expr\}(k_1,\ldots,k_{a-1}) \mapsto val$.

   Al aplicar una función mediante el nombre, este nombre es sustituido por su significado, que es una expresión λ, luego son instanciados los parámetros con los argumentos proporcionados y finalmente es aplicada la expresión para computar el valor de salida, así:
   c) $\{\lambda v_0 \leftarrow\!\!\shortmid k_0,\ldots,v_{a-1} \leftarrow\!\!\shortmid k_{a-1} \bullet expr\} \mapsto val$.

4.2.1 Método de sustitución para evaluar funciones

Para obtener un resultado de la aplicación de una función, seguiremos el método de sustitución, que se describe en los siguientes pasos.

1. Se invoca la función por medio de su nombre y con sus argumentos.
2. Se obtiene el significado del nombre de la función.
3. Se sustituyen los parámetros de la función por los argumentos de la solicitud.
4. Se construye la expresión a evaluar, cambiando las variables paramétricas por sus valores asignados.
5. Se obtiene el cómputo final.

Ejemplo 4.2

Supongamos la siguiente definición: $\texttt{sum} \leftarrow\!\!\shortmid \lambda x,y \bullet x+y$.

Se ha creado una función llamada `sum`, que ha sido definida con el procedimiento que requiere dos variables formales, x y y, con las que se debe calcular la suma aritmética de x con y.

Paso	Descripción	Comentario
1:	$\texttt{sum}(3,4)$	*Se invoca la función sum, con argumentos 3 y 4.*
2:	$\{\lambda x,y \bullet x+y\}(3,4)$	*Se obtiene la definición del concepto sum.*
3:	$\{\lambda x \leftarrow\!\!\shortmid 3, y \leftarrow\!\!\shortmid 4 \bullet x+y\}$	*Se sustituyen los parámetros por los valores de los argumentos.*
4:	$\{3+4\}$	*Se construye la expresión a evaluar.*
5:	$\mapsto 7$	*Se produce el valor 7.*

Ejemplo 4.3

Escribir la definición de un función λ que tome el valor de un número n y multiplique ese número por sí mismo; luego asociar la función λ al identificador `cuad`. Finalmente aplicar la función con el número 4.

1. La función λ requerida, toma un valor de entrada numérico llamado n, y ese número debe ser multiplicado por sí mismo:

$$\lambda n \bullet n * n$$

2. La segunda parte es asociar esta expresión a una variable llamada `cuad`:

$$\texttt{cuad} \leftarrow\!\shortmid\ \lambda n \bullet n * n$$

La expresión completa se puede leer «En adelante, `cuad` es la función que toma un número n, para multiplicar ese número n por sí mismo». La última parte del ejercicio es la aplicación de la función. La aplicación se puede hacer utilizando el identificador asociado o su expresión λ:

1. `cuad`(4) ▹ *Se aplica la función* `cuad` *con el argumento 4*.
2. $\{\lambda n \bullet n * n\}(4)$ ▹ *Se obtiene el significado del concepto* `cuad`.
3. $\{\lambda n \leftarrow\!\shortmid\ 4 \bullet n * n\}$ ▹ *Se asigna el valor 4 a la variable formal n*.
4. $\{4 * 4\}$ ▹ *Se sustituye el valor 4 en cada ocurrencia del identificador n*.
5. $\mapsto 16$ ▹ *Se produce el valor 16*.

4.3 Funciones por casos

Las funciones definidas por casos, a veces llamadas «funciones por partes» o «funciones a trozos», son una forma de expresar una función ya sea lógica, matemática o computacional que tiene diferentes definiciones o reglas en diferentes conjuntos de valores de entrada. Estas funciones son especialmente útiles cuando la regla que describe la función cambia en dependencia del valor de al menos una variable independiente.

Una función definida por casos se expresa típicamente como una serie de reglas o definiciones que se aplican a conjuntos de valores específicos de la variable independiente. Por lo general, se utiliza una notación con llaves o barras verticales para separar las diferentes definiciones. Por ejemplo:

$$f(x) \mapsto \begin{cases} x^2 & \text{si } x \geq 0 \\ -x^2 & \text{si } x < 0 \end{cases}$$

En este caso, la función llamada f toma como entrada un valor x, y en caso de que x sea mayor o igual a cero, entonces se produce el valor de salida x^2. Por otro lado, si el número x es menor que cero, entonces se produce $-x^2$ como salida.

Cada regla o definición en una función definida por casos está asociada a un conjunto de condiciones bajo las cuales se aplica. Estos intervalos o condiciones se expresan en términos de la variable independiente.

Las funciones definidas por casos también son útiles en lógica computacional, donde se pueden implementar las funciones lógicas primitivas utilizando estructuras condicionales [como `if`, `if-else` o incluso `if-elif-else`] para manejar diferentes situaciones.

Una función λ que se define por casos donde un caso se cumple al verificarse una condición y el otro caso sucede cuando esa condición no es verificada, se puede escribir con la sintaxis de una línea:

$$\lambda v_0,\ldots,v_{a-1} \bullet \langle expr\rangle \texttt{ if } \langle expr-bool\rangle \texttt{ else } \langle expr-alt\rangle.$$

Lo que también se puede escribir en forma de función por casos separados:

$$\lambda v_0,\ldots,v_{a-1} \bullet \begin{cases} \langle expr\rangle & \text{si } \langle expr-bool\rangle \\ \langle expr-alt\rangle & eoc\triangleright \textit{«en otro caso»}. \end{cases}$$

donde:

$expr$:	Es una expresión válida en el lenguaje.
$expr-bool$:	Es una proposición con valor **V**.
$expr-alt$:	Es una expresión válida en el lenguaje, que se ejecuta solamente cuando la proposición es **F**.

En `Python` también es posible escribir funciones por casos, y se utilizan expresiones condicionales; de hecho hay tres tipos de expresiones condicionales que son utilizadas en diferentes contextos para expresar uno, dos o más casos. En `Python` se usan expresiones condicionales de:

Un caso `[if]`: Se utilizan cuando se requiere hacer un bloque de instrucciones cuando se verifica un caso. La sintaxis en `Python` es:

```
lambda v0,...,va-1: expr if expr-bool else None
```

La palabra `None` es un valor especial que representa la ausencia de un valor. Se utiliza para indicar que una variable o una expresión no apunta a ningún objeto o no tiene ningún valor asignado. Es similar a `null` en otros lenguajes de programación.

Ejemplo 4.4

Se desea escribir una función lambda que reciba como único argumento un número x y que devuelva **V** cuando el número x sea mayor que 10. Prueba la función con los valores 16 y luego con 5.

```
>>> lambda x: True if x > 10 else None
<function <lambda> at 0x7f14f599d000>
>>> (lambda x: True if x > 10 else None)(16)
True
>>> (lambda x: True if x > 10 else None)(5)
>>>
```

Dos casos `[if-else]`: Se utilizan cuando hay un bloque de instrucciones que se deben hacer cuando se verifica un caso, y hay otro bloque cuando no se verifica el caso [*eoc* en otro caso]. La palabra clave `else` sirve para verificar cualquier otro caso, se lee «de otro modo», «en otro caso». La sintaxis en `Python` es:

```
lambda v0,...,va-1: expr if expr-bool else expr-alt
```

Ejemplo 4.5

Se desea modelar una función anónima que reciba como parámetros de entrada dos números x y y, la función debe devolver el cociente $\frac{x}{y}$ si $y > 0$, de otro modo devolverá 0.

La función anónima se puede escribir como una función de dos casos:

$$\lambda x, y \bullet \begin{cases} \frac{x}{y} & \text{si } y > 0 \\ 0 & eoc \end{cases}$$

Lo que se traduce en `Python` como:

```
>>> lambda x, y: x / y if y > 0 else 0
<function <lambda> at 0x7f14f599d360>
>>> (lambda x, y: x / y if y > 0 else 0)(10, 5)
2.0
>>> (lambda x, y: x / y if y > 0 else 0)(10, 0)
0
>>>
```

Más de dos casos `[if-elif-else]`: Se utilizan cuando hay un bloque de instrucciones que se deben ejecutar si se verifica un caso, y hay varios casos que se deben verificar. El primer caso por verificar se identifica con la palabra clave `if`; desde el segundo caso hasta el penúltimo, se identifican con la palabra clave `elif`, y el último caso siempre es `else`. La sintaxis es:

`lambda` $v_0,\ldots,v_{a-1}$`: expr if expr-bool else (expr if expr-bool else)`* `expr-alt`

En la notación anterior, la parte encerrada entre paréntesis debe ocurrir cero o más veces. Si no se coloca esa parte, la expresión es idéntica a una expresión de dos casos. Cada parte [de lo que está entre el paréntesis] se debe colocar una vez por cada caso.

Ejemplo 4.6

Para modelar una función anónima que tome como entrada un número x, y devuelva ese mismo número x si $x > 0$; pero si x es cero, entonces debe devolver 0; cualquier otro caso se resuelve devolviendo el número $-x$. Este procedimiento tiene la función de obtener el valor absoluto de un número. Claro que hay maneras más simples de hacerlo, pero se hace de esta manera por razones didácticas.

$$\lambda x \bullet \begin{cases} x & \text{si } x > 0 \\ 0 & \text{si } x = 0 \\ -x & eoc \end{cases}$$

En `Python` se puede traducir como:

```
>>> lambda x: x if x > 0 else 0 if x == 0 else -x
<function <lambda> at 0x7f14f599d5a0>
>>> (lambda x: x if x > 0 else 0 if x == 0 else -x)(10)
10
>>> (lambda x: x if x > 0 else 0 if x == 0 else -x)(0)
0
>>> (lambda x: x if x > 0 else 0 if x == 0 else -x)(-10)
10
>>>
```

Python es un lenguaje que no es puramente funcional, de hecho está fuertemente influenciado por lenguajes imperativos como C, de modo que el uso de las funciones lambda se prefiere para procedimientos cortos, de una sola expresión. Para procedimientos más complejos se prefiere la notación habitual def. En esta notación el ejemplo 4.6 es:

```
def absoluto(x:float)-> float:
    """
    devuelve el valor absoluto del argumento.
    """
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

La ventaja de la notación def sobre la notación λ es que se pueden colocar comentarios y ayudas sobre los tipos de datos, además, se pueden escribir procedimientos más complejos sin perder legibilidad.

Ejercicios

1. Los siguientes enunciados sugieren la creación de una variable. Escribe una variable computacional adecuada para el propósito que indica el enunciado. Asígnale un valor inicial de acuerdo al texto.
   a) *Ejemplo.* Un profesor desea llevar el control de las calificaciones de sus alumnos. La calificación de un alumno es un número entero entre 0 y 10, por lo que requiere una variable para almacenar esa calificación. *Debido a que la información que el profesor desea es la «calificación», podemos crear una variable llamada* `calif`*, e instanciarla con un valor 0, que es un número entero acorde a las restricciones del texto. Aunque* `Python` *no requiere la declaración de tipos, se puede dar información sobre el tipo de dato:*

   ```
   calif : int = 0
   ```

   b) Se desea hacer un programa de computadora que almacene el valor de verdad de la premisa p.
   c) Se está haciendo un estudio demográfico, en particular sobre la edad de las personas económicamente activas de un país.
   d) Una cadena de televisión está interesada en saber cuánto tiempo el televidente permanece interesado en un programa de televisión.
2. En los siguientes ejercicios se describen las entradas y salida de algunas funciones. De acuerdo con la clasificación de funciones de la página 96 decide de qué tipo es cada una.
   a) *Ejemplo.* Una función que verifica que la suma de dos números enteros, que son pasados como parámetros, sea mayor que un valor numérico que es pasado como tercer parámetro.
   La función descrita toma como entrada tres valores numéricos. La palabra «verifica» sugiere que la salida debe ser un valor booleano, por lo que la función debe ser un 'predicado'.
   b) Una función que calcula el índice de masa corporal de una persona, considerando el peso de una persona en kilogramos y la estatura en metros.
   c) Una función que toma como datos de entrada dos proposiciones y devuelve como salida el valor de verdad de la proposición creada por la conjunción de los datos de entrada.
   d) Una función que recibe como entrada dos proposiciones p y q. Como salida devuelve el valor de la expresión molecular $\lambda p, q : (\neg p \not\rightarrow (q \wedge \neg p))$
   e) Una función que determina si una persona debe recibir un descuento o no. La decisión se debe tomar en base a la información de la persona. La información de la persona constituye una tupla de datos:
   $\langle$*Nombre, Edad, Lugar de residencia, Tiempo de residencia*$\rangle$.
3. Determina qué tipo de función es cada una de las siguientes.
   a) *Ejemplo.*

   ```
   def foo1(p: bool, q: bool) -> bool:
       """
       Calcula la conjunción de p con q
       """
       if p:
           return q
       else:
           return False
   ```

La función recibe como datos de entrada dos valores booleanos y devuelve un booleano. De acuerdo con el comentario de documentación, se calcula la conjunción, por lo que es una 'función lógica primitiva'.

b)

```
def foo2(p: int, q: str) -> bool:
    """
    Determina si la letra en la posición p de una cadena es
        vocal
    """
    p = p % len(q)
    p = q[p]
    q = p in ['a', 'e', 'i', 'o', 'u']
    return q
```

c)

```
def foo3(p: float, q: float) -> float:
    """
    Calcula el área de una elipse con radio mayor
        p
    y radio menor q
    """
    pi = 3.14159
    area = pi * p * q
    return area
```

d)

```
def foo4(p: bool, q: bool) -> bool:
    """
    Calcula la Verdad de p con q
    """
    if p:
        return True
    else:
        return True
```

e)

```
def foo5(p: bool, q: bool) -> bool:
    """
    Calcula la expresión (λp,q • ¬p ∨ (q↑p))
    """
    p = not(p) or (not(q and p))
    return p
```

4. Escribe una función λ para los siguientes problemas:
   a) *Ejemplo.* Una función que tome dos números enteros a y b. La función calcula la suma de los cuadrados de ambos números.
   $\lambda a, b \bullet a^2 + b^2$.
   b) Una función que tome un número entero y devuelva el número entero siguiente.
   c) Una función que tome un número x y calcule el valor del cuadrado del número más tres veces ese número más diez.
   d) Una función que reciba como parámetros dos números, y devuelva **V** si el primero es mayor o igual que el segundo.
   e) Una función que reciba un número y devuelva 1 si el número es mayor que 100; que devuelva 0 si el número es exactamente igual a 100 y que devuelva -1 si el número dado es menor que 100.
5. Crea una definición para cada función λ del ejercicio anterior. Luego realiza una evaluación de la función con los argumentos propuestos.

a) *Ejemplo.* Nombre: `sumCuad`; invocación: `sumCuad(4,5)`.
1) `sumCuad` $= \lambda a, b \bullet a^2 + b^2$ ▷ *Se define el concepto.*
① `sumCuad(4,5)` ▷ *Se invoca la función.*
② $\{$`sumCuad` $\leftarrow\!\shortmid \lambda a, b \bullet a^2 + b^2\}(4,5)$
③ $\{\lambda a \leftarrow\!\shortmid 4, b \leftarrow\!\shortmid 5 \bullet a^2 + b^2\}$
④ $\{4^2 + 5^2\} \mapsto \{16 + 25\}$
⑤ $\mapsto 41$

b) Nombre: `sig`; invocación: `sig(15)`
c) Nombre: `poli`; invocación: `poli(7)`
d) Nombre: `esMayor`; invocación: `esMayor(15,81)`
e) Nombre: `en100`; invocación: `en100(24)`

6. Traduce las siguientes funciones por casos en expresiones λ de una sola línea que sean válidas en `Python` [ver la página 101].

a) *Ejemplo.* $\lambda x \bullet \begin{cases} 1 & \text{si } x \geq 0 \\ -1 & eoc \end{cases}$
Se puede traducir como `lambda x: 1 si x >= 0 else -1`

b) $\lambda p \bullet \begin{cases} \mathbf{V} & \text{si } p = \mathbf{F} \\ \mathbf{F} & eoc \end{cases}$

c) $\lambda x \bullet \begin{cases} \texttt{'A'} & \text{si } 9.0 < x \\ \texttt{'B'} & \text{si } 7.0 < x \leq 9.0 \\ \texttt{'C'} & \text{si } 6.0 < x \leq 7.0 \\ \texttt{'R'} & eoc \end{cases}$

d) $\lambda x \bullet \begin{cases} 1 & \text{si } x \geq 0 \\ -1 & eoc \end{cases}$

e) $\lambda p, q, r \bullet \begin{cases} \mathbf{V} & \text{si } (p \vee r) \rightarrow (r \wedge \neg q) \\ \mathbf{F} & eoc \end{cases}$

7. En los siguientes ejercicios con funciones λ y sus invocaciones, encuentra y describe el error.

a) *Ejemplo.* `areaTr` $\leftarrow\!\shortmid \lambda b, h \bullet \frac{b \times h}{2}$ ▷ *Calcula el área de un triángulo.*
Con la invocación `AreaTr`$(23, 12)$.
Error: El nombre definido es `areaTr` pero el nombre invocado es `AreaTr`. En general se espera que el nombre invocado sea exactamente el mismo que el nombre en la definición. En matemáticas se suele ser más flexible con esto, por ejemplo sin, sen, $seno$, Sen, Sin son nombres que pueden referirse a la misma función «seno», pero en programación, en general se requiere que los nombres coincidan por completo.

b) `sumaDobles` $\leftarrow\!\shortmid \lambda a, b, c \bullet 2a + 2b + 2c + 2d$.
Con la invocación `sumaDobles`$(3, 4, 6)$.

c) `sobreUmbral` $\leftarrow\!\shortmid \lambda m, \kappa \bullet$ `"`$Acierto$`" if` $\mu \geq \kappa$ `else "`$Fallo$`"`
Con la invocación `sobreUmbral`$(14.78, 15.0)$

Funciones lógicas primitivas 5

En la clasificación de funciones [página 96], las funciones lógicas primitivas o también conocidas como funciones proposicionales primitivas, son las funciones que operan con proposiciones, es decir, con valores de verdad y al ser evaluadas, se produce una nueva proposición, esto es, un nuevo valor de verdad.

5.1 Tablas de Verdad

Una tabla de verdad es una representación en columnas y filas que muestra los posibles valores que tiene una expresión proposicional, en dependencia de los valores asociados a las variables que la componen.

En esa tabla, las columnas son identificadas con una expresión proposicional, se dedican las primeras columnas para las expresiones atómicas y los renglones indican todas las combinaciones posibles de valores de verdad para las variables proposicionales que componen la expresión.

Las columnas de la tabla de verdad están divididas en dos partes, como se puede observar en la tabla 5.1:

1. Las expresiones que tienen valores conocidos. Son todas las columnas, excepto la extrema derecha. Los valores se colocan bajo la columna en cada fila.
2. La nueva expresión cuyos valores se deben calcular. Al calcular cada nuevo valor, se consideran aquellos valores de expresiones conocidas.

Los casos en la tabla de verdad [los renglones] son obtenidos por cada combinación diferente de valores que las variables proposicionales pueden tener. Así, en una expresión unaria hay solamente dos casos, cuando la variable es **V** y cuando es **F**. Para el caso unario [$aridad = 1$], solo hay 2 casos porque $2^{aridad} = 2^1 = 2$. En el caso binario

	Expresiones con valor conocido		*Nueva expresión*	
	p	q	⟨—⟩	
caso 1:	**V**	**V**	□	*Valores por calcular*
caso 2:	**V**	**F**	□	
caso 3:	**F**	**V**	□	
caso 4:	**F**	**F**	□	

Tabla 5.1: *Tabla de verdad para dos proposiciones conocidas y una proposición desconocida.*

$[aridad = 2]$ hay $2^{aridad} = 2^2 = 4$ diferentes casos, estos se pueden ver en la tabla 5.1. Para el caso de 3 variables, $aridad = 3$, por lo que hay $2^{aridad} = 2^3 = 8$ casos.

Una vez calculados los valores de cada caso en la columna de valores por calcular, la columna forma parte de valores conocidos y se puede agregar una nueva columna más a la derecha, para calcular los valores de verdad de una nueva expresión.

Ejemplo 5.1

Para construir la tabla de verdad de $\lambda p, q \bullet \neg(p \wedge q)$ se observa que utiliza dos variables booleanas: p y q, lo que indica que se requiere una tabla con cuatro renglones.

Primero se colocan las columnas con valores conocidos, estas son las columnas para las proposiciones p y q, así se tienen ya dos columnas de valores conocidos.

p	q	⟨—⟩
V	**V**	□
V	**F**	□
F	**V**	□
F	**F**	□

La expresión es evaluada por partes, tomando cada subexpresión entre paréntesis, comenzando desde el más interno. Si dos expresiones se pueden hacer en el mismo tiempo porque tienen la misma prioridad, se puede hacer cualquiera de ellas primero.

p	q	$(p \wedge q)$
V	**V**	**V**
V	**F**	**F**
F	**V**	**F**
F	**F**	**F**

Ahora $p \wedge q$ es parte de las proposiciones conocidas y se procede con $\neg(p \wedge q)$:

p	q	$(p \wedge q)$	$\neg(p \wedge q)$
V	**V**	**V**	□
V	**F**	**F**	□
F	**V**	**F**	□
F	**F**	**F**	□

$\longrightarrow$

p	q	$(p \wedge q)$	$\neg(p \wedge q)$
V	**V**	**V**	**F**
V	**F**	**F**	**V**
F	**V**	**F**	**V**
F	**F**	**F**	**V**

5.1.1 Tablas de verdad de los principales enlaces

Los enlaces también son conocidos como «funciones lógicas primitivas» y pueden ser catalogados de acuerdo a su aridad. De esta manera los enlaces que requieren una variable son «enlaces unarios», aquellos que requieren dos variables son «enlaces binarios» y lo que requieren más de dos, se pueden llamar «enlaces n-arios».

En número de posibles enlaces depende del número de casos posibles. Para el caso unario, hay $2^{casos} = 2^2 = 4$ diferentes funciones lógicas primitivas [tabla 5.2]. En el caso binario, hay $2^{casos} = 2^4 = 16$ diferentes funciones lógicas primitivas [tabla 5.3].

Los enlaces binarios, aún se pueden clasificar en dos grupos: Los «enlaces binarios propios», que son aquellos que *realmente* ocupan las dos variables que requieren; y los «enlaces binarios impropios» son los que no ocupan las dos variables requeridas.

- Enlaces
  - Unarios
    - $\bot$ ▹ *Falsedad.*
    - $\mathtt{I}$ ▹ *Identidad.*
    - $\neg$ ▹ *Negación.*
    - $\top$ ▹ *Verdad.*
  - Binarios
    - Impropios
      - $\bot$ ▹ *Falsedad [0].*
      - $\mathtt{P}$ ▹ *Primera componente [3].*
      - $\mathtt{Q}$ ▹ *Segunda componente [5].*
      - $\overline{\mathtt{Q}}$ ▹ *Negación de segunda componente [10].*
      - $\overline{\mathtt{P}}$ ▹ *Negación de primera componente [12].*
      - $\top$ ▹ *Verdad [15].*
    - Propios
      - $\wedge$ ▹ *Conjunción [1].*
      - $\not\rightarrow$ ▹ *q-Inhibición [2].*
      - $\not\leftarrow$ ▹ *p-Inhibición [4].*
      - $\oplus$ ▹ *Disyunción exclusiva [6].*
      - $\vee$ ▹ *Disyunción [7].*
      - $\downarrow$ ▹ *Flecha de Peirce [8].*
      - $\leftrightarrow$ ▹ *Si y sólo si [9].*
      - $\leftarrow$ ▹ *Conversa [11].*
      - $\rightarrow$ ▹ *Implicación [13].*
      - $\uparrow$ ▹ *Barra de Sheffer [14].*

☞ El número entre corchetes al lado derecho del nombre del enlace se refiere a un índice puesto de manera arbitraria, que sirve de referencia [tabla 5.3].

En la tabla 5.2 se muestran los 4 enlaces unarios, junto con el símbolo que generalmente se utiliza.

p	$\bot$	$\mathtt{I}$	$\neg$	$\top$
V	**F**	**V**	**F**	**V**
F	**F**	**F**	**V**	**V**

Tabla 5.2: *Tablas de verdad de los 4 enlaces lógicos unarios.*

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
p	q	$\bot$	$\wedge$	$\not\rightarrow$	P	$\not\leftarrow$	Q	$\oplus$	$\vee$	$\downarrow$	$\leftrightarrow$	$\overline{\text{Q}}$	$\leftarrow$	$\overline{\text{P}}$	$\rightarrow$	$\uparrow$	$\top$
V	V	F	V	F	V	F	V	F	V	F	V	F	V	F	V	F	V
V	F	F	F	V	V	F	F	V	V	F	F	V	V	F	F	V	V
F	V	F	F	F	F	V	V	V	V	F	F	F	F	V	V	V	V
F	F	F	F	F	F	F	F	F	F	V	V	V	V	V	V	V	V

Tabla 5.3: *Tablas de verdad de los 16 enlaces lógicos binarios.*

En la tabla 5.3 se muestran los enlaces binarios, mientras que en la tabla 5.4, se enuncia el nombre común de cada enlace junto con su símbolo usado en este libro.

	Nombre	Símbolo
0:	Falsedad	$\bot$
1:	Conjunción	$\wedge$
2:	q-Inhibición	$\not\rightarrow$
3:	Primera componente	P
4:	p-Inhibición	$\not\leftarrow$
5:	Segunda componente	Q
6:	Disyunción exclusiva	$\oplus$
7:	Disyunción	$\vee$
8:	Flecha de Peirce	$\downarrow$
9:	Si y sólo si	$\leftrightarrow$
10:	Negación Segunda Comp.	$\overline{\text{Q}}$
11:	Conversa	$\leftarrow$
12:	Negación Primera Comp.	$\overline{\text{P}}$
13:	Implicación	$\rightarrow$
14:	Barra Sheffer	$\uparrow$
15:	Verdad	$\top$

Tabla 5.4: *Nombre y símbolo de los 16 enlaces lógicos binarios.*

5.1.2 Equivalencia lógica

Una de las aplicaciones de las tablas de verdad es demostrar la equivalencia entre dos *fbf*. La equivalencia entre expresiones lógicas queda demostrada cuando se garantiza que ambas *fbf* tienen el mismo valor de verdad cuando las variables booleanas involucradas en cada expresión se unifican con mismo valor de verdad cada una.

Desde el punto de vista de una tabla de verdad, dos expresiones lógicas fbf_1 y fbf_2 son lógicamente equivalentes cuando las columnas de cada *fbf* tienen los mismos valores en los mismos renglones.

Ejemplo 5.2

Mediante una tabla de verdad se demuestra la equivalencia entre las expresiones $\lambda p, q \bullet p \oplus q$ y $\lambda p, q \bullet (p \wedge \neg q) \vee (q \wedge \neg p)$: En primer lugar tomamos los valores de verdad de la disyunción exclusiva que es el enlace [6] de la tabla 5.3:

Los valores en estas columnas deben ser iguales

p	q	$\cdots$	$fbf1$	$\cdots$	$fbf2$
V	**V**	$\cdots$	□	$\cdots$	□
V	**F**	$\cdots$	□	$\cdots$	□
F	**V**	$\cdots$	□	$\cdots$	□
F	**F**	$\cdots$	□	$\cdots$	□

Tabla 5.5: *Equivalencia lógica entre dos fbfs binarias.*

p	q	$p \oplus q$
V	**V**	**F**
V	**F**	**V**
F	**V**	**V**
F	**F**	**F**

Tomando $p \oplus q$ como una expresión con valores conocidos, construimos la tabla de verdad de la segunda *fbf*. La siguiente expresión es $\lambda p, q \bullet (p \wedge \neg q) \vee (q \wedge \neg p)$. Observamos que es una disyunción de dos conjunciones que requieren ambas de la negación de una proposición, por lo que haremos las negaciones primero.

p	q	$p \oplus q$	$\neg p$	$\neg q$
V	**V**	**F**	□	□
V	**F**	**V**	□	□
F	**V**	**V**	□	□
F	**F**	**F**	□	□

$\longrightarrow$

p	q	$p \oplus q$	$\neg p$	$\neg q$
V	**V**	**F**	**F**	**F**
V	**F**	**V**	**F**	**V**
F	**V**	**V**	**V**	**F**
F	**F**	**F**	**V**	**V**

Una vez conocidos los valores de $\neg q$ y $\neg p$, podemos utilizar esos valores para construir las expresiones $(p \wedge \neg q)$ y $(q \wedge \neg p)$. Ambas expresiones dependen de elementos que ya tienen valores conocidos, por lo que el orden en que se deben hacer no es trascendente. Se harán entonces en orden de izquierda a derecha.

p	q	$p \oplus q$	$\neg p$	$\neg q$	$(p \wedge \neg q)$	$(q \wedge \neg p)$
V	**V**	**F**	**F**	**F**	□	□
V	**F**	**V**	**F**	**V**	□	□
F	**V**	**V**	**V**	**F**	□	□
F	**F**	**F**	**V**	**V**	□	□

$\longrightarrow$

p	q	$p \oplus q$	$\neg p$	$\neg q$	$(p \wedge \neg q)$	$(q \wedge \neg p)$
V	**V**	**F**	**F**	**F**	**F**	**F**
V	**F**	**V**	**F**	**V**	**V**	**F**
F	**V**	**V**	**V**	**F**	**F**	**V**
F	**F**	**F**	**V**	**V**	**F**	**F**

Por último, obtenemos la tabla de verdad final agregando la columna para la expresión $(p \wedge \neg q) \vee (\neg p \wedge q)$, considerando todas las columnas anteriores como datos ya conocidos.

p	q	$p \oplus q$	$\neg p$	$\neg q$	$(p \wedge \neg q)$	$(q \wedge \neg p)$	$(p \wedge \neg q) \vee (q \wedge \neg p)$
V	**V**	**F**	**F**	**F**	**F**	**F**	**F**
V	**F**	**V**	**F**	**V**	**V**	**F**	**V**
F	**V**	**V**	**V**	**F**	**F**	**V**	**V**
F	**F**	**F**	**V**	**V**	**F**	**F**	**F**

Observamos que las columnas marcadas tienen los mismos valores considerando la lectura de los renglones en el mismo orden. Esto garantiza que $\lambda p,q \cdot p \oplus q \equiv \lambda p,q \cdot (p \wedge \neg q) \vee (\neg p \wedge q)$.

5.1.3 Tautologías

La palabra «tautología» significa «repetición de una misma idea o pensamiento». En Lógica, una tautología es un tipo de *fbf* cuyo valor es **V** sin importar el valor de verdad asociado a sus variables componentes, esto es que se llega al mismo valor aún cambiando el valor de verdad de sus variables. La tabla de verdad de una tautología tiene **V** en cada renglón.

Por ejemplo, la función $\lambda p,q \cdot p \top q$ es una tautología, puesto que acorde al criterio del enlace Verdad $\top$ [índice 15 en la tabla 5.3, página 110], la interpretación siempre es **V** sin importar el valor de cada variable.

Ejemplo 5.3

Para determinar si $\lambda p \cdot p \vee \neg p$ es una tautología, se hace la tabla de verdad:

p	q	$\neg p$	$p \vee \neg p$ ▽
V	**V**	**F**	**V**
V	**F**	**F**	**V**
F	**V**	**V**	**V**
F	**F**	**V**	**V**

La columna marcada con ▽ tiene todas sus entradas con **V**, lo que indica que es una tautología.

Ejemplo 5.4

Verificar que $\lambda p,q \cdot (((p \rightarrow q) \wedge p) \rightarrow q)$ es una tautología.

p	q	$(p \rightarrow q)$	$((p \rightarrow q) \wedge p)$	$(((p \rightarrow q) \wedge p) \rightarrow q)$ ▽
V	**V**	**V**	**V**	**V**
V	**F**	**F**	**F**	**V**
F	**V**	**V**	**F**	**V**
F	**F**	**V**	**F**	**V**

Como la columna $((p \rightarrow q) \wedge p) \rightarrow q$ contiene únicamente valores **V**, decimos que es una tautología.

Ejemplo 5.5

Demostrar mediante una tabla de verdad que $\lambda p,q,r \cdot ((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$ es una tautología.

Para demostrarlo, se requiere una tabla de verdad que incluya las tres variables, por lo cual tendremos una tabla con 3 columnas iniciales, una para cada variable.

p	q	r	$p \to q$	$q \to r$	$(p \to q) \wedge (q \to r)$	$(p \to r)$	∇ $((p \to q) \wedge (q \to r)) \to (q \to r)$
V	**V**	**V**	**V**	**V**	**V**	**V**	**V**
V	**V**	**F**	**V**	**F**	**F**	**F**	**V**
V	**F**	**V**	**F**	**V**	**F**	**V**	**V**
V	**F**	**F**	**F**	**F**	**F**	**F**	**V**
F	**V**	**V**	**V**	**V**	**V**	**V**	**V**
F	**V**	**F**	**V**	**V**	**V**	**V**	**V**
F	**F**	**V**	**V**	**V**	**V**	**V**	**V**
F	**F**	**F**	**V**	**V**	**V**	**V**	**V**

5.2 Negación

Definición 5.2.1 -- Negación. Es una función que requiere una proposición p, con la que produce una nueva proposición con valor diferente de p y se define por casos como: $\neg \leftarrowtail \lambda p \cdot \mathbf{F}$ `if` p `else` $\mathbf{V}$.

Código 5.1: La negación

```
"""
La negación de una proposición
"""
neg = lambda p: False if p else True
```

Para verificar la definición, se comparan los resultados de la función `neg`, con la interpretación de la negación [tabla de verdad 5.2 página 109]:

p	$\neg p$
V	**F**
F	**V**

```
>>> neg(True)
False
>>> neg(False)
True
>>>
```

Cada aplicación de la negación a una proposición con valor diferente se muestra en la tabla de verdad, pero se puede escribir cada caso como:

1. $\{\lambda p \cdot \mathbf{F}$ `if` p `else` $\mathbf{V}\}(\mathbf{V}) \mapsto \mathbf{F}$. Al sustituir la función λ por el símbolo $\neg$ que ha sido asociado a esta definición, se tiene: $\neg(\mathbf{V}) \mapsto \mathbf{F}$.

   ☞ Por comodidad, en el caso de la negación se suele omitir los paréntesis si la función afecta a una sola variable, así $\neg(p)$ usualmente se escribe $\neg p$, como ya se ha hecho antes [ver página 42].

2. $\{\lambda p \cdot \mathbf{F}$ `if` p `else` $\mathbf{V}\}(\mathbf{F}) \mapsto \mathbf{V}$, con lectura «la negación de **F** es **V** ». Utilizando el nombre $\neg$ en lugar de la definición λ se tiene $\neg\mathbf{F} \mapsto \mathbf{V}$.

Ejemplo 5.6

Considera la proposición verdadera ⟨*La película ha empezado*⟩. Calcular la negación de la proposición y la doble negación de la misma.

1. ⟨*La película ha empezado*⟩ es **V**.
2. $p \leftarrowtail$ ⟨*La película ha empezado*⟩, por lo que $p \leftarrowtail \mathbf{V}$.
3. $\{\lambda p \cdot \mathbf{F}$ `if` p `else` $\mathbf{V}\}(\mathbf{V}) \mapsto \mathbf{F}$
4. $\{\lambda p \cdot \mathbf{F}$ `if` p `else` $\mathbf{V}\}(\neg\mathbf{V}) \equiv \{\lambda p \cdot \mathbf{F}$ `if` p `else` $\mathbf{V}\}(\mathbf{F}) \mapsto \mathbf{V}$

Al utilizar la función `neg`:

```
>>> la_pelicula_ha_empezado = True
>>> p = la_pelicula_ha_empezado
>>> neg(p)
False
>>> neg(neg(p))
True
>>>
```

5.3 La disyunción [7] y conjunción [1]

Definición 5.3.1 -- Disyunción. Sean p y q dos proposiciones. La función disyunción se define $\vee \leftarrow \lambda p, q \bullet \mathbf{V}$ `if` p `else` q.

Código 5.2: La disyunción

```
"""
La disyunción lógica entre p y q.
"""
o = lambda True if p else q
```

La definición está basada en la tabla de verdad de la disyunción. Observa que si la primera entrada p es **V**, el valor de la disyunción es **V** sin importar el valor de q. Por otro lado, si p no es **V**, entonces el valor de la disyunción es exactamente el mismo que el valor de q. Esto se aprovecha para tomar directamente el valor de q, cuyo valor ya ha sido pasado como parámetro y no hay necesidad de hacer cálculo alguno.

La aplicación de la función para el caso binario [funciones de aridad 2] se enuncia también para cada caso en la tabla de verdad:

1. $\{\lambda p, q \bullet \mathbf{V}$ `if` p `else` $q\}(\mathbf{V}, \mathbf{V}) \equiv \{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{V} : \mathbf{V}$ `if` p `else` $q\}$
   $\equiv \{\mathbf{V}$ `if` $\mathbf{V}$ `else` $\mathbf{V}\}$
   $\mapsto \mathbf{V}$
2. $\{\lambda p, q \bullet \mathbf{V}$ `if` p `else` $q\}(\mathbf{V}, \mathbf{F}) \equiv \{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{F} : \mathbf{V}$ `if` p `else` $q\}$
   $\equiv \{\mathbf{V}$ `if` $\mathbf{V}$ `else` $\mathbf{F}\}$
   $\mapsto \mathbf{V}$
3. $\{\lambda p, q \bullet \mathbf{V}$ `if` p `else` $q\}(\mathbf{F}, \mathbf{V}) \equiv \{\lambda p \leftarrow \mathbf{F}, q \leftarrow \mathbf{V} : \mathbf{V}$ `if` p `else` $q\}$
   $\equiv \{\mathbf{V}$ `if` $\mathbf{F}$ `else` $\mathbf{V}\}$
   $\mapsto \mathbf{V}$
4. $\{\lambda p, q \bullet \mathbf{V}$ `if` p `else` $q\}(\mathbf{F}, \mathbf{F}) \equiv \{\lambda p \leftarrow \mathbf{F}, q \leftarrow \mathbf{F} : \mathbf{V}$ `if` p `else` $q\}$
   $\equiv \{\mathbf{V}$ `if` $\mathbf{F}$ `else` $\mathbf{F}\}$
   $\mapsto \mathbf{F}$

El comportamiento de la función computacional de la disyunción se puede comprobar con una interacción en la consola de `Python`.

p	q	$p \vee q$
V	**V**	**V**
V	**F**	**V**
F	**V**	**V**
F	**F**	**F**

```
>>> o(True, True)
True
>>> o(True, False)
True
>>> o(False, True)
True
>>> o(False, False)
False
>>>
```

☞ En el caso de las funciones binarias [que son definidas con el requerimiento de dos parámetros], la expresión prefija $\vee(p,q)$, en la notación comúnmente utilizada en lógica, se suele escribir en formato entrefijo, es decir, con el operador entre los operandos como en $p \vee q$ que ya se ha utilizado antes [ver página 42]. Ambas notaciones son equivalentes, pero la notación entrefija $\vee(p,q)$ es la que se utiliza mayormente en los lenguajes de programación cuando se define una nueva función.

Ejemplo 5.7

Para modelar la frase «quiero ir a correr o ir a nadar», podemos simbolizar cada proposición atómica como:

$p \leftarrow\!\shortmid$ ⟨*Quiero ir a correr*⟩.
$q \leftarrow\!\shortmid$ ⟨*Quiero ir a nadar*⟩.

Como es una expresión disyuntiva, se modela $p \vee q$, como ambas proposiciones al ser consideradas como premisas son **V**, el valor de la proposición molecular es **V** porque:

$$\{\lambda p, q \bullet \mathbf{V} \texttt{ if } p \texttt{ else } q\}(\mathbf{V},\mathbf{V}) \equiv \vee(\mathbf{V},\mathbf{V}) \mapsto \mathbf{V}.$$

La expresión «No quiero ir a correr o quiero ir a nadar». En este caso, si $p \leftarrow\!\shortmid$ ⟨*quiero ir a correr*⟩ es una proposición **V**, la proposición del texto ⟨*no quiero ir a correr*⟩ se puede modelar como $\neg p$ que tiene valor **F**. Por su parte $q \leftarrow\!\shortmid$ ⟨*quiero ir a nadar*⟩ con valor **V**, por lo que podemos calcular el valor de $\neg p \vee q$ como:

$$\{\lambda p, q \bullet \mathbf{V} \texttt{ if } p \texttt{ else } q\}(\neg(\mathbf{V}),\mathbf{V}) \equiv \vee(\mathbf{F},\mathbf{V}) \mapsto \mathbf{V}.$$

Computacionalmente, utilizando las funciones ya definidas, podemos hacer el mismo ejercicio:

```
>>> p = quiero_ir_a_correr = True
>>> q = quiero_ir_a_nadar = True
>>> o(p, q)
True
>>> o(neg(p), q)
True
>>>
```

En `Python` es posible hacer asignaciones múltiples, como la que se hizo en el ejemplo 5.7: `p = quiero_ir_a_correr = True`, lo que equivale a hacer dos asignaciones usando el mismo valor de verdad:

```
quiero_ir_a_correr = True
p = quiero_ir_a_correr
```

Otro tipo de asignación en `Python` es asignar múltiples valores a igual número de variables, por ejemplo: `a, b = 4, 8`, que tiene el efecto de asignar el 4 a la variable `a`, y el 8 a la variable `b`.

Definición 5.3.2 -- Conjunción. Si p y q son proposiciones. La conjunción de p y q es una nueva proposición **V** cuando ambas proposiciones p y q son **V**; en cualquier otro caso tiene valor **F**. Se define como $\wedge \leftarrow\!\shortmid \lambda p, q \bullet q \texttt{ if } p \texttt{ else } \mathbf{F}$.

Código 5.3: La conjunción

```
1 """
2 La conjunción lógica entre p y q.
3 """
4 y = lambda p, q: q if p else False
```

La definición está basada en el comportamiento observado en la tabla de verdad de la conjunción. La tabla de verdad puede ser dividida en dos partes dependiendo del valor de p. Si p es **V**, lo que ocurre en los primeros dos renglones de la tabla, el valor de la conjunción $p \wedge q$ es el mismo que el valor de q, si q es **V**, también lo es la conjunción, si q es **F**, también la conjunción. Por otro lado, si p es **F**, no hay necesidad de hacer más, el resultado es **F**.

p	q	$p \wedge q$
V	**V**	**V**
V	**F**	**F**
F	**V**	**F**
F	**F**	**F**

```
>>> y(True, True)
True
>>> y(True, False)
False
>>> y(False, True)
False
>>> y(False, False)
False
>>>
```

El comportamiento de la función computacional de la conjunción se puede comprobar al aplicar la función en cada uno de los cuatro posibles casos:

1. $\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \mathbf{F}\}(\mathbf{V},\mathbf{V}) \equiv \{\lambda p \leftarrow\!\shortmid \mathbf{V}, q \leftarrow\!\shortmid \mathbf{V} \bullet q \texttt{ if } p \texttt{ else } \mathbf{F}\} \equiv \{\mathbf{V} \texttt{ if } \mathbf{V} \texttt{ else } \mathbf{F}\} \mapsto \mathbf{V}$
2. $\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \mathbf{F}\}(\mathbf{V},\mathbf{F}) \equiv \{\lambda p \leftarrow\!\shortmid \mathbf{V}, q \leftarrow\!\shortmid \mathbf{F} \bullet q \texttt{ if } p \texttt{ else } \mathbf{F}\} \equiv \{\mathbf{F} \texttt{ if } \mathbf{V} \texttt{ else } \mathbf{F}\} \mapsto \mathbf{F}$
3. $\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \mathbf{F}\}(\mathbf{F},\mathbf{V}) \equiv \{\lambda p \leftarrow\!\shortmid \mathbf{F}, q \leftarrow\!\shortmid \mathbf{V} \bullet q \texttt{ if } p \texttt{ else } \mathbf{F}\} \equiv \{\mathbf{V} \texttt{ if } \mathbf{F} \texttt{ else } \mathbf{F}\} \mapsto \mathbf{F}$
4. $\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \mathbf{F}\}(\mathbf{F},\mathbf{F}) \equiv \{\lambda p \leftarrow\!\shortmid \mathbf{F}, q \leftarrow\!\shortmid \mathbf{F} \bullet q \texttt{ if } p \texttt{ else } \mathbf{F}\} \equiv \{\mathbf{F} \texttt{ if } \mathbf{F} \texttt{ else } \mathbf{F}\} \mapsto \mathbf{F}$

Ejemplo 5.8

Para modelar «Me gusta el helado y ver el atardecer», se puede simbolizar cada proposición haciendo $p \leftarrow\!\shortmid\langle$*Me gusta el helado*$\rangle$ y $q \leftarrow\!\shortmid\langle$*Me gusta ver el atardecer*$\rangle$, ambas de valor **V**. Como es una expresión conjuntiva, se modela como $p \wedge q$, que es **V**.

Ahora la expresión «me gusta el helado y no me gusta ver el atardecer». En este nuevo ejemplo la proposición es ⟨*me gusta el helado y no es verdad que me gusta ver el atardecer*⟩, tenemos las mismas proposiciones atómicas, pero ahora la proposición q debe ser modificada con la negación para que modele correctamente el enunciado $p \wedge \neg q$. Al modelar computacionalmente estas proposiciones:

```
>>> me_gusta_el_helado = True
>>> p = me_gusta_el_helado
>>> me_gusta_ver__el_atardecer = True
>>> q = me_gusta_ver_el_atardecer
>>>  # Me gusta el helado y ver el atardecer :
>>> y(p, q)
True
>>>  # me gusta el helado y no es verdad que me gusta ver el atardecer :
>>> y(p, neg(q))
False
>>>
```

¿Cómo se modela con símbolos la expresión «Ni me gusta el helado, ni ver el atardecer»?

5.4 Implicación [13]

Definición 5.4.1 -- Implicación. Sean p y q dos proposiciones. La implicación genera una nueva proposición de valor **F** únicamente cuando p es **V** y q es **F**; en cualquier otro caso es **V**. Se define $\rightarrow \leftarrow \lambda p, q \bullet q$ `if` p `else` **V**.

Código 5.4: La implicación

```
1 """
2 La implicación lógica
3 """
4 impl = lambda p, q: q if p else True
```

La expresión $\lambda p, q \bullet q$ `if` p `else` **V** de la definición se obtiene de la tabla de verdad. Cuando p es **V**, el valor de $p \rightarrow q$ es el mismo que el valor de q. Por otro lado, de acuerdo con los renglones 3 y 4, si p no es **V**, el resultado siempre es **V** sin importar el valor de q.

La definición 5.4.1 no es la única, otra posible definición es $\rightarrow \leftarrow \lambda p, q \bullet \neg p \vee q$ que se basa en las funciones previamente definidas. La equivalencia entre estas definiciones es más clara al usar la tabla de verdad.

		∇		∇
p	q	$p \rightarrow q$	$\neg p$	$\neg p \vee q$
V	**V**	**V**	**F**	**V**
V	**F**	**F**	**F**	**F**
F	**V**	**V**	**V**	**V**
F	**F**	**V**	**V**	**V**

La definición basada en $\neg p \vee q$ tiene la ventaja de ser más comprensible para quien utiliza la función, pero tiene la desventaja de ser computacionalmente más lenta que la usada en la definición 5.4.1, porque cada aplicación de una función previamente definida tiene que ser sustituida por su definición, este es el caso de '$\neg$' y posteriormente '$\vee$'.

Paso	$\lambda p, q \bullet q$ `if` p `else` **V**	$\lambda p, q \bullet \neg p \vee q$
1:	$\{\lambda p, q \bullet q$ `if` p `else` $\mathbf{V}\}(\mathbf{V}, \mathbf{F})$	$\{\lambda p, q \bullet \neg p \vee q\}(\mathbf{V}, \mathbf{F})$
2:	$\{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{F} \bullet q$ `if` p `else` $\mathbf{V}\}$	$\{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{F} \bullet \neg p \vee q\}$
3:	$\{\mathbf{F}$ `if` $\mathbf{V}$ `else` $\mathbf{F}\}$	$\{\neg \mathbf{V} \vee \mathbf{F}\}$
4:	$\mapsto \mathbf{F}$	$\{\{\lambda p \bullet \mathbf{F}$ `if` p `else` $\mathbf{V}\}(\mathbf{V}) \vee \mathbf{F}\}$
5:		$\{\{\lambda p \leftarrow \mathbf{V} \bullet \mathbf{F}$ `if` p `else` $\mathbf{V}\} \vee \mathbf{F}\}$
6:		$\{\{\mathbf{F}$ `if` $\mathbf{V}$ `else` $\mathbf{V}\} \vee \mathbf{F}\}$
7:		$\{\mathbf{F} \vee \mathbf{F}\}$
8:		$\{\lambda p, q \bullet \mathbf{V}$ `if` p `else` $q\}(\mathbf{F}, \mathbf{F})$
9:		$\{\lambda p \leftarrow \mathbf{F}, q \leftarrow \mathbf{F} \bullet \mathbf{V}$ `if` p `else` $q\}$
10:		$\{\mathbf{V}$ `if` $\mathbf{F}$ `else` $\mathbf{F}\}$
11:		$\mapsto \mathbf{F}$.

La función implicación es la abstracción de la interpretación de una proposición implicativa para cualquiera de los cuatro casos [ver la sección 2.2.4 en la página 39]. Para verificar el comportamiento de la definición computacional de la implicación,

observamos las entradas en la tabla de verdad de la implicación y verificamos con los resultados del cómputo de la función.

1. $\{\lambda p,q \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}(\mathbf{V},\mathbf{V}) \equiv \{\lambda p \leftarrow\!\shortmid \mathbf{V}, q \leftarrow\!\shortmid \mathbf{V} \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}$
   $\equiv \{\mathbf{V} \texttt{ if } \mathbf{V} \texttt{ else } \mathbf{V}\}$
   $\mapsto \mathbf{V}$
2. $\{\lambda p,q \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}(\mathbf{V},\mathbf{F}) \equiv \{\lambda p \leftarrow\!\shortmid \mathbf{V}, q \leftarrow\!\shortmid \mathbf{F} \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}$
   $\equiv \{\mathbf{F} \texttt{ if } \mathbf{V} \texttt{ else } \mathbf{V}\}$
   $\mapsto \mathbf{F}$
3. $\{\lambda p,q \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}(\mathbf{F},\mathbf{V}) \equiv \{\lambda p \leftarrow\!\shortmid \mathbf{F}, q \leftarrow\!\shortmid \mathbf{V} \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}$
   $\equiv \{\mathbf{V} \texttt{ if } \mathbf{F} \texttt{ else } \mathbf{V}\}$
   $\mapsto \mathbf{V}$
4. $\{\lambda p,q \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}(\mathbf{F},\mathbf{F}) \equiv \{\lambda p \leftarrow\!\shortmid \mathbf{F}, q \leftarrow\!\shortmid \mathbf{F} \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}$
   $\equiv \{\mathbf{F} \texttt{ if } \mathbf{F} \texttt{ else } \mathbf{V}\}$
   $\mapsto \mathbf{V}$

```
>>> impl(True, True)
True
>>> impl(True, False)
False
>>> impl(False, True)
True
>>> impl(False, False)
True
>>>
```

Una expresión lógica $p \rightarrow q$, cuyo antecedente es la proposición a la izquierda de '$\rightarrow$' y el consecuente es la proposición a la derecha, puede entenderse como:

- Una regla, como sucede con los refranes, por ejemplo ⟨*si el río suena, entonces piedras tiene*⟩.
- Una cadena de sucesos, como en ⟨*si el jarrón es de barro y se cae, entonces se rompe*⟩.

Al modelar una implicación, es bueno saber que hay muchas maneras en cómo se puede presentar en el lenguaje natural:

$$\underbrace{\boxed{p}}_{\text{antecedente}} \longrightarrow \underbrace{\boxed{q}}_{\text{consecuente}}$$

- Si p, entonces q. ▹ ⟨*Si llueve, entonces se humedece la tierra*⟩.
- q si p. ▹ ⟨*Se humedece la tierra si llueve*⟩.
- Es suficiente q para p. ▹ ⟨*Es suficiente que llueva para que se humedezca la tierra*⟩.
- q porque p. ▹ ⟨*Se humedece la tierra porque llueve*⟩.
- Para q se requiere p. ▹ ⟨*Para que se humedezca la tierra se requiere que llueva*⟩.

Ejemplo 5.9

Para modelar la expresión «Si no es fin de semana, entonces hay que trabajar» se pueden crear las siguientes premisas:

$p \leftarrow\!\shortmid$ ⟨*Es fin de semana*⟩.
$q \leftarrow\!\shortmid$ ⟨*Hay que trabajar*⟩.

La misma expresión se puede escribir con símbolo como $\neg p \rightarrow q$. El valor de verdad se puede calcular con la función implicación haciendo:

1. $p \leftarrow\!\shortmid \mathbf{V}$ ▹ *Premisa*
2. $q \leftarrow\!\shortmid \mathbf{V}$ ▹ *Premisa*

3. $\neg p \rightarrow q$ ▹ *La expresión a evaluar*
4. $\mathbf{F} \rightarrow \mathbf{V} \equiv \rightarrow(\mathbf{F}, \mathbf{V})$
5. $\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}(\mathbf{F}, \mathbf{V})$
6. $\{\lambda p \leftarrow \mathbf{F}, q \leftarrow \mathbf{V} \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}$
7. $\mathbf{V} \texttt{ if } \mathbf{F} \texttt{ else } \mathbf{V}$
8. $\mapsto \mathbf{V}$

Utilizando la función computacional hacemos:

```
>>> es_fin_de_semana = True
>>> hay_que_trabajar = True
>>> p = es_fin_de_semana
>>> q = hay_que_trabajar
>>> impl(neg(p), q)
True
>>>
```

5.4.1 Si y sólo si [9]

Definición 5.4.2 -- Si y sólo si. La función «si y sólo si» es también conocida como «doble implicación», es la función $\leftrightarrow \leftarrow \lambda p, q : q \texttt{ if } p \texttt{ else } \neg q$.

Código 5.5: Si y sólo si

```
"""
Si y sólo si, o doble implicación
"""
ssi = lambda p, q: q if p else neg(q)
```

La función si y sólo si obedece al criterio mostrado en la tabla de verdad, en la columna etiquetada con 9 de la tabla 5.3 en la página 110, que para beneficio del lector se reproduce enseguida, junto con la verificación de la definición computacional.

p	q	$p \leftrightarrow q$
V	**V**	**V**
V	**F**	**F**
F	**V**	**F**
F	**F**	**V**

```
>>> ssi(True, True)
True
>>> ssi(True, False)
False
>>> ssi(False, True)
False
>>> ssi(False, False)
True
>>>
```

La definición se obtiene al observar que cuando p tiene valor **V**, lo que ocurre en los primeros dos renglones, el valor de $p \leftrightarrow q$ es el mismo que el valor que ya ha sido asignado a q, por lo que la primera parte de la regla dice que es q si el valor de p es **V**. Por otra parte, cuando p no es **V**, y en consecuencia es **F**, el valor de $p \leftrightarrow q$ se obtiene con $\neg q$, esto se puede observar en los renglones 3 y 4 de la tabla.

Una expresión lógica que relaciona dos proposiciones p y q mediante la función si y sólo si, se escribe $p \leftrightarrow q$, también es posible escribir la función en notación prefija $\leftrightarrow(p, q)$, considerando los cuatro posibles casos para valores de p y q tenemos:

1. $\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \neg q\}(\mathbf{V},\mathbf{V}) \equiv \{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{V} \bullet q \texttt{ if } p \texttt{ else } \neg q\}$
   $\equiv \{\mathbf{V} \texttt{ if } \mathbf{V} \texttt{ else } \mathbf{F}\}$
   $\mapsto \mathbf{V}$
2. $\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \neg q\}(\mathbf{V},\mathbf{F}) \equiv \{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{F} \bullet q \texttt{ if } p \texttt{ else } \neg q\}$
   $\equiv \{\mathbf{F} \texttt{ if } \mathbf{V} \texttt{ else } \mathbf{V}\}$
   $\mapsto \mathbf{F}$
3. $\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \neg q\}(\mathbf{F},\mathbf{V}) \equiv \{\lambda p \leftarrow \mathbf{F}, q \leftarrow \mathbf{V} \bullet q \texttt{ if } p \texttt{ else } \neg q\}$
   $\equiv \{\mathbf{V} \texttt{ if } \mathbf{F} \texttt{ else } \mathbf{F}\}$
   $\mapsto \mathbf{F}$
4. $\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \neg q\}(\mathbf{F},\mathbf{F}) \equiv \{\lambda p \leftarrow \mathbf{F}, q \leftarrow \mathbf{F} \bullet q \texttt{ if } p \texttt{ else } \neg q\}$
   $\equiv \{\mathbf{F} \texttt{ if } \mathbf{F} \texttt{ else } \mathbf{V}\}$
   $\mapsto \mathbf{V}$

Comparado con la definición de la implicación, la función si y sólo si es más lenta cuando p tiene el valor **F**, pues el resultado debe ser computado haciendo primero una llamada a la función negación. Observa la siguiente comparación entre una invocación a la función implicación y a la función doble implicación con los mismos parámetros:

Pasos	Implicación: $\rightarrowtail(\mathbf{F},\mathbf{V})$	si y sólo si: $\leftrightarrow(\mathbf{F},\mathbf{V})$
1:	$\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}(\mathbf{F},\mathbf{V})$	$\{\lambda p, q \bullet q \texttt{ if } p \texttt{ else } \neg q\}(\mathbf{F},\mathbf{V})$
2:	$\{\lambda p \leftarrow \mathbf{F}, q \leftarrow \mathbf{V} \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}$	$\{\lambda p \leftarrow \mathbf{F}, q \leftarrow \mathbf{V} \bullet q \texttt{ if } p \texttt{ else } \neg q\}$
3:	$\{\mathbf{V} \texttt{ if } \mathbf{F} \texttt{ else } \mathbf{V}\}$	$\{\mathbf{V} \texttt{ if } \mathbf{F} \texttt{ else } \neg\mathbf{V}\}$
4:	$\mapsto \mathbf{V}$	$\neg(\mathbf{V})$
5:		$\{\lambda p \bullet \mathbf{F} \texttt{ if } p \texttt{ else } \mathbf{V}\}(\mathbf{V})$
6:		$\{\lambda p \leftarrow \mathbf{V} \bullet \mathbf{F} \texttt{ if } p \texttt{ else } \mathbf{V}\}$
7:		$\{\mathbf{F} \texttt{ if } \mathbf{V} \texttt{ else } \mathbf{V}\}$
8:		$\mapsto \mathbf{F}$

La doble implicación se utiliza en matemáticas y lógica para expresar equivalencias entre afirmaciones. También se usa comúnmente en demostraciones matemáticas para establecer relaciones bidireccionales entre proposiciones.

Ejemplo 5.10

Para modelar computacionalmente la frase «Un número es par si y sólo si el residuo de la división de ese número por 2 es cero». Primero identificamos las proposiciones moleculares y los términos de enlace:

1. $p \leftarrowtail \langle n \textit{ es par} \rangle$
2. La expresión $\langle \textit{El residuo de la división de un número } n \textit{ por dos} \rangle$ se puede expresar en términos matemáticos como $n \mod 2$, por lo que $\langle q \leftarrowtail n \mod 2 = 0 \rangle$ es la segunda proposición.
3. el término de enlace es la función si y sólo si, que tiene el símbolo $\leftrightarrow$.

Así la expresión original se puede expresar en términos simbólicos como:

1. $\langle n \textit{ es par} \rangle$ es **V**
2. $p \leftarrowtail \mathbf{V}$
3. $\langle q \mod 2 = 0 \textit{ es } V \rangle$
4. $q \leftarrowtail \mathbf{V}$
5. $p \leftrightarrow q$, que se puede escribir $\leftrightarrow(p,q)$, donde:

$$\begin{aligned}\leftrightarrow(p,q) &\equiv \{\lambda p,q \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}(\mathbf{V},\mathbf{V})\\ &\equiv \{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{V} \bullet q \texttt{ if } p \texttt{ else } \mathbf{V}\}\\ &\equiv \{\mathbf{V} \texttt{ if } \mathbf{V} \texttt{ else } \mathbf{V}\}\\ &\mapsto \mathbf{V}.\end{aligned}$$

Computacionalmente podemos hacer las siguientes interacciones:

```
>>> n_es_par = True
>>> p = n_es_par
>>> n_mod_2 = True
>>> q = n_mod_2
>>> ssi(p, q)
True
>>>
```

Una importante aplicación de la función «si y sólo si» es la creación de definiciones, pues se puede aprovechar el comportamiento de equivalencia para introducir un nuevo concepto a partir del resultado de un procedimiento. Recordando que `%` es el operador de módulo en `Python`, se puede crear la definición:

```
esPar(n) = lambda n: n % 2 == 0
```

5.4.2 Disyunción exclusiva

Definición 5.4.3 -- Disyunción exclusiva. La disyunción exclusiva se escribe $\oplus$ y es la función binaria $\lambda p,q \bullet \neg q \texttt{ if } p \texttt{ else } q$.

Código 5.6: La disyunción exclusiva

```
"""
La disyunción exclusiva
"""
xor = lambda neg(q) if p else q
```

La computación realizada en una invocación de la función disyunción exclusiva se muestra en los siguientes cuatro casos:

1. $\{\lambda p,q \bullet \neg q \texttt{ if } p \texttt{ else } q\}(\mathbf{V},\mathbf{V})$
$$\begin{aligned}&\equiv \{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{V} \bullet \neg q \texttt{ if } p \texttt{ else } q\}\\ &\equiv \{\neg\mathbf{V} \texttt{ if } \mathbf{V} \texttt{ else } \mathbf{F}\}\\ &\equiv \neg\mathbf{V}\\ &\mapsto \mathbf{F}\end{aligned}$$
2. $\{\lambda p,q \bullet \neg q \texttt{ if } p \texttt{ else } q\}(\mathbf{V},\mathbf{F})$
$$\begin{aligned}&\equiv \{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{F} \bullet \neg q \texttt{ if } p \texttt{ else } q\}\\ &\equiv \{\neg\mathbf{F} \texttt{ if } \mathbf{V} \texttt{ else } \mathbf{V}\}\\ &\equiv \neg\mathbf{F}\\ &\mapsto \mathbf{V}\end{aligned}$$
3. $\{\lambda p,q \bullet \neg q \texttt{ if } p \texttt{ else } q\}(\mathbf{F},\mathbf{V})$
$$\begin{aligned}&\equiv \{\lambda p \leftarrow \mathbf{F}, q \leftarrow \mathbf{V} \bullet \neg q \texttt{ if } p \texttt{ else } q\}\\ &\equiv \{\neg\mathbf{V} \texttt{ if } \mathbf{F} \texttt{ else } \mathbf{F}\}\\ &\mapsto \mathbf{F}\end{aligned}$$
4. $\{\lambda p,q \bullet \neg q \texttt{ if } p \texttt{ else } q\}(\mathbf{F},\mathbf{F})$
$$\begin{aligned}&\equiv \{\lambda p \leftarrow \mathbf{F}, q \leftarrow \mathbf{F} \bullet \neg q \texttt{ if } p \texttt{ else } q\}\\ &\equiv \{\neg\mathbf{F} \texttt{ if } \mathbf{F} \texttt{ else } \mathbf{V}\}\\ &\mapsto \mathbf{V}\end{aligned}$$

Al explorar los valores de verdad, se tiene la siguiente tabla de verdad, junto con la computación realizada en `Python`:

p	q	$p \oplus q$
V	**V**	**F**
V	**F**	**V**
F	**V**	**V**
F	**F**	**F**

```
>>> ox(True, True)
False
>>> ox(True, False)
True
>>> ox(False, True)
True
>>> ox(False, False)
False
>>>
```

La disyunción exclusiva entre las proposiciones p y q es un caso particular de la disyunción, pues en ocasiones la disyunción no logra modelar exactamente lo que en la realidad sucede con la disyunción exclusiva. Esta función también se conoce como «inequivalencia».

La diferencia entre la disyunción y la disyunción exclusiva radica en el caso en que ambas proposiciones p y q, que son pasadas como parámetros, son de valor **V**, pues en la disyunción $p \vee q \mapsto \mathbf{V}$, mientras que en la disyunción exclusiva $p \oplus q \mapsto \mathbf{F}$, el énfasis de esta operación está en los valores de verdad distintos.

La disyunción exclusiva se utiliza cuando es importante garantizar que mientras una proposición es verdadera, la otra es falsa.

Ejemplo 5.11

«Antonio está en la biblioteca o en la cancha». Siendo $p \longleftrightarrow \langle$*Antonio está en la biblioteca*$\rangle$ y $q \longleftrightarrow \langle$*Antonio está en la cancha*$\rangle$, se puede escribir en símbolos $p \oplus q$ para establecer la proposición $\langle$*o bien Antonio está en la biblioteca o en la cancha*$\rangle$.

Si Antonio no puede estar en ambos lados al mismo tiempo, modelar la frase con una disyunción es menos preciso que modelarlo con una disyunción exclusiva.

```
>>> Antonio_esta_en_la_biblioteca = True
>>> p = Antonio_esta_en_la_biblioteca
>>> Antonio_esta_en_la_cancha = True
>>> q = Antonio_esta_en_la_cancha
>>> xor(p, q)
False
>>>
```

Ejercicios

1. Elabora la tabla de verdad de las siguientes expresiones:
   a) *Ejemplo.* $((p \oplus q) \rightarrow (p \wedge r))$

p	q	r	$p \oplus q$	$p \wedge r$	$((p \oplus q) \rightarrow (p \wedge r))$
V	V	V	F	V	V
V	V	F	F	F	V
V	F	V	V	V	V
V	F	F	V	F	F
F	V	V	V	F	F
F	V	F	V	F	F
F	F	V	F	F	V
F	F	F	F	F	V

   b) $(p \wedge \neg q) \vee (q \wedge \neg p)$
   c) $(p \rightarrow q) \rightarrow (q \rightarrow r)$
   d) $(\neg p \vee \neg q) \wedge (q \leftrightarrow \neg p)$
2. Utilizando una tabla de verdad, demuestra que las expresiones dadas son lógicamente equivalentes.
   a) *Ejemplo.* $((p \oplus q) \oplus r)$ con $((p \vee q) \wedge \neg(p \wedge q)) \oplus r$.

p	q	r	$p \oplus q$	$((p \oplus q) \oplus r)$ ▽	$p \vee q$	$p \wedge q$	$\neg(p \wedge q)$	$(p \vee q) \wedge \neg(p \wedge q)$	$((p \vee q) \wedge \neg(p \wedge q)) \oplus r$ ▽
V	V	V	F	V	V	V	F	F	V
V	V	F	F	F	V	V	F	F	F
V	F	V	V	F	V	F	V	V	F
V	F	F	V	V	V	F	V	V	V
F	V	V	V	F	V	F	V	V	F
F	V	F	V	V	V	F	V	V	V
F	F	V	F	V	F	F	V	F	V
F	F	F	F	F	F	F	V	F	F

   b) $(\neg p \wedge (q \leftrightarrow \neg r))$ con $(p \vee (q \wedge r))$.
   c) $(q \leftrightarrow (p \oplus (\neg q \wedge p)))$ con $(q \wedge \neg p) \vee (\neg q \wedge p)$.
   d) $(r \leftrightarrow \neg s) \wedge (s \oplus \neg r)$ con $(r \vee s) \wedge \neg r$.
3. Las siguientes expresiones lógicas están escritas con dos variables lógicas. Escribe la función lógica primitiva que es lógicamente equivalente a la expresión dada.
   a) *Ejemplo.* $(r \leftrightarrow \neg s) \oplus (s \wedge \neg r)$ es lógicamente equivalente a $r \not\rightarrow s$ que es la columna 2 en la tabla 5.3 en la página 110.
   b) $(f \rightarrow g) \leftrightarrow (g \rightarrow f)$
   c) $(\neg p \vee (q \leftrightarrow p)) \wedge q$
   d) $((q \vee p) \rightarrow \neg p) \oplus (q \leftrightarrow \neg(p \wedge q))$
4. Utiliza una tabla de verdad para demostrar que las siguientes expresiones son tautologías.
   a) *Ejemplo.* $(p \vee q) \vee (\neg p \vee \neg q)$

p	q	$\neg p$	$\neg q$	$p \vee q$	$\neg p \vee \neg q$	$(p \vee q) \vee (\neg p \vee \neg q)$ ▽
V	V	F	F	V	F	V
V	F	F	V	V	V	V
F	V	V	F	V	V	V
F	F	V	V	F	V	V

   Es una tautología porque en la columna marcada solo se encuentra **V**.

b) $((p \vee q) \rightarrow (\neg q \vee \neg p)) \vee p$
c) $\neg((p \oplus q) \leftrightarrow (q \oplus \neg p))$
d) $(p \rightarrow q) \vee ((p \leftrightarrow q) \oplus p)$

5. Escribe una definición para las siguientes funciones lógicas primitivas. Escríbela en términos de matemáticas y en el lenguaje `Python`. Para hacerlo apóyate de la tabla de verdad [ver la tabla 5.3 en la página 110]:

a) Función Falsedad [0]

p	q	$\bot(p,q)$
V	**V**	**F**
V	**F**	**F**
F	**V**	**F**
F	**F**	**F**

Mat.:	$\bot \leftarrow \lambda p, q \bullet$
Python:	`F = lambda p, q:`

b) *Ejemplo.* Función Conjunción [1]: *Ver la página 115*

p	q	$\wedge(p,q)$
V	**V**	**V**
V	**F**	**F**
F	**V**	**F**
F	**F**	**F**

Mat.:	$\wedge \leftarrow \lambda p, q \bullet q$ `if` p `else` **F**
Python:	`y = lambda p, q: q if p else False`

c) Función q-Inhibición [2]

p	q	$\not\rightarrow(p,q)$
V	**V**	**F**
V	**F**	**V**
F	**V**	**F**
F	**F**	**F**

Mat.:	$\not\rightarrow \leftarrow \lambda p, q \bullet$
Python:	`nimpl = lambda p, q:`

d) Función Primera Componente [3]

p	q	$\mathtt{P}(p,q)$
V	**V**	**V**
V	**F**	**V**
F	**V**	**F**
F	**F**	**F**

Mat.:	$\mathtt{P} \leftarrow \lambda p, q \bullet$
Python:	`P = lambda p, q:`

e) Función p-Inhibición [4]

p	q	$\not\leftarrow(p,q)$
V	**V**	**F**
V	**F**	**F**
F	**V**	**V**
F	**F**	**F**

Mat.:	$\not\leftarrow \leftarrow \lambda p, q \bullet$
Python:	`nconv = lambda p, q:`

f) Función Segunda Componente [5]

p	q	$\mathtt{Q}(p,q)$
V	**V**	**V**
V	**F**	**F**
F	**V**	**V**
F	**F**	**F**

Mat.:	$\mathtt{Q} \leftarrow \lambda p, q \bullet$
Python:	`Q = lambda p, q:`

g) *Ejemplo.* Función Disyunción exclusiva [6]: *Ver la página 121*

p	q	$\oplus(p,q)$
V	**V**	**F**
V	**F**	**V**
F	**V**	**V**
F	**F**	**F**

Mat.:	$\oplus \leftarrow \lambda p,q \bullet \neg q \texttt{ if } p \texttt{ else } q$
Python:	`ox = lambda p, q: neg(q) if p else q`

h) *Ejemplo.* Función Disyunción [7]: *Ver la página 114*

p	q	$\vee(p,q)$
V	**V**	**V**
V	**F**	**V**
F	**V**	**V**
F	**F**	**F**

Mat.:	$\vee \leftarrow \lambda p,q \bullet \mathbf{V} \texttt{ if } p \texttt{ else } q$
Python:	`o = lambda p, q: True if p else q`

i) Función NOR [8]

p	q	$\downarrow(p,q)$
V	**V**	**F**
V	**F**	**F**
F	**V**	**F**
F	**F**	**V**

Mat.:	$\downarrow \leftarrow \lambda p,q \bullet$
Python:	`nor = lambda p, q:`

j) *Ejemplo.* Función Si y sólo si [9]: *Ver la página 119*

p	q	$\leftrightarrow(p,q)$
V	**V**	**V**
V	**F**	**F**
F	**V**	**F**
F	**F**	**V**

Mat.:	$\leftrightarrow \leftarrow \lambda p,q \bullet q \texttt{ if } p \texttt{ else } \neg q$
Python:	`ssi = lambda p, q: q if p else neg(q)`

k) Función Negación de la segunda componente [10]

p	q	$\overline{\mathrm{Q}}\,(p,q)$
V	**V**	**F**
V	**F**	**V**
F	**V**	**F**
F	**F**	**V**

Mat.:	$\overline{\mathrm{Q}} \leftarrow \lambda p,q \bullet$
Python:	`nQ = lambda p, q:`

l) Función Conversa [11]

p	q	$\leftarrow(p,q)$
V	**V**	**V**
V	**F**	**V**
F	**V**	**F**
F	**F**	**V**

Mat.:	$\leftarrow \leftarrow \lambda p,q \bullet$
Python:	`conv = lambda p, q:`

m) Función Negación de la primera componente [12]

p	q	$\overline{\mathrm{P}}\,(p,q)$
V	**V**	**F**
V	**F**	**F**
F	**V**	**V**
F	**F**	**V**

Mat.:	$\overline{\mathrm{P}} \leftarrow \lambda p,q \bullet$
Python:	`nP = lambda p, q:`

n) Función Implicación [13]: *Ver la página 117*

p	q	$\rightarrow(p,q)$
V	**V**	**V**
V	**F**	**F**
F	**V**	**V**
F	**F**	**V**

Mat.:	$\rightarrow \leftarrow \lambda p,q \bullet q$ `if` p `else` **V**
Python:	`impl = lambda p, q: q if p else True`

ñ) Función NAND [14]

p	q	$\uparrow(p,q)$
V	**V**	**F**
V	**F**	**V**
F	**V**	**V**
F	**F**	**V**

Mat.:	$\uparrow \leftarrow \lambda p,q \bullet$
Python:	`nand = lambda p, q:`

o) Función Verdad [15]

p	q	$\top(p,q)$
V	**V**	**V**
V	**F**	**V**
F	**V**	**V**
F	**F**	**V**

Mat.:	$\top \leftarrow \lambda p,q \bullet$
Python:	`T = lambda p, q:`

Predicados 6

6.1 Funciones lógicas

El capítulo anterior terminó con la definición matemática y computacional de las funciones lógicas primitivas [páginas 113 y siguientes]. En este capítulo utilizaremos esas funciones para construir otras funciones lógicas que no son primitivas, es decir, que utilizan combinaciones de funciones primitivas para computar el resultado.

6.1.1 Construcción de funciones lógicas prefijas

En la escritura a mano de las expresiones lógicas es común utilizar la notación infija, por varias razones:

1. Los operadores se colocan entre los operandos, esto es una práctica muy común en las operaciones aritméticas como en $4 * 6$, $2 - 7$, $10 + 2$.
2. Es una notación intuitiva, que sigue una lectura de izquierda a derecha, por ejemplo en la expresión «llueve y hay sol» aparece un enlace conjuntivo entre dos proposiciones.
3. Puede ser más fácil de escribir y leer en expresiones simples.
4. Se requieren reglas de jerarquía de operadores y paréntesis para resolver ambigüedades sobre qué operación debe hacerse primero y cual después.

Esta notación pierde utilidad cuando el operador debe actuar sobre un número diferente de operandos, por ejemplo para la suma de los números 43, 52 y 284, se deben utilizar dos símbolos de suma, agrupando cada operación entre paréntesis:

$$(43 + 52) + 284 \quad \text{o} \quad 43 + (52 + 284).$$

Por su parte, la notación prefija, también conocida como notación polaca[1], tiene algunas ventajas sobre la notación infija:

1. Se utiliza un solo operador para todos los operandos, ya sean 0 o más operandos, se sigue la misma notación.
2. Las subexpresiones se escriben en notación prefija, eliminando las ambigüedades sobre qué expresión debe ser evaluada en primer lugar, pues la evaluación se hace de izquierda a derecha.
3. Muchos lenguajes de programación utilizan esta notación, muchas veces encerrando entre paréntesis los argumentos con los que se invoca la función.

Prioridad	Función	Aridad	Infijo	Prefijo	Prefijo computacional
1	Negación	1	No tiene	$\neg p$	`neg(p)`
2	Conjunción	2	$(p \wedge q)$	$\wedge(p,q)$	`y(p,q)`
3	Disyunción	2	$(p \vee q)$	$\vee(p,q)$	`o(p,q)`
4	Implicación	2	$(p \rightarrow q)$	$\rightarrow(p,q)$	`impl(p,q)`
5	Si y sólo si	2	$(p \leftrightarrow q)$	$\leftrightarrow(p,q)$	`ssi(p,q)`
6	Disyunción exclusiva	2	$(p \oplus q)$	$\oplus(p,q)$	`ox(p,q)`

Tabla 6.1: *Notaciones infija y prefija para la invocación de las principales funciones lógicas primitivas.*

Para traducir una expresión lógica infija a una notación prefija, se inicia con una *fbf* con todos los paréntesis. El procedimiento es convertir cada expresión y subexpresión entre paréntesis, empezando desde la expresión que tiene el enlace principal [ver la página 48], luego cada subexpresión de izquierda a derecha siguiendo el mismo criterio.

Ejemplo 6.1

Convertir la expresión infija $((r \vee q) \wedge ((p \rightarrow s) \leftrightarrow (s \wedge \neg q)))$ a prefija.

1. Localizamos el enlace principal, que está localizado a la derecha del segundo paréntesis con índice 1:
$$\underset{12}{(}\underset{}{(}r \vee q\underset{1}{)} \wedge \underset{23}{(}\underset{}{(}p \rightarrow s\underset{2}{)} \leftrightarrow \underset{3}{(}s \wedge \neg q\underset{}{)}\underset{}{)}\underset{210}{)}$$
2. Convertimos la expresión principal, que es una conjunción que tiene dos subexpresiones.
$$(\underbrace{(r \vee q)}_{A} \wedge \underbrace{((p \rightarrow s) \leftrightarrow (s \wedge \neg q))}_{B}) \quad \text{se convierte en} \quad \wedge(\underbrace{(r \vee q)}_{A}, \underbrace{((p \rightarrow s) \leftrightarrow (s \wedge \neg q))}_{B})$$
3. En orden de izquierda a derecha, convertimos las subexpresiones A y B:
   a) Para el caso A se tiene $\vee(r,q)$ y así:
   $$\wedge(\underbrace{(r \vee q)}_{A}, \underbrace{((p \rightarrow s) \leftrightarrow (s \wedge \neg q))}_{B}) \quad \text{se convierte en} \quad \wedge(\vee(r,q), \underbrace{((p \rightarrow s) \leftrightarrow (s \wedge \neg q))}_{B})$$
   b) Para el caso B se tiene la subexpresión $\underset{12}{(}\underset{}{(}p \rightarrow q\underset{1}{)} \leftrightarrow \underset{2}{(}s \wedge \neg q\underset{}{)}\underset{10}{)}$, que tiene el enlace principal a la derecha del segundo 1, este es $\rightarrow$, por lo que convierte:
   $$\underbrace{\leftrightarrow(\underbrace{(p \rightarrow q)}_{B_1}, \underbrace{(s \wedge \neg q)}_{B_2})}_{B}$$
   Trabajando con las subexpresiones de B:

[1] La notación polaca fue inventada por el matemático polaco Jan Łukasiewicz a principios de la década de 1920 [Sim21]. Esta notación es la base para lenguajes de programación como `Lisp` o `Racket`.

1) La expresión $B_1 : (p \rightarrow q)$ se convierte en $\rightarrow(p,q)$
2) La expresión $B_2 : (s \wedge \neg q)$ se convierte en $\wedge(s, \neg(q))$, porque $\neg q$ ya está en prefijo.

Así la expresión B se convierte en: $\underbrace{\boxed{\leftrightarrow(\rightarrow(p,q), \wedge(s, \neg(q)))}}_{B}$

4. Se construye la expresión en prefijo:

$$\wedge(\vee(r,q), \leftrightarrow(\rightarrow(p,q), \wedge(s, \neg(q)))).$$

En la notación computacional se tiene:

```
y(o(r,q),ssi(impl(p,q),y(s,neg(q)))).
```

6.1.2 Definición de funciones lógicas

Como se estableció anteriormente [página 96] las funciones lógicas se definen con variables formales de tipo booleano y devuelven como resultado un valor de tipo booleano; como una subclase de las funciones lógicas, están las funciones lógicas primitivas que se estudiaron en el capítulo anterior [página 127].

Una función lógica, como las funciones λ [página 98], queda definida al declarar dos elementos:

1. Las variables formales, conocidas como parámetros de la función. La cantidad de formales declarados determina la aridad de la función.
2. La expresión lógica. Aquí hay dos situaciones, una es el uso de la función en notación infija, mayormente usada por las personas; la otra es en notación infija, que es utilizada en los lenguajes de programación.

Una función puede ser asignada a un identificador para una manipulación más sencilla. La asignación se hace también con el símbolo $\leftarrow\!\!\shortmid$, como ya se ha hecho anteriormente [página 98].

Ejemplo 6.2

Construir una función lógica con la expresión $((\neg q \leftrightarrow p) \oplus (p \wedge (\neg q \vee p)))$.

(1) Para el caso de una notación infija, simplemente hay que agregar la declaración de las variables formales a la expresión lógica dada:

$$\lambda p, q \bullet ((\neg q \leftrightarrow p) \oplus (p \wedge (\neg q \vee p)))$$

Con el fin de manipular la función en diferentes ocasiones, es mejor asignarle un nombre, el nombre es arbitrario, digamos F:

$$F \leftarrow\!\!\shortmid \lambda p, q \bullet ((\neg q \leftrightarrow p) \oplus (p \wedge (\neg q \vee p)))$$

(2) Para el caso de la notación prefija:

a) Se transforma la expresión infija en una expresión prefija con paréntesis.

$$((\neg q \leftrightarrow p) \oplus (p \wedge (\neg q \vee p))) \quad \text{se convierte en} \quad \oplus(\leftrightarrow(\neg(q), p), \wedge(p, \vee(\neg(q), p)))$$

b) Se declaran las variables formales utilizadas. En este caso se utilizan dos variables formales: p y q; por lo que la función es:

$$\lambda p, q \bullet \oplus(\leftrightarrow(\neg(q), p), \wedge(p, \vee(\neg(q), p))),$$

que ya está lista para ser escrita en el lenguaje `Python` de acuerdo con los nombres establecidos en la tabla 6.1 en la página 128.

```
lambda p,q:ox(ssi(neg(q),p), y(p,o(neg(q), p)))
```

Para utilizar la función, es útil asignarla a un identificador, digamos `F`, sin olvidar que se utiliza el símbolo '=' para una asignación:

```
F = lambda p,q:ox(ssi(neg(q),p), y(p,o(neg(q), p)))
```

```
>>> F = lambda p,q:ox(ssi(neg(q),p), y(p,o(neg(q), p)))
>>> F(True, True)
True
>>> F(True, False)
False
>>> F(False, True)
True
>>> F(False, False)
False
>>>
```

6.1.3 Evaluación de las funciones lógicas

Las funciones lógicas, especialmente las que tienen expresiones moleculares como definición, se pueden evaluar usando un árbol de *fbf* [página 48] o con el método de sustitución [página 99].

Para el uso del árbol de *fbf*, es mejor utilizar la notación prefija, pues las operaciones ya están ordenadas, tomemos por ejemplo la expresión del ejemplo 6.2:

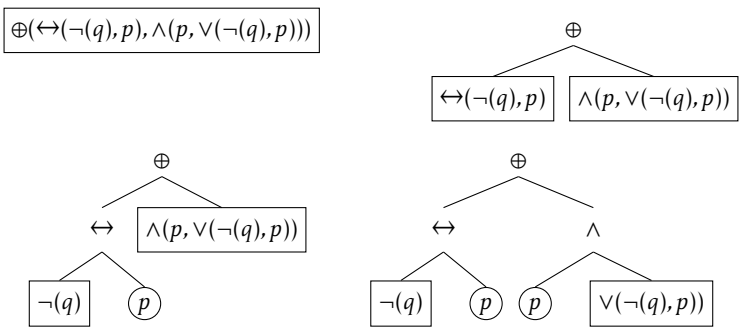
$$\oplus(\leftrightarrow(\neg(q),p),\wedge(p,\vee(\neg(q),p)))$$

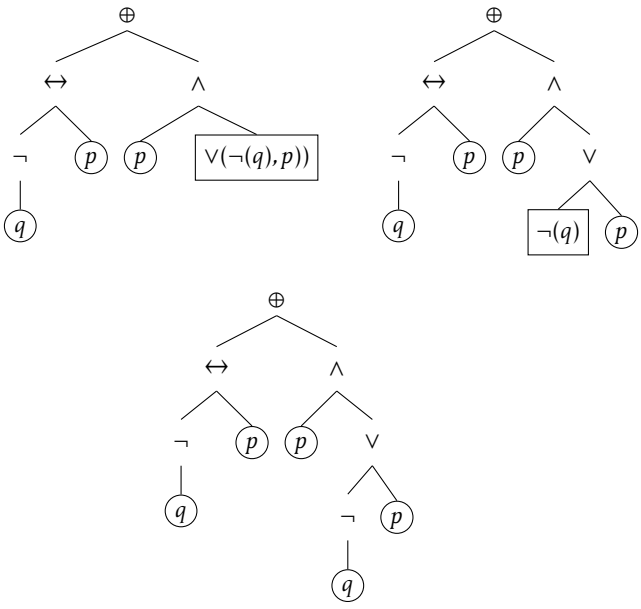
La manera de hacerlo es tomar el operador y crear un subárbol por cada operando. Luego, cada operando o bien es una proposición atómica o una proposición molecular infija, por lo que se sigue el mismo criterio para la construcción del árbol, enseguida se describe el proceso paso a paso:

1. Crear un árbol con el operador más externo como raíz.
2. Crear tantos hijos como aridad tenga el operador. Colocar cada operando como uno de los hijos, preservando el orden de ocurrencia.
3. Para cada hijo:
   a) Si es una literal, entonces se ha llegado al final y encerrarlo en un círculo.
   b) Si es una expresión lógica que inicia con un operando, entonces encerrarlo en un rectángulo y seguir este mismo criterio recursivamente.

Ejemplo 6.3

Crear el árbol *fbf* de la expresión $\oplus(\leftrightarrow(\neg(q),p),\wedge(p,\vee(\neg(q),p)))$.





El procedimiento de construcción del árbol *fbf* termina cuando cada rama del árbol termina en una proposición atómica, estas son las «hojas» del árbol. Una vez que se alcancen las hojas, el procedimiento de evaluación empieza, sustituyendo las variables lógicas por un valor booleano predeterminado y se evalúan las operaciones de abajo hacia arriba, siguiendo el criterio de evaluación de cada operador, de acuerdo a la tabla de verdad [ver la página 110 para el caso de los enlaces binarios].

Ejemplo 6.4

Evalúa la función $F(\mathbf{V},\mathbf{F})$ [ver ejemplo 6.2] usando el árbol *fbf* construido en el ejemplo anterior.

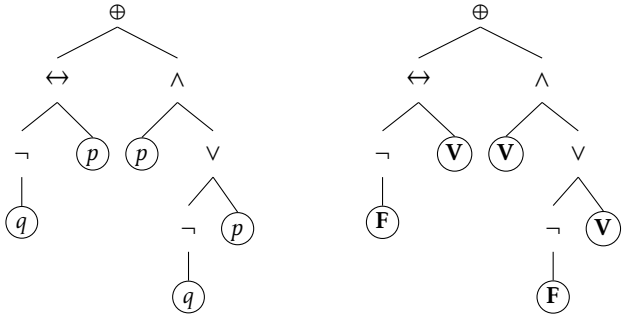
1. Se sustituye el nombre F por su significado $F \leftarrow \lambda p,q \bullet \oplus(\leftrightarrow(\neg(q),p),\wedge(p,\vee(\neg(q),p)))$, y se reescribe la invocación de la función con sus argumentos:

$$\{\lambda p,q \bullet \oplus(\leftrightarrow(\neg(q),p),\wedge(p,\vee(\neg(q),p)))\}(\mathbf{V},\mathbf{F})$$

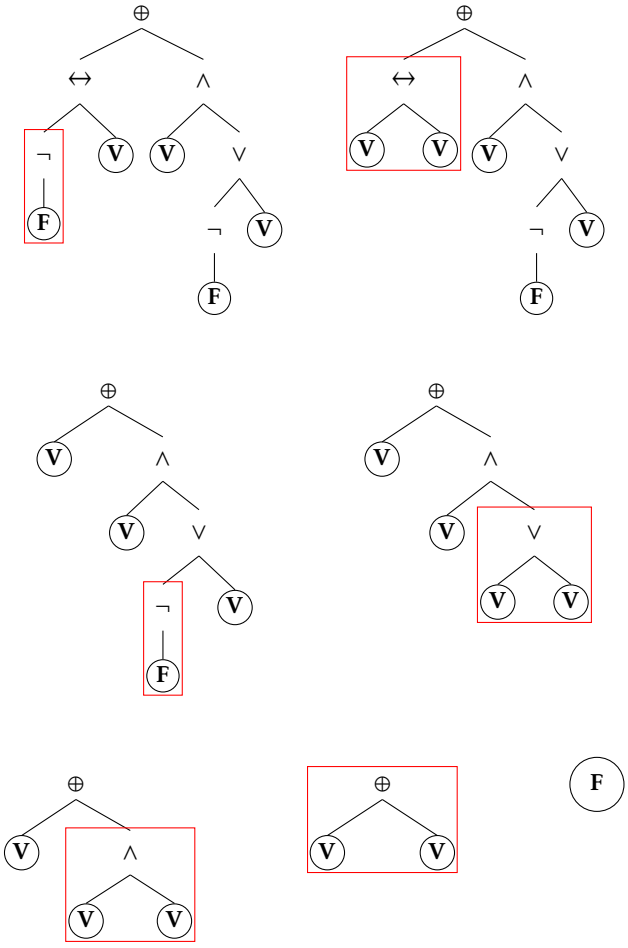
2. Se asocia cada variable formal con su argumento correspondiente.

$$\{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{F} \bullet \oplus(\leftrightarrow(\neg(q),p),\wedge(p,\vee(\neg(q),p)))\}$$

3. En el árbol *fbf*, se sustituye cada literal por su valor asociado.



4. Se resuelven todas las hojas de izquierda a derecha, resolvemos los enlaces de acuerdo con su criterio de interpretación [páginas 109 y 110]:



Así $F(\mathbf{V},\mathbf{F}) \mapsto \mathbf{F}$

Ahora, para evaluar una función lógica creada con una expresión molecular, sin hacer el árbol *fbf*, se procede del siguiente modo:

1. Se sustituye el nombre de la función por su expresión λ.
2. En la parte de las variables formales, se sustituye cada formal por el valor dado en el argumento actual de la invocación.
3. Dentro de la expresión, se sustituye cada ocurrencia de variable por su valor asignado en la sección de variables formales.
4. Mientras la expresión inicie con un operador lógico:
   a) Se lee la expresión de izquierda a derecha, buscando la primera operación no postergada, es decir, el primer operador cuyos operandos sean todos proposiciones; luego se realiza tal operación, de acuerdo al método de sustitución de la página 99.
5. El resultado de la evaluación es la proposición que aparece en primer lugar.

Ejemplo 6.5

Evalúa la función $F(\mathbf{V},\mathbf{F})$ [ver ejemplo 6.2] usando el método de sustitución.

1. Se sustituye el nombre F por su significado $F \leftarrow \lambda p,q \bullet \oplus(\leftrightarrow(\neg(q),p),\wedge(p,\vee(\neg(q),p)))$, y se reescribe la invocación de la función con sus argumentos:

$$\{\lambda p,q \bullet \oplus(\leftrightarrow(\neg(q),p),\wedge(p,\vee(\neg(q),p)))\}(\mathbf{V},\mathbf{F})$$

2. Se sustituye cada variable formal por el valor asignado en la invocación:

$$\{\lambda p \leftarrow \mathbf{V}, q \leftarrow \mathbf{F} \bullet \oplus(\leftrightarrow(\neg(q),p),\wedge(p,\vee(\neg(q),p)))\}$$

3. En la expresión, se sustituye cada variable por su valor asignado:

$$\{\oplus(\leftrightarrow(\neg(\mathbf{F}),\mathbf{V}),\wedge(\mathbf{V},\vee(\neg(\mathbf{F}),\mathbf{V})))\}$$

4. De izquierda a derecha, se busca la primera operación no postergada:

$$\oplus(\leftrightarrow(\boxed{\neg(\mathbf{F})},\mathbf{V}),\wedge(\mathbf{V},\vee(\neg(\mathbf{F}),\mathbf{V})))$$

Se resuelve:

$$(\oplus(\leftrightarrow(\mathbf{V},\mathbf{V}),\wedge(\mathbf{V},\vee(\neg(\mathbf{F}),\mathbf{V}))))$$

5. Se repite el proceso de búsqueda de operaciones no diferidas resolviendo una a una las funciones invocadas:

$$(\oplus(\boxed{\leftrightarrow(\mathbf{V},\mathbf{V})},\wedge(\mathbf{V},\vee(\neg(\mathbf{F}),\mathbf{V})))) \qquad (\oplus(\mathbf{V},\wedge(\mathbf{V},\vee(\neg(\mathbf{F}),\mathbf{V}))))$$

$$(\oplus(\mathbf{V},\wedge(\mathbf{V},\vee(\boxed{\neg(\mathbf{F})},\mathbf{V})))) \qquad (\oplus(\mathbf{V},\wedge(\mathbf{V},\vee(\mathbf{V},\mathbf{V}))))$$

$$(\oplus(\mathbf{V},\wedge(\mathbf{V},\boxed{\vee(\mathbf{V},\mathbf{V})}))) \qquad (\oplus(\mathbf{V},\wedge(\mathbf{V},\mathbf{V})))$$

$$(\oplus(\mathbf{V},\boxed{\wedge(\mathbf{V},\mathbf{V})})) \qquad (\oplus(\mathbf{V},\mathbf{V}))$$

$$(\boxed{\oplus(\mathbf{V},\mathbf{V})}) \qquad (\mathbf{F})$$

$$\mapsto \mathbf{F}$$

Hasta ahora hemos utilizado una función con ciertos argumentos para poder evaluarla, como resultado se obtiene un valor de verdad. Podemos escribir un programa en `Python` que nos ayude a escribir una tabla de verdad para evaluar de manera automática todas las posibles combinaciones de valores asignados a una función lógica de aridad 2, del siguiente modo:

Primero describimos el problema computacional, especificando los valores de entrada y lo que se espera tener al finalizar el cómputo.

Se desea una función que se llame `TV` [*Tabla de Verdad*] que reciba como datos de entrada una función lógica de cualquier aridad, misma que será identificada con el nombre `FLog`. La función `TV` debe escribir un texto en la pantalla, que muestre cada una de las combinaciones de valores de entrada, junto con la evaluación de la función dada. Observa el siguiente ejemplo que muestra la tabla de verdad de una función de aridad 3 definida por la expresión λ:

$$\lambda r,s,t \bullet \wedge(r,\oplus(\leftrightarrow(\neg(r),s),t))$$

Muestra de cómo trabaja la función `TV`

```
>>> F = lambda r,s,t:y(r,ox(ssi(neg(r),s),t))
>>> TV(F, "F")
    r      |    s     |    t     | F(r,s,t)
-----------+----------+----------+----------
   True    |   True   |   True   |   True
   True    |   True   |  False   |  False
   True    |  False   |   True   |  False
   True    |  False   |  False   |   True
  False    |   True   |   True   |  False
  False    |   True   |  False   |  False
  False    |  False   |   True   |  False
  False    |  False   |  False   |  False
>>>
```

6.1.4 El código fuente

La función `TV` propuesta, es de aridad 2, declara dos variables formales: una función [de tipo `callable`], por lo que esta función es un ejemplo de «funcional» [ver la sección 4.1.4 en la página 96]; la otra variable formal es un título [de tipo `str`] como encabezado de la función dada. La función puede ser cualquier función lógica, es decir, una función que recibe como entrada valores booleanos y devuelve un valor booleano. Esta función puede ser dada ya sea como función anónima [función λ] o bien ser dada por el nombre que la define.

Código 6.1: *Tabla de verdad de una función lógica*

```
1  def vTest(a:int, VT: list = [[True], [False]]) -> list:
2      """
3      Calcula los casos que deberán ser probados en la tabla de verdad.
4      a  : int  -- La aridad de la función
5      VT : list -- Los Vectores de pruebas, no es necesario darlo.
6      """
7      if a == 0:
8          return [[]]
9      elif a == 1:
10         return VT
11     else:
12         B = [[True], [False]]
13         L = []
14         for t in VT:
15             for b in B:
16                 L = L + [t+b]
17         return vTest(a-1, L)
18
19 def TV(FLog: callable = lambda x:x, fnom: str = 'F'):
20     """
21     FLog : callable -- Es una función lógica dada como una expresión
22            lambda o bien por su identificador asociado.
23     fnom : str -- Proporciona el encabezado para las columna de la función
24     """
25     args = FLog.__code__.co_varnames
26     for ti in args: print(f"{ti.center(10)}|", end='')
27     print(f'{fnom+"("+",".join(args)+")"}'.center(10))
28     for ti in args: print(f"{'-'*10}+", end='')
29     print(f"{'-'*10}")
30     casos = vTest(len(args))
31     for caso in casos:
32         for val in caso:
33             print(f"{str(val).center(10)}|", end='')
34         print(f"{str(FLog(*caso)).center(10)}")
```

1. La función `TV` es un ejemplo de «funcional», una función que opera con funciones. `Python` tiene la capacidad de crear funciones que reciben funciones como parámetros y además devuelven funciones como resultado [ver capítulos 7 y 8].
2. `Python` puede manejar texto en diferentes maneras, una de ellas es crear texto usando formatos personalizados. Una cadena de texto con formato se escribe `f"texto"`, donde `texto` es la cadena de texto que se va a escribir. Dentro de esta cadena se puede colocar entre llaves, alguna expresión en `Python` cuyo resultado debe ser impreso, por ejemplo `f"hola, mi nombre es {varNom}"`, aquí `varNom` debe ser una variable definida anteriormente que tiene un string, por ejemplo `carNom = "Arturo"`.

6.1.5 Anotaciones sobre el código fuente 6.1

La función del código 6.1 tiene algunas cosas que es necesario comentar, aunque son propias del lenguaje de programación:

1. La función `TV` hace uso de una función auxiliar llamada `vTest`, que recibe como entrada un número entero que representa la aridad de la función. El propósito es generar todos los vectores de prueba con los que se deberá evaluar la función. Sobre esta función auxiliar se tienen las siguientes notas:
   a) El segundo parámetro `VT` es una lista que contiene los vectores de prueba que serán devueltos al terminar el cómputo.
   b) El algoritmo para calcular el vector de pruebas es el siguiente:

$$vTest(a,[VT \hookleftarrow \langle\langle \mathbf{V}\rangle,\langle \mathbf{F}\rangle\rangle]) \mapsto \begin{cases} \langle\langle\rangle\rangle & \text{si } a = 0. \\ VT & \text{si } a = 1. \\ \left.\begin{array}{l} 1.-B \hookleftarrow \langle\langle \mathbf{V}\rangle,\langle \mathbf{F}\rangle\rangle \\ 2.-L \hookleftarrow \langle\rangle \\ 3.-\text{En cada vector } t \text{ de prueba}: \\ \qquad 3.1.-\text{En cada valor } b \text{ en } B: \\ \qquad\qquad 3.1.1.-L \hookleftarrow L + \langle t + b\rangle \\ 4.-vTest(a-1,L) \end{array}\right\} & eoc \end{cases}$$

   Caso $a = 0$**:** Corresponde a una función de aridad 0. No hay tabla de verdad, por lo que se genera una lista de vectores de pruebas con un vector vacío.

   Caso $a = 1$**:** Es para una función de aridad 1. Devuelve una lista de vectores unitarios $\langle\langle \mathbf{V}\rangle,\langle \mathbf{F}\rangle\rangle$.

   Caso $a > 1$**:** Para computar el vector de pruebas se realizan los siguientes 4 pasos:
   (*i*) Se requiere una lista B que contiene los vectores unitarios de prueba.
   (*ii*) La lista resultante L se construye con la siguiente iteración:
   (*iii*) Para cada vector de prueba t en VT, y en cada valor en B, se agrega a la lista L, un vector construido anexando a t cada uno de los vectores unitarios en B. Por ejemplo, si $L \hookleftarrow \langle\langle \mathbf{V}\rangle,\langle \mathbf{F}\rangle\rangle$, este procedimiento agrega cada valor en $B \hookleftarrow \langle\langle \mathbf{V}\rangle,\langle \mathbf{F}\rangle\rangle$ a cada uno en L, produciendo cuatro vectores:

$$\langle\langle \mathbf{V},\mathbf{V}\rangle,\langle \mathbf{V},\mathbf{F}\rangle,\langle \mathbf{F},\mathbf{V}\rangle,\langle \mathbf{F},\mathbf{F}\rangle\rangle.$$

(*iv*) Se recursa el algoritmo ahora con los valores $a \leftarrow\!\shortmid a-1$ y $VT \leftarrow\!\shortmid L$ [el valor recién calculado].

```
>>> vTest(0)
 [[]]
>>> vTest(1)
 [[True], [False]]
>>> vTest(2)
 [[True, True], [True, False], [False, True], [False, False]]
>>> vTest(3)
[[True, True, True],
 [True, True, False],
 [True, False, True],
 [True, False, False],
 [False, True, True],
 [False, True, False],
 [False, False, True],
 [False, False, False]]
>>>
```

2. Los comentarios están estratégicamente colocados justo después del encabezado de la función, porque sirven como ayuda para el programador. Esta ayuda es accesible con el comando `help()`, colocando entre los paréntesis el nombre de la función de la cual se solicita la ayuda, por ejemplo:

```
>>> help(TV)
Help on function TV2 in module __main__:

TV(FLog=<function <lambda> at 0x7f6624153d00>, fnom='F')
    FLog : callable -- Es una función lógica dada como una expresión
           lambda o bien por su identificador asociado.
    tit : str -- Proporciona el encabezado para las columna de la función
>>>
```

3. En la línea 25 se obtiene una tupla que contiene los nombres dados a las variables lógicas de la función que ha sido pasada como argumento. Esa tupla nos servirá para obtener la aridad.
4. Las líneas 26 y 27 son encargadas de imprimir en pantalla el primer renglón que contiene los nombres de las variables y de la función. Cada nombre es centrado en un espacio de 10 caracteres, luego se coloca `"|"` para dividir cada columna, excepto la última, que es compuesta por un string creado con el nombre de la función y entre paréntesis el nombre de las variables separadas por coma.
5. Las líneas 28 y 29 imprimen una línea horizontal.
6. En la línea 30, se utiliza una variable `casos` que tiene los vectores de prueba obtenidos de la función `vTest` para la aridad de la función. Su propósito es generar todas las combinaciones posibles de valores booleanos para las variables de la función lógica `FLog`.
7. Las líneas 31 a 34 imprimen renglón por renglón los casos de prueba y el valor obtenido por la función. Algo notable ocurre en la línea 33. Esta línea tiene como objetivo imprimir cada valor de entrada para la función lógica `FLog` de manera centrada, creando una tabla estructurada y alineada. Los valores se imprimen en la misma línea, separados por una barrra vertical, hasta que se complete la fila. Posteriormente, se imprime el resultado de la función lógica para esa combinación de entradas.

6.2 Construcción de predicados

En gramática, un «predicado» es la parte de la oración que contiene información sobre la acción, el estado o la condición del sujeto. En las proposiciones, el predicado es la parte de la oración que contiene la información principal sobre la acción, el estado o la condición que se está afirmando acerca del sujeto.

El predicado en una expresión declarativa suele incluir al verbo que indica la acción o el estado y, en muchos casos, también puede contener complementos que amplían o especifican la acción o el estado.

Ejemplo 6.6

Los siguientes ejemplos son predicados, pero formalmente no son proposiciones.

- «es muy ágil.» ▹ *No se sabe quién.*
- «$3x > 12$.» ▹ *Falta el valor de x.*
- «es rico y nutritivo.» ▹ *No tiene sujeto.*
- «estudia por la mañana y trabaja por la tarde.» ▹ *Es un predicado compuesto, sin sujeto.*

Un predicado puede llegar a ser una proposición. Cuando se tiene solamente el predicado, hace falta el sujeto para determinar su valor de verdad. Una vez que se tenga la información faltante, el predicado se convierte en proposición.

Por ejemplo, el predicado «es madre». Por sí mismo, no es una proposición, pero solamente hace falta la información del sujeto para que se pueda dar un valor de verdad:

	es madre
sujeto	predicado

Puede servir para crear diferentes proposiciones:

Erika	es madre
sujeto	predicado

La vecina	es madre
sujeto	predicado

De modo que agregando un sujeto al predicado se puede lograr una proposición. En ocasiones se comete un «abuso» al permitir que falte el sujeto de la oración y considerar el texto del predicado como una proposición, por ejemplo la expresión «Llueve y hace sol». Esta expresión es un predicado compuesto y no tiene explícitamente un sujeto; formalmente no es una proposición, pero debido a las bondades del lenguaje, «se supone» que se está hablando de «el día de hoy», por lo que podemos trabajar con ella como si fuera una proposición con sujeto y predicado:

El día de hoy	llueve y hace sol
sujeto	predicado

En la siguiente tabla, las expresiones de la columna izquierda son proposiciones, mientras que aquellas de la columna derecha son predicados.

Proposición	Predicado
El perro es dócil	es dócil
El niño juega y se divierte	juega y se divierte
El problema es difícil	es difícil
Ana Martha hace compras los jueves	hace compras los jueves

Un predicado no tiene valor de verdad, hasta que se conozca el sujeto y la expresión pueda ser evaluada. Un predicado sirve como base para una función, porque el sujeto se proporciona como argumento actual de alguna variable formal. Así, el predicado adquiere la forma de una expresión λ y eventualmente puede ser asociado a identificador que haga la expresión más manejable.

Ejemplo 6.7

Se desea hacer un predicado que determine si el doble de un número dado es mayor que cien. El predicado lo podemos encontrar como «es mayor que cien», y quien realiza la acción [el sujeto de la oración] es *ese* número dado.

Creamos entonces el siguiente predicado, que ha sido relacionado con el símbolo `ndoble`:

$$\texttt{ndoble} \leftarrow\!\shortmid\ \lambda n \bullet 2n > 100$$

Para determinar algún valor de verdad, es necesario proporcionar el valor a la variable formal n, por ejemplo 20, así:

$$\begin{aligned}
&\texttt{ndoble}(20)\\
&\{\lambda n \bullet 2n > 100\}(20)\\
&\{\lambda n \leftarrow\!\shortmid\ 20 \bullet 2n > 100\}\\
&\{2(20) > 100\}\\
&\{40 > 100\}\\
&\mapsto \mathbf{F}
\end{aligned}$$ ■

En `Python` escribimos el siguiente predicado `ndoble = lambda n: 2 * n >100`

```
>>> ndoble = lambda n: 2 * n > 100
>>> ndoble(20)
False
>>>
```

6.2.1 Predicados en acción

Consideremos ahora la expresión «Si la edad de una persona es menor que la edad de otra persona, entonces la primera persona es más joven que la segunda». La siguiente base de datos muestra el registro de algunas personas. Los datos coleccionados son el nombre, la edad y el sexo:

BD

Índice	**Nombre**:`str`	**Edad**:`int`	**Sexo**:`str`	**Salario**:`float`
0:	Ana	19	F	10,000.0
1:	Fer	32	M	15,000.0
2:	Sam	25	F	18,000.0
3:	Gus	11	M	0.0
4:	Mar	24	F	18,000.0

En el ejemplo, se desea hacer un programa que, dada la información de dos personas, determine si una de ellas es más joven que la otra.

Para resolver el problema, describiremos una estrategia que exprese cómo se debe llegar a la respuesta adecuada. Aunque antes debemos anotar algunas observaciones:

1. La base de datos se llama `BD` y es una lista de registros.
2. Cada registro se alcanza por el índice, así por ejemplo $\texttt{BD}_0$ se refiere al primer registro, el que tiene índice 0 y que empieza con el nombre 'Ana'.

3. Cada registro colecciona la información disponible de una persona. También es una lista, por ejemplo el registro ⟨Ana, 19, F⟩ con la información de Ana.
4. Cada dato almacenado se puede ver como una lista de longitud 4.
5. Para identificar una lista no vacía con un nombre, se escribe una asignación $id \leftarrow\!\shortmid \langle id_0, id_1, \ldots, id_n \rangle$; donde id es un identificador. Para obtener un valor id_i con $0 \leq i < n$:
   a) Se sustituye el identificador por la lista.
   b) Se ubica el índice i.
   c) Se obtiene el valor en esa localidad.

Volviendo al problema de escribir una función que devuelva **V** cuando una persona es más joven que otra, definimos entonces el siguiente predicado:

▹ *Siendo a y b registros no vacíos de la forma ⟨ Nombre, Edad, Sexo, Salario ⟩*

$\texttt{esMasJoven} \leftarrow\!\shortmid \lambda a, b \bullet edad\,(a) < edad\,(b)$.

Claro que es necesario definir qué hace la función *edad*, aunque es fácil determinar que esta función devuelve la edad de una persona, tomando la información del registro de persona dado:

▹ *Siendo x un registro no vacío de la forma ⟨ Nombre, Edad, Sexo, Salario ⟩*

$edad \leftarrow\!\shortmid \lambda x \bullet x_1$.

Además de las restricciones propias del lenguaje, se espera que los identificadores no se repitan dentro de la misma definición. Definiciones diferentes pueden considerar el mismo nombre de alguno de sus parámetros formales.

Veamos cómo funciona:

$\texttt{esMasJoven}(\texttt{BD}_1, \texttt{BD}_3)$
$\{\lambda a, b \bullet edad\,(a) < edad\,(b)\}(\texttt{BD}_1, \texttt{BD}_3)$
$\{\lambda a, b \bullet edad\,(a) < edad\,(b)\}(\langle \text{Fer, 32, M}\rangle, \langle \text{Gus, 11, M}\rangle)$
$\{\lambda a \leftarrow\!\shortmid \langle \text{Fer, 32, M}\rangle, b \leftarrow\!\shortmid \langle \text{Gus, 11, M}\rangle \bullet edad\,(a) < edad\,(b)\}$
$\texttt{edad}(\langle \text{Fer, 32, M}\rangle) < \texttt{edad}(\langle \text{Gus, 11, M}\rangle)$
$\{\lambda x \bullet x_1\}(\langle \text{Fer, 32, M}\rangle) < edad\,(\langle \text{Gus, 11, M}\rangle)$
$\{\lambda x \leftarrow\!\shortmid \langle \text{Fer, 32, M}\rangle \bullet x_1\} < edad\,(\langle \text{Gus, 11, M}\rangle)$
$\{\langle \text{Fer, 32, M}\rangle_1\} < edad\,(\langle \text{Gus, 11, M}\rangle)$
$32 < edad\,(\langle \text{Gus, 11, M}\rangle)$
$32 < \{\lambda x \bullet x_1\}(\langle \text{Gus, 11, M}\rangle)$
$32 < \{\lambda x \leftarrow\!\shortmid \langle \text{Gus, 11, M}\rangle \bullet x_1\}$
$32 < \{\langle \text{Gus, 11, M}\rangle_1\}$
$32 < 11$
$\mapsto$ **F** ■

```
BD = [['Ana', 19, 'F', 10000.0],
      ['Fer', 32, 'M', 15000.0],
      ['Sam', 25, 'F', 18000.0],
      ['Gus', 11, 'M', 0.0],
      ['Mar', 24, 'F', 18000.0]]

esMasJoven = lambda a,b: edad(a) < edad(b)
edad = lambda x: x[1]
```

```
>>> esMasJoven(BD[1], BD[3])
False
>>>
```

6.3 Propiedad asociativa

Para lo siguiente, requerimos trabajar con listas de proposiciones. Ahora introduciremos algunas notaciones sobre listas que nos permitirán un manejo e interpretación más sencilla.

Una lista vacía es una lista que no tiene elementos. Denotaremos esta situación como $\langle\,\rangle$. Computacionalmente una lista vacía sirve para:

1. **Como valor inicial.** Frecuentemente se debe realizar un procedimiento recursivo o iterativo, por lo que es importante iniciar con un valor constante. Usualmente se elige un valor mínimo.
2. **Iterador.** En ocasiones se prefiere iterar sobre los elementos de una lista en lugar de los números enteros, como en un ciclo ordinario. El concepto es equivalente, pues los elementos en una lista se relacionan con los números enteros uno a uno.
3. **Como condición base.** Cuando se emplean funciones recursivas, la condición base es fundamental para evitar que la recursión continúe indefinidamente. Cuando se cumple la condición base, el proceso recursivo se detiene y se realizan las actividades finales del procedimiento.

Cuando una lista tiene uno o más elementos, digamos $k \geq 1$ elementos, lo denotamos como $\langle x_0, \ldots, x_{k-1}\,\rangle$. Pero en ocasiones no sabemos cuántos elementos tiene la lista o solo podemos saber que la lista no es vacía. En este caso, sabemos ciertamente que tiene al menos un elemento.

Una lista que tiene al menos un elemento [no vacía], la denotaremos como $\langle x_0|x'\rangle$, donde x_0 representa el primer elemento de la lista, que es aquel que se encuentra más a la derecha; y x' representa a la lista que contiene al resto de los elementos. Si la lista tuviera un solo elemento, x' es $\langle\,\rangle$. Usualmente al elemento x_0 se le conoce como el «primero» y x' es el «resto» de la lista.

Podemos entonces crear un par de funciones que nos permitirán manipular una lista con al menos un elemento [no vacía] y poder acceder a cualquier elemento de la lista con la aplicación de una combinación de funciones:

$$primero \leftarrow\!\shortmid \lambda L \bullet L_0 \quad \triangleright\ Cuando\ L \leftarrow\!\shortmid \langle L_0|L'\rangle.$$
$$resto \leftarrow\!\shortmid \lambda L \bullet L' \quad \triangleright\ Cuando\ L \leftarrow\!\shortmid \langle L_0|L'\rangle.$$

Ejemplo 6.8

Sea $L \leftarrow\!\shortmid \langle 3,2,4,6\,\rangle$ una lista.

- El primero de la lista es 3 y el resto es la lista $\langle 2,4,6\,\rangle$.
- $primero\,(L) \mapsto 3$.
- $resto\,(L) \mapsto \langle 2,4,6\,\rangle$.
- $primero\,(resto\,(L)) \mapsto 2$.

Código 6.2: Primero y resto

```
"""
El primero de una lista
"""
primero = lambda L: L[0]

"""
El resto de una lista
"""
resto = lambda L: L[1:]
```

```
>>> L = [3,2,4,6]
>>> primero(L)
3
>>> resto(L)
[2,4,6]
>>> primero(resto(L))
2
>>>
```

Otro concepto importante en Python que utilizaremos para computar la conjunción generalizada y la disyunción generalizadas es el concepto de lista indeterminada.

Una lista indeterminada es una lista que puede tener 0, 1 o más elementos, se utilizan frecuentemente a la hora de pasar un número no determinado de argumentos a una función. A los argumentos de esta forma se les conoce como *args. args y es la tupla de argumentos de una función. Colocar * antes de la lista desempaqueta los argumentos.

En Python una función puede ser definida con o sin variables formales [parámetros]. Observa las siguientes definiciones de funciones, en donde las variables formales se definen entre paréntesis:

```
def foo1():
    pass
```

y

```
def foo2(a, b):
    pass
```

La función foo1 está definida sin formales, una llamada adecuada a esa función puede ser:

```
>>> foo1()
>>>
```

la segunda función está definida con exactamente 2 formales. Una invocación a la función foo2, que por cierto es de aridad 2, es la siguiente:

```
>>> foo2(1,2)
>>> foo2(1)
Traceback (most recent call last):
File "<stdin>", line 1, in module
TypeError: foo2() missing 1 required positional argument: 'b'
>>>
```

En este caso, el lenguaje considera un error invocar la función con una cantidad de argumentos diferente a 2, ya que se han definido dos argumentos que son obligatorios.

Sin embargo hay otras maneras de definir variables formales en las funciones. Una muy útil es tener una lista no determinada de parámetros, con los que se logra una función de aridad múltiple:

```
def foo3(*L):
   pass
```

Ahora es posible invocar la función foo3 usando 0 o más argumentos:

```
>>> foo3()
>>> foo3(1)
>>> foo3(1,2)
>>> foo3(1,2,3)
>>>
```

Dentro de la función `foo3` se crea una tupla llamada `L` que tiene todos los argumentos.

En `Python`, una tupla es una estructura de datos muy parecida a las listas, pero a diferencia de una lista, en una tupla no se pueden modificar los elementos definidos, por eso una tupla es una lista inmutable. Las tuplas no vacías no necesitan ser escritas entre paréntesis, un caso particular es la tupla unitaria `(1,)` que es lo mismo que escribir simplemente `1,`. Observa que la coma es necesaria para determinar el final de la tupla, cuando hay más de un elemento ya no es necesario terminar con una coma.

En caso de invocar `foo3` sin argumentos, `L` es una tupla vacía. Varios ejemplos de la ejecución del siguiente código servirán para observar qué valores adquiere `L` en cada caso:

```
def foo3(*L):
    """
    Devuelve la tupla de argumentos
    """
    return L
```

```
>>> foo3()
()
>>> foo3(1)
(1,)
>>> foo3(1,2)
(1, 2)
>>> foo3(1,2,3)
(1, 2, 3)
>>>
```

6.3.1 Conjunción generalizada

Digamos que $L \leftarrow \langle\,\rangle|\langle p_0|L'\rangle$ es una lista indeterminada de proposiciones que puede o no ser vacía. Implementaremos ahora el algoritmo 3.21 de la página 82.

Podemos crear un predicado molecular que tome como argumento la lista L y devuelva **V** cuando la lista sea vacía o bien si todas las proposiciones en la lista son **V**; pero debe devolver **F** en cualquier otro caso.

Tenemos entonces el siguiente algoritmo:

Supongamos entonces, que $L \leftarrow \langle\,\rangle|\langle p_0|L'\rangle$ es la lista de proposiciones. Para determinar el valor de verdad de la conjunción generalizada de todas las proposiciones dadas hacemos:

$$\bigwedge(\langle\,\rangle|\langle p_0|L'\rangle) \mapsto \begin{cases} \mathbf{V} & \text{si } L = \langle\,\rangle \\ \bigwedge(L') & \text{si } p_0 \\ \mathbf{F} & eoc \end{cases} \tag{6.1}$$

En dependencia de la lista de proposiciones se debe realizar alguna de las tres opciones, cada una se verifica en turno de arriba hacia abajo, de modo que una opción es probada solamente si es la primera, o bien, la anterior no pudo ser realizada porque no se cumplió la condición establecida.

Podemos describir cada una de las opciones como sigue:

1. Si la lista es vacía, se genera un valor **V**.
2. Si no fue vacía, y si la primera proposición es **V**, seguiremos probando con el resto de las proposiciones.
3. En otro caso, se termina el procedimiento con **F**.

Código 6.3: *Conjunción generalizada*

```
def Y(*LP) -> bool:
    """
    Calcula la conjunción de una lista no determinada de proposiciones.
    """
    if LP == ():
        return True
    elif LP[0]:
        return Y(*LP[1:])
    else:
        return False
```

Crearemos unos predicados para poder utilizar esta conjunción, tomaremos en cuenta la base de datos de la página 138:

1. $mayor18 \leftarrowtail \lambda a \bullet a_1 > 18$ ▷ ***V*** *si n es mayor que 18.*
2. $esMujer \leftarrowtail \lambda a \bullet a_2 = \text{'F'}$ ▷ ***V*** *si* $s = \text{'F'}$.
3. $salarioAlto \lambda a \bullet a_3 \geq 15000.0$ ▷ ***V*** *si* $s \geq 15000.0$

Código 6.4: *BD y predicados*

```
mayor18 = lambda a: a[1]>18
esMujer = lambda a: a[2]=='F'
salarioAlto = lambda a: a[3] >= 15000
```

Ejemplo 6.9

Probaremos la conjunción generalizadas con varios casos:

1. Cuando se proporciona una lista vacía.
2. Deseamos saber si Gus es mujer y tiene un salario alto.
3. Queremos obtener el valor de verdad de la proposición ⟨*Ana tiene mas de 18 años y Sam es mujer y Mar tiene un salario alto*⟩.

```
>>> Y()
True
>>> Y(esMujer(BD[3]), salarioAlto(BD[3]))
False
>>> Y(mayor18(BD[0]), esMujer(BD[2]), salarioAlto(BD[4]))
True
>>>
```

Otra manera de trabajar con la conjunción generalizada es cuando se tiene una lista determinada de proposiciones $\Phi \leftarrowtail \langle p_0, \ldots, p_{n-1} \rangle$ y se desea operar la conjunción sobre todas estas proposiciones. En este caso se puede escribir:

$$\bigwedge \Phi \quad = \quad \bigwedge_{i=0}^{n-1} \langle p_0, \ldots, p_{n-1} \rangle$$

$$\mapsto \quad p_0 \wedge \cdots \wedge p_{n-1}.$$

6.3.2 Disyunción generalizada

Para generalizar la disyunción lógica se puede seguir un razonamiento similar al caso de la conjunción generalizada. Consideremos también una lista no determinada de proposiciones $L \leftarrow \langle\,\rangle|\langle p_0|L'\rangle$ que puede o no ser vacía. El algoritmo es el siguiente:

$$\bigvee(\langle\,\rangle|\langle p_0|L'\rangle) \mapsto \begin{cases} \mathbf{F} & \text{si } L = \langle\,\rangle \\ \mathbf{V} & \text{si } p_0 \\ \bigvee(L') & eoc \end{cases} \tag{6.2}$$

Aunque el algoritmo es ligeramente diferente respecto al de la conjunción generalizada, su comportamiento cambia de manera sustancial, en particular cuando no hay proposiciones para ser evaluadas, es decir, cuando L es una lista vacía.

Podemos describir cada una de las opciones como sigue:

1. Si no hay proposiciones, se genera un valor **F**.
2. Si la primera proposición es **V**, es motivo suficiente para terminar el procedimiento con un valor **V**.
3. En otro caso, seguiremos probando con el resto de las proposiciones.

Código 6.5: Disyunción generalizada

```
def O(*LP) -> bool:
    """
    Calcula la disyunción de una lista no determinada de proposiciones.
    """
    if LP == ():
        return False
    elif primero(LP):
        return True
    else:
        return O(*resto(LP))
```

Otra manera de trabajar con la disyunción generalizada es cuando se tiene una lista determinada de proposiciones $\Phi \leftarrow \langle p_0, \ldots, p_{n-1}\rangle$ y se desea operar la disyunción sobre todas estas proposiciones. En este caso se puede escribir:

$$\begin{aligned} \bigvee \Phi &= \bigvee_{i=0}^{n-1} \langle p_0, \ldots, p_{n-1}\rangle \\ &\mapsto p_0 \vee \cdots \vee p_{n-1}. \end{aligned}$$

Ejercicios

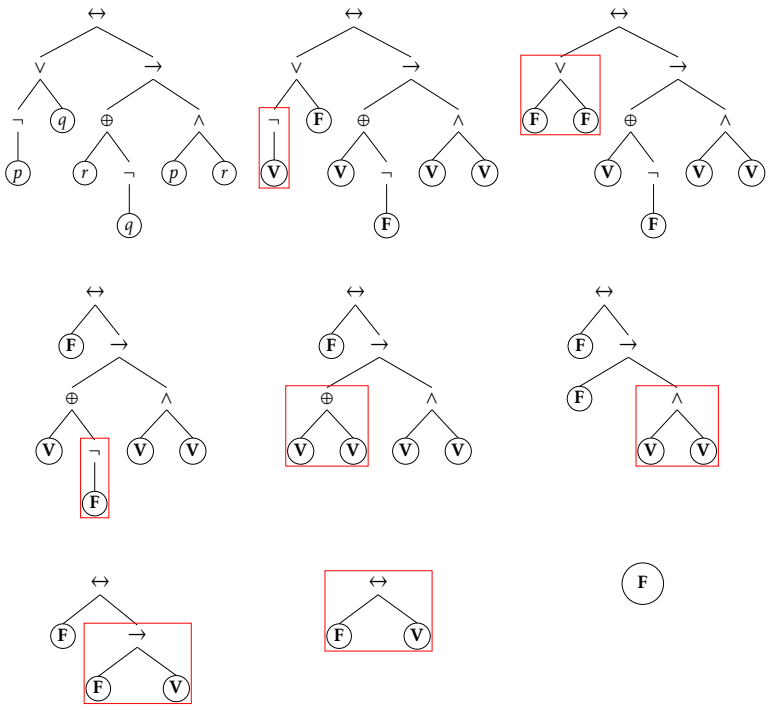
1. Construye una función lógica prefija para las siguientes expresiones lógicas entrefijas.
   a) *Ejemplo.* $((\neg p \vee q) \leftrightarrow ((r \oplus \neg q) \rightarrow (p \wedge r)))$

   $$\lambda p, q, r \bullet \leftrightarrow (\vee(\neg(p), q), \rightarrow(\oplus(r, \neg(q)), \wedge(p, r)))$$

   b) $((p \wedge \neg r) \vee \neg(s \wedge p))$
   c) $(\neg(s \wedge (r \leftrightarrow q)) \rightarrow (\neg q \vee (p \wedge r)))$
   d) $(\neg(q \rightarrow \neg s) \leftrightarrow (\neg q \oplus (s \wedge q)))$
2. Construye una función lógica en notación lambda en lenguaje `Python` para las siguientes expresiones infijas:
   a) *Ejemplo.* $((\neg p \vee q) \leftrightarrow ((r \oplus \neg q) \rightarrow (p \wedge r)))$

   ```
   lambda p,q,r: ssi(o(neg(p), q), impl(ox(r,neg(q)),y(p,r)))
   ```

   b) $(\neg((r \vee q) \wedge (s \leftrightarrow \neg r)))$
   c) $((q \leftrightarrow \neg p) \oplus (\neg s \wedge q))$
   d) $(\neg((q \wedge s) \leftrightarrow (\neg(s \wedge p))))$
3. *Ejemplo.* Utiliza un árbol *fbf* para evaluar las siguientes expresiones:
   a) *Ejemplo.* $((\neg p \vee q) \leftrightarrow ((r \oplus \neg q) \rightarrow (p \wedge r)))$ cuando $p \leftarrow \mathbf{V}$, $q \leftarrow \mathbf{F}$ y $r \leftarrow \mathbf{V}$.



   b) $((a \wedge \neg b) \leftrightarrow (a \vee b))$, cuando $a \leftarrow \mathbf{F}$ y $b \leftarrow \mathbf{V}$.
   c) $(((p \wedge q) \oplus (\neg p \vee r)) \vee (r \rightarrow \neg p))$, cuando $p \leftarrow \mathbf{V}$; $q \leftarrow \mathbf{F}$ y $r \leftarrow \mathbf{F}$.
   d) $((\neg q \wedge (r \vee q)) \oplus (p \rightarrow \neg q))$, cuando $p \leftarrow \mathbf{V}$, $q \leftarrow \mathbf{V}$ y $r \leftarrow \mathbf{F}$.

4. Utiliza el programa de la página 134 para obtener la tabla de verdad de las siguientes funciones lógicas:
   a) *Ejemplo.* $\lambda a,b,c \bullet \leftrightarrow(\neg(b),\rightarrow(a,\neg(c)))$

```
>>> F = lambda a, b, c:ssi(neg(b), impl(a,neg(c)))
>>> TV(F)
    a     |    b     |    c     |F(a,b,c)
----------+----------+----------+----------
   True   |   True   |   True   |   True
   True   |   True   |  False   |  False
   True   |  False   |   True   |  False
   True   |  False   |  False   |   True
  False   |   True   |   True   |  False
  False   |   True   |  False   |  False
  False   |  False   |   True   |   True
  False   |  False   |  False   |   True
>>>
```

   b) $\lambda x,y \bullet \neg(\vee(\wedge(\neg(y),\vee(y,x)),x))$
   c) $\lambda r,s,t \bullet \oplus\Big(\wedge(s,r),\uparrow\big(\downarrow(r,t),s\big)\Big)$
   d) $\lambda a,b \bullet \neg(\oplus(\neg(b),a))$
5. En la siguiente expresión lógica determina cuál es la primera operación no postergada. Enciérrala en un rectángulo.
   a) *Ejemplo.* $\wedge(\leftrightarrow(\mathbf{V},\neg(\mathbf{F})),\neg(\vee(\mathbf{F},\mathbf{V})))$

$$\wedge(\leftrightarrow(\mathbf{V},\boxed{\neg(\mathbf{F})}),\neg(\vee(\mathbf{F},\mathbf{V})))$$

   b) $(\leftrightarrow(\neg(\vee(\oplus(\mathbf{V},\mathbf{F}),\mathbf{V})),\neg(\wedge(\mathbf{V},\mathbf{F}))))$
   c) $(\neg(\wedge(\oplus(\neg(\wedge(\mathbf{V},\mathbf{V}))),\vee(\neg(\mathbf{F}),\leftrightarrow(\mathbf{V},\mathbf{F})))))$
   d) $(\oplus(\neg(\wedge(\oplus(\mathbf{V},\mathbf{F}),\wedge(\mathbf{V},\mathbf{V}))),\neg(\vee(\mathbf{V},\mathbf{F}))))$
6. Evalúa las siguientes funciones lógicas, utilizando el método de sustitución. Para hacer la evaluación, considera los valores de argumentos actuales para cada caso.
   a) *Ejemplo.* $\lambda p,q \bullet \wedge(\leftrightarrow(p,\neg(q)),\neg(\vee(q,p)))$ cuando $p \leftarrowtail \mathbf{V}$ y $q \leftarrowtail \mathbf{V}$.

$$\{\lambda p,q \bullet \wedge(\leftrightarrow(p,\neg(q)),\neg(\vee(q,p)))\}(\mathbf{V},\mathbf{V})$$
$$\{\lambda p \leftarrowtail \mathbf{V}, q \leftarrowtail \mathbf{V} \bullet \wedge(\leftrightarrow(p,\neg(q)),\neg(\vee(q,p)))\}$$
$$\wedge(\leftrightarrow(\mathbf{V},\neg(\mathbf{V})),\neg(\vee(\mathbf{V},\mathbf{V})))$$
$$\wedge(\leftrightarrow(\mathbf{V},\boxed{\neg(\mathbf{V})}),\neg(\vee(\mathbf{V},\mathbf{V})))$$
$$\wedge(\boxed{\leftrightarrow(\mathbf{V},\mathbf{F})},\neg(\vee(\mathbf{V},\mathbf{V})))$$
$$\wedge(\mathbf{F},\neg(\boxed{\vee(\mathbf{V},\mathbf{V})}))$$
$$\wedge(\mathbf{F},\boxed{\neg(\mathbf{V})})$$
$$\boxed{\wedge(\mathbf{F},\mathbf{F})}$$
$$\mapsto \mathbf{F} \qquad \blacksquare$$

   b) $\lambda f,g,h \bullet \oplus(\neg(f),\wedge(g,\vee(h,s)))$, cuando $f \leftarrowtail \mathbf{F}$; $g \leftarrowtail \mathbf{V}$ y $h \leftarrowtail \mathbf{F}$.
   c) $\lambda r,s \bullet \leftrightarrow(\wedge(\oplus(s,\neg(\wedge(r,\neg(s)))),\neg(s)),\vee(\wedge(\neg(s),r),s))$, cuando $r \leftarrowtail \mathbf{F}$ y $s \leftarrowtail \mathbf{F}$.
   d) $\lambda x,y \bullet \wedge(\neg(\vee(y,\neg(x))),\rightarrow(y,\vee(y,\neg(x))))$, cuando $x \leftarrowtail \mathbf{V}$ y $y \leftarrowtail \mathbf{F}$.
7. Para el siguiente ejercicio, considera la base de datos de computadoras que llamaremos c que se muestra enseguida:

C

Índice	Marca	Tipo	Puertos	RAM	MemSec	Fecha
0	HP	Laptop	3	32	512	2016
1	Lenovo	Escritorio	4	8	256	2021
2	IBM	Notebook	2	64	1024	2019
3	Alienware	Laptop	5	16	512	2023
4	Dell	Escritorio	3	32	1024	2022
5	Samsung	Laptop	4	8	256	2021
6	Huawei	Escritorio	2	64	512	2017
7	Apple	Laptop	3	16	1024	2020
8	Lenovo	Laptop	4	8	256	2016

a) *Ejemplo.* Asocia a la variable `regNum` una función λ que reciba una base de datos $\langle BD \rangle$ no vacía y un número entero no negativo $\langle n \rangle$. La función debe devolver el `n`-ésimo registro en la base de datos `BD`. Escribe la función en el lenguaje de lógica computacional y en el lenguaje `Python`.
   a) $regNum \leftarrow \lambda_{\texttt{BD,n}} \texttt{BD}_n$
   b) `regNum = lambda BD, n : BD[n]`

b) *Ejemplo.* Asocia a la variable `puertos` una función λ que reciba un registro de la base de datos $\langle BD \rangle$. La función debe devolver el número de puertos USB que contiene la computadora en el registro que es pasado como argumento. Escribe la función en el lenguaje de lógica computacional y en el lenguaje `Python`.
   a) $puertos \leftarrow \lambda r \bullet r_2$
   b) `puertos = lambda r: r[2]`

c) Asocia a la variable `marca` una función λ que reciba un registro de la base de datos $\langle BD \rangle$. La función debe devolver la marca de la computadora en el registro que es pasado como argumento. Escribe la función en el lenguaje de lógica computacional y en el lenguaje `Python`.

d) Asocia a la variable `P1` un predicado λ que reciba un registro de la base de datos $\langle BD \rangle$. La función debe devolver **V** si se cumple que, si la marca es 'HP', entonces tiene 3 o más puertos USB; de otro modo debe devolver **F**. Escribe el predicado en el lenguaje de lógica computacional y en el lenguaje `Python`.

e) Asocia a la variable `P2` un predicado λ que reciba un registro de la base de datos $\langle BD \rangle$. La función debe devolver **V** si se cumple que, si tiene más de 256 Gb de memoria secundaria, entonces no es laptop. Escribe el predicado en el lenguaje de lógica computacional y en el lenguaje `Python`.

8. Considera la siguiente base de datos. Escribe las funciones y los predicados que se solicitan, luego asígnale un nombre adecuado:

IN

Índice	Nombre	Sexo	Edad	IMC
0	Ana Medina	F	25	23.5
1	Javier Gil	M	30	27.8
2	Diego Saavedra	M	28	29.3
3	Laura Ferrer	F	45	26.7
4	Sofía Vega	F	29	25.3
5	Daniel Ruiz	M	33	31.2

a) `Ejemplo.` Escribe una función que obtenga la información de una persona, que está dada como una lista $\langle Nombre, Sexo, Edad, IMC \rangle$, y que recupere el sexo de la persona.
El sexo de la persona ocupa el segundo lugar en la lista, esto es, ocupa el lugar cuyo índice es 1. Si p es la lista, p_1 se refiere precisamente al sexo.

$$sexo \leftarrow\!\shortmid \lambda p \bullet p_1$$

Así, cuando se requiera el sexo de una persona por ejemplo el de Sofía Vega, bastará hacer $sexo(\texttt{IN}_5)$.

b) Escribe una variable llamada *edad* y asígnale una función λ que recupere la edad de una persona.

c) Escribe una variable llamada *imc* y asígnale una función λ que recupere el IMC de una persona.

La siguiente tabla trata sobre el IMC de una persona adulta.

$18.5 < IMC$	Bajo de peso
$18.5 \leq IMC < 25.0$	Peso saludable
$25.0 \leq IMC < 30.0$	Sobrepeso
$30.0 \leq IMC$	Obesidad

d) Escribe un predicado λ que reciba como entrada la información de una persona y que devuelva **V** si se trata de una persona adulta [con edad mayor o igual a 18 años] y **F** en otro caso.

e) Escribe un predicado λ que reciba como entrada la información de una persona y que devuelva **V** si se trata de un hombre y **F** en otro caso.

f) Escribe un predicado λ que reciba como entrada la información de una persona y que devuelva **V** si se trata de una mujer y **F** en otro caso.

g) Escribe un predicado λ que reciba como entrada la información de una persona y que devuelva **V** si se trata de una mujer con peso saludable y **F** en otro caso.

9. Considera la base de datos `AC` de autos compactos y escribe las siguientes conjunciones o disyunciones generalizadas:

a) Un predicado llamado `esNuevo` que devuelva **V** si el auto es nuevo, sabiendo que un auto se considera «nuevo» si el modelo no es de hace más de 2 años y tiene capacidad para 5 personas y un torque mayor a 200 Nm y tiene transmisión automática.

b) Un predicado llamado `esComodo` que devuelva **V** si el auto es de un modelo 2020 o superior o tiene más de 3 puertas o es de transmisión automática.

AC

Índice	Marca	Modelo	Capacidad	Velocidad	Puertas	Torque	Transmisión
0	Toyota	2020	5	180 km/h	4	200 Nm	Automática
1	Honda	2018	4	200 km/h	2	220 Nm	Manual
2	Ford	2023	5	170 km/h	4	190 Nm	Automática
3	Chevrolet	2021	5	185 km/h	4	210 Nm	Manual
4	BMW	2020	4	220 km/h	2	250 Nm	Automática
5	Mercedes-Benz	2017	5	190 km/h	4	230 Nm	Automática
6	Audi	2021	4	210 km/h	2	240 Nm	Manual
7	Nissan	2019	5	175 km/h	4	195 Nm	Automática
8	Volkswagen	2020	5	180 km/h	4	205 Nm	Manual
9	Kia	2022	4	195 km/h	2	215 Nm	Automática

Funciones de orden superior 7

7.1 Mapeos

Hasta ahora los predicados utilizados se han formulado en singular, como «Jorge salió a jugar» o «El programa de contabilidad esta actualizado». En esta parte del libro, trataremos con predicados que se formulan en plural como en «Los niños de 5^o grado salieron a jugar» o «Los programas de mi computadora están actualizados».

Aunque lingüísticamente parezcan similares, pues ambas expresiones generan proposiciones lógicas, computacionalmente deben ser tratadas de manera diferente. Una expresión lingüística enunciada en plural, genera múltiples expresiones enunciadas en singular, tomemos por ejemplo el enunciado «Los niños de 5^o grado salieron a jugar», que será **V** solamente hasta que se verifique que, en verdad, cada niño de 5^o grado salió a jugar; observando que el predicado está declarado en singular.

Necesitamos entonces una manera de saber el valor de un predicado aplicado en cada uno de los sujetos coleccionados como sujetos de ese predicado.

En esta sección introduciremos la noción de «funciones de orden superior», también conocidas como «Meta funciones» o «funcionales».

Una función de orden superior, en adelante, funcional, es un tipo de función que puede recibir una función y/o producir como salida una función. Así, una función es un funcional si se cumple al menos una de las siguientes condiciones:

1. Recibe como entrada al menos una función.
2. Devuelve como salida una función.

Definición 7.1.1 -- Funcional. Una función Φ es de orden superior si

$$F \in args(\Phi) \vee esFun(\Phi(^{*}args(\Phi)))$$

Donde:

F:	Es una función.
$args$:	Es una función que recibe como entrada cualquier función y devuelve una lista con los argumentos de esa función.
$esFun$:	Es un predicado que devuelve **V** si lo que recibe como entrada es una función; y devuelve **F** en otro caso.
$^{*}args(\Phi)$:	Colocar un $*$ antes de una tupla o lista tiene un efecto de desempaquetamiento. En este caso se coloca $*$ antes de la tupla de los argumentos, con lo que se obtienen los argumentos, fuera de una tupla. Así si $args(\Phi) \mapsto \langle \alpha_1, \ldots, \alpha_a \rangle$, aplicando el desempaquetamiento, $^{*}args(\Phi) \mapsto \alpha_1, \ldots, \alpha_n$.

La definición anterior se aplica a una función Φ para determinar si es o no una función de orden superior; establece que para serlo, debe haber una función F que puede o no ser un funcional, que ocurra como uno de los parámetros de Φ o que el resultado de operar Φ con sus propios argumentos sea una función.

Las funciones de orden superior tienen características que las hacen importantes en el campo de la lógica computacional, programación y otras áreas de las matemáticas. Algunas de estas características son:

Dominio de operación: Las funciones de orden superior operan sobre funciones, es decir, hay funciones entre los parámetros que recibe.

Abstracción y generalidad: Los funcionales permiten abstraer propiedades y trabajar con conceptos más generales. Se pueden estudiar conjuntos de funciones que comparten ciertas características, por ejemplo las funciones polinomiales de grado 2. Esto se aplica fuertemente en el análisis de la complejidad de los algoritmos.

Aplicabilidad: Como los funcionales pueden generar funciones, es posible encapsular ciertas operaciones en funciones más generales, por ejemplo mapeos, filtros y transformadores de datos. El uso de funciones que devuelven funciones es particularmente útil en la creación de composición de funciones de un solo argumento que tengan el mismo comportamiento que una función multiargumentos.

Definición 7.1.2 Un mapeo $\mathbb{m}$ es una función de orden superior porque uno de sus parámetros es una función que se aplica *en cada* argumento creado con una o más listas. Un mapeo que requiere una función F de aridad $\mathtt{a}$ y también $\mathtt{a}$ listas, cada una de ellas de longitud k. Un mapeo tiene la siguiente notación:

$$\mathbb{m}\left(F, L_1^k, \ldots, L_{\mathtt{a}}^k\right)$$

Donde:

F: Es una función de aridad $\mathtt{a}$. Puede ser una función anónima o no.

$L_1^k,\ldots,L_a^k$: Son $\mathtt{a}$ listas, cada una de longitud k. Estas listas contienen los k valores para un solo argumento de la función F. Así con las $\mathtt{a}$ listas:

$$L_1^k,\ldots,L_{\mathtt{a}}^k \quad = \langle \alpha_1,\ldots,\alpha_k \rangle_1,\ldots,\langle \alpha_1,\ldots,\alpha_k \rangle_{\mathtt{a}}$$

Se genera una secuencia de k argumentos de longitud $\mathtt{a}$:

$$\langle\langle \alpha_{1_1},\ldots,\alpha_{1_{\mathtt{a}}} \rangle,\ldots,\langle \alpha_{k_1},\ldots,\alpha_{k_{\mathtt{a}}} \rangle\rangle$$

Con lo que

$$\mathbb{m}\left(F,L_1^k,\ldots,L_{\mathtt{a}}^k\right) \mapsto \langle F\left(^*\langle \alpha_{1_1},\ldots,\alpha_{1_{\mathtt{a}}} \rangle\right),\ldots,F\left(^*\langle \alpha_{k_1},\ldots,\alpha_{k_{\mathtt{a}}} \rangle\right)\rangle$$

Cada una de las $\mathtt{a}$ listas contiene los k valores que deberá tomar uno de los argumentos de la función. Así la i-ésima lista contiene los valores que serán asignados al i-ésimo parámetro de la función F que, recordemos, es de aridad $\mathtt{a}$.

La función F será aplicada k veces, en cada oportunidad con una tupla de argumentos creada con los valores de cada lista. El resultado es una lista de longitud k con los resultados de las k aplicaciones de la función F con cada una de las k tuplas de argumentos [ver la figura 7.1].

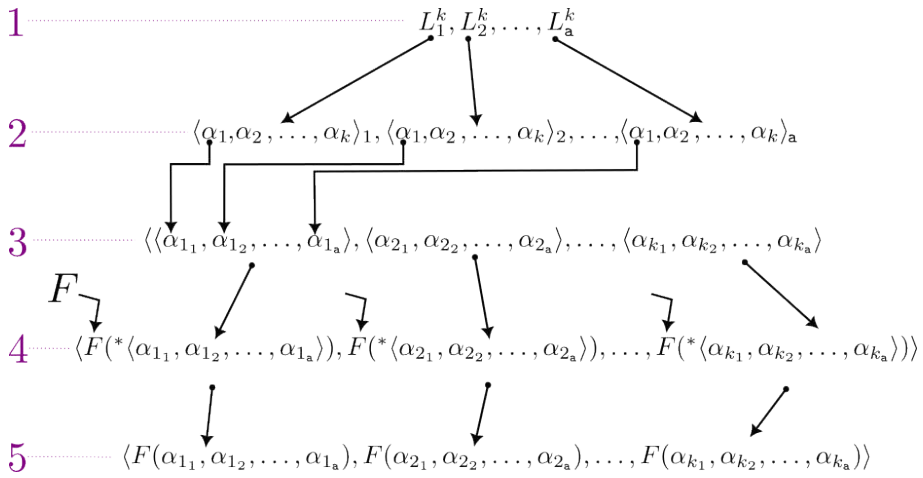


Figura 7.1: *(1) Se proponen $\mathtt{a}$ listas. (2) Cada lista es de longitud k. (3) Cada lista aporta su i-ésimo término para formar una tupla de argumentos. (4) Se integra la invocación de la función F. (5) Se desempaquetan los argumentos y se aplica la función.*

Cuando la aridad de la función F es 1, la notación se simplifica: $\mathbb{m}(F,L)$.

Aquí $L \leftarrow\!\shortmid \langle \alpha_1,\ldots,\alpha_k \rangle$, otorgando las k aplicaciones para la función F.

$$\begin{aligned}\mathbb{M}(F,L) &= \mathbb{M}(F,\langle \alpha_1,\ldots,\alpha_k \rangle)\\ &\mapsto \langle F(\alpha_1),\ldots,F(\alpha_k)\rangle.\end{aligned}$$

Ejemplo 7.1

Tomemos por ejemplo la función $suma4 \leftarrow \lambda x \bullet 4+x$, que toma un número x y devuelve la suma de ese número x con 4. Ahora tomemos la lista de argumentos $\langle 7,8,9,10,11 \rangle$.

El mapeo $\mathbb{M}(suma4, \langle 7,8,9,10,11 \rangle)$ produce: $\langle 11,12,13,14,15 \rangle$ como se muestra en el siguiente cuadro:

Argumento x:	7	8	9	10	11
Aplicación :	$suma4(7)$	$suma4(8)$	$suma4(9)$	$suma4(10)$	$suma4(11)$
	$4+7$	$4+8$	$4+9$	$4+10$	$4+11$
Resultados :	11	12	13	14	15

El mapeo aplica la función F, *en cada* uno de los valores en la lista de argumentos L la expresión clave aquí es «en cada». El resultado es una lista con la evaluación de la función en cada argumento. Claramente la cantidad de evaluaciones que se obtienen es la misma que la cantidad de argumentos en L.

Código 7.1: *Mapeo*

```
1  def mapeo(F, *L):
2      """
3      Aplica una función F a los argumentos pasados por la o las listas en L
4      Ejemplos de aplicación:
5          mapeo(lambda x: x+1, [1,3,5]) --> [2,4,6]
6          mapeo(lambda x, y: x**y, [1,3,5], [2,2,2]) --> [1,9,25]
7      """
8      return list(map(F, *L))
```

```
>>> suma4 = lambda x: 4 + x
>>> mapeo(suma4, [7,8,9,10,11])
[11, 12, 13, 14, 15]
>>> mapeo(lambda x,y:x**y, [1,3,5], [2,2,2])
[1, 9, 25]
>>>
```

La función `map` es una función primitiva que permite procesar los elementos de un iterable y coleccionar el resultado de cada proceso. Devuelve un objeto `map` que contiene los resultados obtenidos al aplicar una función `fun` a los elementos de uno o más iterables `*iterables`, dependiendo de la aridad de la función. Tiene la sintaxis `map(fun, *iterables)`.

```
>>> map(lambda x: 4+x, [7,8,9,10,11])
<map object at 0x7fe3f006bc70>
>>> list(map(lambda x: 4+x, [7,8,9,10,11])
[11, 12, 13, 14, 15]
>>>
```

Al utilizar `map` se deben observar dos reglas fundamentales:

1. La aridad de la función determina la cantidad de listas de argumentos. Si la aridad de la función es diferente que la cantidad de listas, habrá alguna variable formal que se quede sin argumentos.
2. La cantidad de argumentos debe ser la misma en todas las listas de argumentos. Si hay una lista que tenga menos argumentos que las demás, también habrá alguna variable formal sin instanciar.

Ejemplo 7.2

Hay un granjero que tiene que calcular las ganancias que le dejará cada una de sus ovejas para fin de mes. El granjero tiene 9 ovejas y cada oveja le reporta una ganancia que es calculada [quien sabe porqué es así, pero así lo hace] al multiplicar el peso de la oveja por la edad que tiene. Así una oveja de 4 años que pesa 20Kg, le reporta una ganancia de $80. El granjero te da acceso al corral para que las peses y obtengas su edad. Así obtienes las siguientes dos listas:

peso	25.1	24.3	27.2	23.1	26.1	23	28.5	26.9	25.1
edad	4.1	4.2	3.8	5.1	3.9	4.4	4.5	4.5	5.1

Para resolver este problema requerimos el procedimiento y las listas de los argumentos:

1. La función que toma un peso p y una edad e, con los que se calcula el producto de ellos con la función gan, que es de aridad 2:

$$gan \leftarrow \lambda p, e \bullet p * e$$

2. La lista de pesos:

$$pesos \leftarrow \langle\, 25.1, 24.3, 27.2, 23.1, 26.1, 23, 28.5, 26.9, 25.1 \,\rangle$$

y la lista de edades

$$edades \leftarrow \langle\, 4.1, 4.2, 3.8, 5.1, 3.9, 4.4, 4.5, 4.5, 5.1 \,\rangle$$

Al aplicar el mapeo $\mathbb{M}(gan, pesos, edades)$ se obtiene la siguiente secuencia:

$$\begin{aligned}\mathbb{M}(gan, pesos, edades) =& \mathbb{M}(gan, \langle\, 25.1, 24.3, 27.2, 23.1, 26.1, 23, 28.5, 26.9, 25.1 \,\rangle,\\ & \langle\, 4.1, 4.2, 3.8, 5.1, 3.9, 4.4, 4.5, 4.5, 5.1 \,\rangle)\\ =& \langle gan(25.1, 4.1), gan(24.3, 4.2), gan(27.2, 3.8), gan(23.1, 5.1),\\ & gan(26.1, 3.9), gan(23, 4.4), gan(28.5, 4.5), gan(26.9, 4.5), gan(25.1, 5.1)\rangle\\ =& \langle (25.1 * 4, 1), (24.3 * 4.2), (27.2 * 3.8), (23.1 * 5.1), (26.1 * 3.9), (23 * 4.4),\\ & (28.5 * 4.5), (26.9 * 4.5), (25.1 * 5.1)\rangle\\ \mapsto& \langle 102.91, 102.06, 103.36, 117.81, 101.79, 101.2, 128.25, 121.05, 128.01\rangle\end{aligned}$$

Aunque es posible escribir en `Python` la instrucción incluyendo las listas, por motivos de legibilidad se prefiere crear la función `gan`, aquí como una función λ, además de cada lista por separado, la lista `pesos` y la lista `edades`:

```
>>> gan = lambda p,e:p*e
>>> pesos = [25.1, 24.3, 27.2, 23.1, 26.1, 23, 28.5, 26.9, 25.1]
>>> edades = [4.1, 4.2, 3.8, 5.1, 3.9, 4.4, 4.5, 4.5, 5.1]
>>> enCada(gan, pesos, edades)
[102.91, 102.06, 103.36, 117.81, 101.79, 101.2, 128.25, 121.05, 128.01]
>>>
```

7.1.1 Mapeos con predicados

Un mapeo con predicados es un mapeo que utiliza un predicado como función y una o más listas de argumentos; el resultado entonces es una lista de valores booleanos, que resultan de aplicar el predicado con los argumentos construidos a partir de las listas.

Consideremos como ejemplo la tabla `L` siguiente, que contiene información laboral sobre un grupo de personas, que servirá para crear el siguiente predicado:

$$ant5 \leftarrow \lambda p \bullet p_3 > 5.$$

El predicado *ant*5 recibe como entrada una tupla con la información laboral de una persona, tomada de la colección `L` [la tabla de valores siguiente], cada tupla tiene la forma: ⟨Nombre, Puesto, Nivel, Antigüedad, Salario⟩. El resultado devuelto es **V** si el valor en la posición 3 [la antigüedad] es mayor que 5 y **F** en caso contrario.

`L`

Índice	Nombre	Puesto	Nivel	Antigüedad (años)	Salario
0:	Juan	Gerente de Ventas	1	8	14882
1:	María	Analista de Marketing	1	3	14693
2:	Carlos	Desarrollador de Software	3	10	15273
3:	Laura	Contadora	2	15	15220
4:	Pedro	Diseñador Gráfico	3	5	13900
5:	Ana	Asistente Administrativo	1	2	14977
6:	José	Ingeniero de Proyectos	1	20	12517
7:	Luisa	Recursos Humanos	2	12	16902
8:	Roberto	Jefe de Producción	1	30	16000
9:	Isabel	Analista de Datos	2	7	15978

A manera de ilustración, una posible aplicación de la función `ant5` puede ser:

$$
\begin{aligned}
& ant5(\mathtt{L}_0) \\
& \mapsto ant5(\langle \text{Juan, Gerente de Ventas, 1, 8, 14882} \rangle) \\
& \mapsto (\lambda p \bullet p_3 > 5)(\langle \text{Juan, Gerente de Ventas, 1, 8, 14882} \rangle) \\
& \mapsto (\lambda p \leftarrow\!\shortmid \langle \text{Juan, Gerente de Ventas, 1, 8, 14882} \rangle \bullet p_3 > 5 \\
& \mapsto (\langle \text{Juan, Gerente de Ventas, 1, 8, 14882} \rangle_3 > 5) \\
& \mapsto 8 > 5 \\
& \mapsto \mathbf{V} \quad \blacksquare
\end{aligned}
$$

Podemos entonces hacer un mapeo con todos los elementos de la tabla al aplicar el predicado `ant5` en cada registro $\text{M}(ant5, \mathtt{L})$. El resultado es una lista de valores booleanos:

$$
\begin{aligned}
ant5(\mathtt{L}_0) &= ant5(\langle \text{Juan, Gerente de Ventas, 1, 8, 14882} \rangle) && \mapsto \mathbf{V} \\
ant5(\mathtt{L}_1) &= ant5(\langle \text{María, Analista de Marketing, 1, 3, 14693} \rangle) && \mapsto \mathbf{F} \\
ant5(\mathtt{L}_2) &= ant5(\langle \text{Carlos, Desarrollador de Software, 3, 10, 15273} \rangle) && \mapsto \mathbf{V} \\
ant5(\mathtt{L}_3) &= ant5(\langle \text{Laura, Contadora, 2, 15, 15220} \rangle) && \mapsto \mathbf{V} \\
ant5(\mathtt{L}_4) &= ant5(\langle \text{Pedro, Diseñador Gráfico, 3, 5, 13900} \rangle) && \mapsto \mathbf{F} \\
ant5(\mathtt{L}_5) &= ant5(\langle \text{Ana, Asistente Administrativo, 1, 2, 14977} \rangle) && \mapsto \mathbf{F} \\
&\vdots
\end{aligned}
$$

A manera de resumen, cada índice señala un sujeto que será evaluado con la función, obteniendo los siguientes resultados:

i:	0	1	2	3	4	5	6	7	8	9
$\text{M}(ant5, \mathtt{L})$:	**V**	**F**	**V**	**V**	**F**	**F**	**V**	**V**	**V**	**V**

La implementación en `Python` resulta directa:

```
L = [['Juan', 'Gerente de Ventas', 1, 8, 14882],
    ['María', 'Analista de Marketing', 1, 3, 14693],
    ['Carlos', 'Desarrollador de Software', 3, 10, 15273],
    ['Laura', 'Contadora', 2, 15, 15220],
    ['Pedro', 'Diseñador Gráfico', 3, 5, 13900],
    ['Ana', 'Asistente Administrativo', 1, 2, 14977],
    ['José', 'Ingeniero de Proyectos', 1, 20, 12517],
    ['Luisa', 'Recursos Humanos', 2, 12, 16902],
    ['Roberto', 'Jefe de Producción', 1, 30, 16000],
    ['Isabel', 'Analista de Datos', 2, 7, 15978]]

ant5 = lambda p: p[3]>5
```

```
>>> mapeo(ant5, L)
[True, False, True, True, False, False, True, True, True, True]
>>>
```

7.2 Cuantificador Universal

En el esquema de la página 28 se menciona que una categoría de la lógica computacional es la lógica de orden superior. Esta parte de la lógica computacional se dedica al cálculo [y cómputo] de predicados que requieren predicados para determinar el valor de verdad.

En esta sección estudiaremos dos predicados de orden superior que se conocen como «cuantificadores». Estos dos cuantificadores son el «cuantificador universal» y el «cuantificador existencial». Cada uno tiene su propia notación, aunque se parecen mucho una de la otra:

$$\text{Predicados de orden superior} \begin{cases} \text{Cuantificador Universal} \\ \\ \text{Cuantificador Existencial} \end{cases}$$

Definición 7.2.1 -- Cuantificador universal. El cuantificador universal es un predicado de orden superior denotado

$$\forall \alpha_0 \in L_0^k, \ldots, \alpha_{a-1} \in L_{a-1}^k : P(\alpha_0, \ldots, \alpha_{a-1})$$

donde:

- $\forall$: Se puede leer como «para todo».
- $L_0, \ldots, L_{a-1}$: Representan a listas de argumentos, cada una de longitud k.
- P: Es un predicado que también puede ser dado como una expresión lambda.
- $\alpha_0, \ldots, \alpha_{a-1}$: Son los argumentos del predicado.

Con esto se tiene:

$$\forall \alpha_0 \in L_0^k, \ldots, \alpha_{a-1} \in L_{a-1}^k : P(\alpha_0, \ldots, \alpha_{a-1}) \mapsto \bigwedge \mathbb{m}\left(P, L_0^k, \ldots, L_{a-1}^k\right).$$

Expresiones proposicionales que describen una verdad generalizada son modeladas con cuantificadores universales.

Ejemplo 7.3

Algunos ejemplos de predicados que son modelados como cuantificadores universales son:

- ⟨*Todos los perros de mi barrio son obedientes*⟩.
- ⟨*Todos los días de la semana hay tiempo libre*⟩.
- ⟨*Todos los triángulos tienen tres lados*⟩.
- ⟨*Para todo número entero, se cumple que el producto de ese número entero por un número entero constante, también es entero*⟩.
- ⟨*Cualquier número primo es divisible solo por 1 y por sí mismo*⟩.
- ⟨*Todos los estudiantes pasaron el examen*⟩.

Hay observaciones respecto a la notación del cuantificador universal que también son válidas para el cuantificador existencial:

1. Solamente hay un predicado utilizado en el cuantificador. Debe ser enunciado en forma singular. Se trata de verificar ese mismo predicado en cada sujeto coleccionado en la lista de argumentos.
2. Cada una de las listas proporcionan valores de cada uno de los argumentos del predicado.
3. Todas las listas deben tener la misma longitud, para asegurar que todas las aplicaciones del predicado sean correctos.

7.2.1 Transcripción de una expresión universal

Como el lenguaje natural es más flexible que el lenguaje formal, en ocasiones no coinciden las formas de expresar el predicado involucrado en expresiones de índole universal [o existencial]. Por ejemplo en español decimos «Todos los artículos de la tienda son baratos», aquí el predicado «son baratos» está expresado en plural, sin embargo, para el cómputo del valor de verdad, debe ser expuesto en forma singular para cada uno de los elementos en la lista dada, como «[el artículo x de la tienda] es barato», luego cambiar el valor de x para que represente otro artículo y probar el predicado nuevamente, continuar de ese modo hasta terminar.

Existen varias formas en las que se puede presentar un cuantificador universal dentro de un texto. Hay palabras clave que se dicen o escriben, que sugieren la modelación con expresiones lógicas universales, no todas están aquí, pero estas son bastante representativas:

«Todos» por ejemplo en «Todos los mis vecinos son silenciosos». Aquí la lista de argumentos puede ser denotada por V para referirse a «mis *Vecinos*». El predicado $\lambda v \bullet esSilencioso(v)$, donde $esSilencioso(v) \mapsto \mathbf{V}$ cuando el vecino v es silencioso y **F** en otro caso.

$$\forall v \in V : esSilencioso(v)$$

Aquí el predicado es de aridad 1, por lo que solo se requiere una lista de valores.

«Siempre» por ejemplo en «Siempre que hago ejercicio, me siento más saludable». Aquí la lista de argumentos incluye todas las ocasiones en que hago ejercicio, puede ser D, para hacer referencia a «los días en que hago ejercicio»; y el predicado de aridad 1 $\lambda d \bullet meSientoSaludable(d)$, donde d es un día en particular miembro del conjunto D.

$$\forall d \in D : meSientoSaludable(d)$$

Ejemplo 7.4

Escribir en lenguaje formal las siguientes expresiones lógicas:

1. «Todas las computadoras del laboratorio están actualizadas»

$$\forall c \in Lab : estaAct(c)$$

Donde:

$Lab \leftarrow\!\!\shortmid \langle C_0, \ldots, C_m \rangle$:	una lista de tuplas que contienen la descripción de las computadoras en cuestión. Como ilustración, imagina cada descripción como una tupla ⟨Marca, Modelo, SerieNum, InventarioNum, VersionSO⟩.
$estaAct$:	Es el predicado $\lambda c \bullet \mathbf{V}$ `if` $versionSO(c) \geq 2.0$ `else` $\mathbf{F}$.
$versionSO$:	Es una función que recupera la información del campo 4 en la tupla que define la computadora.

2. «Todos los servidores tienen su carga completa», además, se tiene la lista de servidores: $servs \leftarrow\!\!\shortmid \langle \langle s1, 97521024091, 5, 75 \rangle, \langle s2, 132242131226, 7, 82 \rangle, \langle s3, 240252214042, 10, 45 \rangle \rangle$, donde cada descripción de servidor está dado por una tupla ⟨Id,IP,TiempoUso,CargaX100⟩. Una posible expresión lógica que describe la solicitud es:

$$\forall t \in \langle 5, 7, 10 \rangle, c \in \langle 75, 82, 45 \rangle : cargaCompleta(t, c)$$

Donde:

$\langle 5, 7, 10 \rangle$:	Se obtiene haciendo un mapeo sobre la lista de descripciones de servidores, tomando de cada servidor el valor en el campo 2 [el tiempo de uso], se obtiene con $\text{⨍} c \in servs : c_2$.
$\langle 75, 82, 45 \rangle$:	Se obtiene haciendo un mapeo sobre la lista de descripciones de servidores, tomando de cada servidor el valor en el campo 3 [el porcentaje de carga], se obtiene con $\text{⨍} c \in servs : c_3$.
$cargaCompleta$:	Es el predicado $\lambda t, c \bullet \mathbf{V}$ `if` $\wedge (t > 8, c \geq 75)$ `else` $\mathbf{F}$, que indica que un servidor tiene carga completa si tiene un uso diario de más de 8 horas al menos al 75% de capacidad.

7.2.2 Evaluación de un cuantificador universal

Para evaluar un cuantificador universal, debemos tener en cuenta que se requiere hacer la conjunción generalizada sobre una lista de proposiciones, donde cada proposición ha sido el resultado de aplicar un mismo predicado a los argumentos construidos con los elementos de las listas dadas.

Un cuantificador universal $\forall \alpha_0 \in L_0^k, \ldots, \alpha_{a-1} \in L_{a-1}^k : P(\alpha_0, \ldots, \alpha_{a-1})$ puede ser evaluado haciendo los siguientes pasos:

1. Con las listas $L_0^k, \ldots, L_{a-1}^k$, se crea una lista de argumentos

$$L \leftarrow\!\!\shortmid \langle \langle \alpha_0, \ldots, \alpha_{a-1} \rangle_0, \ldots, \langle \alpha_0, \ldots, \alpha_{a-1} \rangle_{k-1} \rangle$$

2. Se crea un mapeo con el predicado P [de aridad `a`] y la lista de argumentos L.

$$\text{⨍}(P, L)$$

3. Se obtienen las k proposiciones, aplicando P sobre cada argumento desempaquetado [ver la página 150]:

$$\begin{array}{c} \langle P(^*\langle \alpha_0, \ldots, \alpha_{a-1} \rangle)_0, \ldots, P(^*\langle \alpha_0, \ldots, \alpha_{a-1} \rangle)_{k-1} \rangle \\ \downarrow \\ \langle P(\alpha_0, \ldots, \alpha_{a-1})_0, \ldots, P(\alpha_0, \ldots, \alpha_{a-1})_{k-1} \rangle \end{array}$$

4. Se devuelve el resultado de la conjunción generalizada sobre la lista de proposiciones.

$$\mapsto \bigwedge \langle P(\alpha_0,\ldots,\alpha_{a-1})_0,\ldots,P(\alpha_0,\ldots,\alpha_{a-1})_{k-1}\rangle$$

De modo que, para obtener el valor de $\forall \alpha_0 \in L_0^k,\ldots,\alpha_{a-1} \in L_{a-1}^k : P(\alpha_0,\ldots,\alpha_{a-1})$, es necesario evaluar el predicado con todos y cada uno de los elementos en L [la lista de argumentos formados de las listas dadas], o bien, expresar un razonamiento válido que permita concluir que todos los elementos en L harán que la proposición $P(\alpha_0,\ldots,\alpha_{a-1}) \mapsto \mathbf{V}$ o en su caso, que no todos lo hacen.

Ejemplo 7.5

Evaluar la expresión «El doble de cualquiera de los números 22, 32, 42 y 52, es mayor que 40». Lo primero que debemos hacer es formalizar la expresión, para hacerlo necesitamos descubrir el predicado, y los valores que alimentarán los argumentos:

1. El predicado: $P \leftarrow \lambda x \bullet 2x > 40$ ▹ *El doble del número x es mayor que 40.*
2. La lista de argumentos: $L \leftarrow \langle 22,32,42,52\rangle$

Con lo que se escribe el predicado:

$$\forall x \in L : P(x).$$

$$\begin{aligned}
\forall x \in L : P(x) &= \forall x \in \langle 22,32,42,52\rangle : 2x > 40 \\
&= \bigwedge \mathbb{M}(\lambda x \bullet 2x > 40, \langle 22,32,42,52\rangle) \\
&= \bigwedge \langle 2(22) > 40, 2(32) > 40, 2(42) > 40, 2(52) > 40\rangle \\
&= \bigwedge \langle 44 > 40, 64 > 40, 84 > 40, 104 > 40\rangle \\
&= \bigwedge \langle \mathbf{V},\mathbf{V},\mathbf{V},\mathbf{V}\rangle \\
&= \mathbf{V} \wedge \mathbf{V} \wedge \mathbf{V} \wedge \mathbf{V} \\
&\mapsto \mathbf{V}
\end{aligned}$$

Código 7.2: Cuantificador universal

```
def paraTodo(P, *L):
    """
    Verifica que todos los elementos de un mapeo de un predicado sobre
    una o varias listas de argumentos sean V.
    Ejemplos de aplicación:
        paraTodo(lambda x: x>1, [3,5,8]) --> True
        paraTodo(lambda x, y: x == y, [1,3,5], [1,3,6]) --> False
    """
    return all(enCada(P,*L))
```

```
>>>  paraTodo(lambda x:2*x>40, [22,32,42,52])
True
>>>
```

En `Python` contamos con una función primitiva llamada `all`, que devuelve **V** si todos los ítems en un iterable son **V**, en otro caso devuelve **F**. Si el objeto iterable es vacío, la función `all()` también devuelve **V**. La sintaxis es `all(iterable)`.

7.3 Cuantificador existencial

Definición 7.3.1 -- Cuantificador universal. El cuantificador universal es un predicado de orden superior denotado

$$\exists \alpha_0 \in L_0^k, \ldots, \alpha_{a-1} \in L_{a-1}^k : P(\alpha_0, \ldots, \alpha_{a-1}),$$

donde:

- $\exists$: Se puede leer como «existe un».
- $L_0^k, \ldots, L_{a-1}^k$: Representan a listas de argumentos, cada una de longitud k.
- P: Es un predicado de aridad a que también puede ser dado como una expresión λ.
- $\alpha_0, \ldots, \alpha_{a-1}$: Son los argumentos del predicado.

Con esto se tiene:

$$\exists \alpha_0 \in L_0^k, \ldots, \alpha_{a-1} \in L_{a-1}^k : P(\alpha_0, \ldots, \alpha_{a-1}) \mapsto \bigvee \mathbb{m}\left(P, L_0^k, \ldots, L_{a-1}^k\right).$$

Una expresión proposicional que describen una verdad no generalizada son modeladas con un cuantificador existencial.

Ejemplo 7.6

Ejemplos de proposiciones que son tomadas de expresiones cotidianas que son modeladas con cuantificadores existenciales son:

- ⟨*Hay días de la semana que no llueve*⟩.
- ⟨*Al menos un estudiante de la universidad es destacado*⟩.
- ⟨*Hay una figura geométrica que tiene más de 5 lados*⟩.
- ⟨*En el cajón de la ropa sucia, hay alguna ropa limpia*⟩.
- ⟨*Hay al menos un pájaro en el jardín*⟩.
- ⟨*Uno de nuestros estudiantes ganó la competencia*⟩.

Así como con el cuantificador universal, las siguientes observaciones son válidas:

1. Se define un solo predicado, y es utilizado en forma singular sobre los argumentos proporcionado por la o las listas.
2. Cada argumento generado por las listas $L_0^k, \ldots, L_{a-1}^k$, debe coincidir en longitud con la aridad del predicado. De no hacerlo, habría alguna variable formal sin ser instanciada, lo que ocasionaría un error en el cómputo.

Transcripción de una expresión existencial

En frecuentes ocasiones se requiere expresar formalmente una proposición enunciada para un grupo de sujetos, donde no se requiere que obligatoriamente todos cumplan el predicado, sino al menos uno. Por ejemplo en español decimos «de los ciudadanos, hay quien no salió a votar», aquí la expresión «no salió a votar» está en plural, porque debe ser verificado para cada sujeto. En el lenguaje natural se pueden expresar enunciados que evocan un cuantificador existencial en distintas maneras. Hay palabras clave que se dicen o escriben, que sugieren la modelación con expresiones lógicas existenciales:

«Algunos», por ejemplo en «Algunos baches de mi calle son profundos». Aquí la lista de argumentos puede ser $B \leftarrow$«los *Baches* de mi calle» y el predicado `esProfundo`,

que puede ser definido como:

$$\texttt{esProfundo} \leftarrow \lambda b \bullet \mathbf{V} \texttt{ if } profundidad(b) > k \texttt{ else } \mathbf{F}$$

donde $profundidad(b)$ devuelve la profundidad del bache en algún sistema métrico y k es una constante. Así el predicado debe devolver **V** si la profundidad del bache b es mayor que tal constante y **F** en otro caso.
De modo que la formalización de la expresión existencial dada puede ser:

$$\exists b \in B : esProfundo(b)$$

«Hay», por ejemplo en «Hay personas en la sala de espera que han esperado mucho». Aquí la lista de argumentos está construida con la información de las personas que están en alguna sala de espera, podríamos identificarla con LE de *Lista de Espera*; y el predicado *haEsperadoMucho* puede ser definido como:

$$haEsperadoMucho \leftarrow \lambda p \bullet \mathbf{V} \texttt{ if } tiempoEsperando(p) > k \texttt{ else } \mathbf{F}.$$

Aquí p es la información de una persona en particular y $tiempoEsperado$ es una función que devuelve alguna cantidad en cierta unidad de tiempo y k, así la expresión existencial se puede transcribir como:

$$\exists p \in \texttt{LE} : haEsperadoMucho(p).$$

Otras formas de encontrar expresiones que involucran cuantificadores existenciales son «al menos uno», como en «al menos uno de los invitados llegó a tiempo»; «algunos», como en «algunos paquetes no se recibieron íntegramente»; «hubo menos alguien que», como en «hubo menos alguien que tomó nota»; «cierto», como en «cierto perro tiene rabia».

Ejemplo 7.7

Escribir en lenguaje formal las siguientes expresiones lógicas:

1. «Hay computadoras en el laboratorio que tienen más de 1 GB de RAM».

$$\exists c \in Lab : estaAct(c)$$

Donde:

$Lab \leftarrow \langle C_0, \ldots, C_k \rangle$:	una lista de tuplas que contienen la descripción de las computadoras en cuestión. Como ilustración, imagina cada descripción como una tupla ⟨Marca, Modelo, SerieNum, InventarioNum, VersionSO⟩.
estaAct:	Es el predicado $\lambda c \bullet \mathbf{V}$ if $versionSO(c) \geq 2.0$ else **F**.
versionSO:	Es una función que recupera la información del campo 4 en la tupla que define la computadora.

En la notación, la única diferencia con el ejemplo 7.4 de la página 157 es el símbolo ∃. En este caso se requiere saber si hay *al menos* una computadora en la colección dada que cumple el predicado, mientras que en el ejemplo7.4 se requería *todas*.

2. «Hay mascotas en el edificio que deben ser vacunadas».

$$\exists m \in E : debeVacunarse(m)$$

Donde:

$E \leftarrow \langle m_0, \ldots, m_k \rangle$:	una lista de tuplas que contienen la descripción de las mascotas en cuestión. Como ilustración, imagina cada descripción como una tupla $\langle$ Nombre, Especie, Raza, Edad, UltimaVacuna $\rangle$.
$debeVacunarse$:	Es el predicado $\lambda m \cdot \mathbf{V}$ `if` $fDif(fechaUV(m), fechaHoy()) \geq 12$ `else` $\mathbf{F}$.
$fDif$:	Es una función que devuelve el número de meses entres dos fechas.
$fechaUV$	Es una función que devuelve la fecha de la última vacuna de la mascota [el campo 4 de la tupla de información de la mascota].
$fechaHoy$	Es una función que devuelve la fecha actual. Es una función de aridad 0, pues no requiere parámetro alguno.

Evaluación de un cuantificador existencial

Al tratar de evaluar una expresión lógica modelada como un cuantificador existencial, al igual que con el cuantificador universal, se remite a la definición donde se observa que es la disyunción generalizada sobre una lista de proposiciones, que son el resultado de la evaluación del predicado con algún argumento formado con los valores de las listas dadas.

Entonces, el cuantificador existencial $\exists \alpha_0 \in L_0^k, \ldots, \alpha_{a-1} \in L_{a-1}^k : P(\alpha_0, \ldots, \alpha_{a-1})$ se evalúa siguiendo los siguientes pasos:

1. Con las listas $L_0^k, \ldots, L_{a-1}^k$, se crea una lista de argumentos

$$L \leftarrow \langle \langle \alpha_0, \ldots, \alpha_{a-1} \rangle_0, \ldots, \langle \alpha_0, \ldots, \alpha_{a-1} \rangle_{k-1} \rangle$$

2. Se crea un mapeo con el predicado P [de aridad `a`] y la lista de argumentos L.

$$\mathbb{M}(P, L)$$

3. Se obtienen las k proposiciones, aplicando P sobre cada argumento desempaquetado [ver la página 150]:

$$\langle P(^*\langle \alpha_0, \ldots, \alpha_{a-1} \rangle)_0, \ldots, P(^*\langle \alpha_0, \ldots, \alpha_{a-1} \rangle)_{k-1} \rangle$$
$$\downarrow$$
$$\langle P(\alpha_0, \ldots, \alpha_{a-1})_0, \ldots, P(\alpha_0, \ldots, \alpha_{a-1})_{k-1} \rangle$$

4. Se devuelve el resultado de la disyunción generalizada sobre la lista de proposiciones.

$$\mapsto \bigvee \langle P(\alpha_0, \ldots, \alpha_{a-1})_0, \ldots, P(\alpha_0, \ldots, \alpha_{a-1})_{k-1} \rangle$$

El resultado es $\mathbf{V}$ si se encuentra al menos un $P(\alpha_0, \ldots, \alpha_{a-1})_i \mapsto \mathbf{V}$.

Ejemplo 7.8

Determinar el valor de verdad de la siguiente expresión «hay al menos un número, entre el 15 16 o 17, que es par».

- Lo primero que debemos hacer es formalizar la expresión y lo primero que aparece es la frase «hay al menos un», lo que sugiere el uso de un cuantificador existencial $\exists$, y para utilizarlo hay que completar la sintaxis con un predicado y una lista [o listas].

$$\exists \boxed{lista[s]} : \boxed{predicado}$$

- Antes de configurar las listas y argumentos, conviene formalizar el predicado. En este ejemplo el predicado está determinado por la expresión «es par», por lo que hacemos:

$$\lambda n \bullet esPar(n)$$

Si hubiera necesidad de expresar la función *esPar*, se puede hacer considerando el procedimiento habitual para verificar que un número es par haciendo $esPar \leftarrowtail \lambda n \bullet (n \bmod 2) = 0$. Así se tiene:

$$\exists \boxed{lista[s]} : esPar(n)$$

Aquí observamos que el predicado *esPar* es de aridad 1, por lo que se requiere solamente una lista.

- Lo siguiente es configurar la lista [o listas] de donde provienen los argumentos para el predicado. Como se ha determinado que el predicado es de aridad 1, la lista de argumentos es $n \in \langle 15, 16, 17 \rangle$ y el cuantificador es:

$$\exists n \in \langle 15, 16, 17 \rangle : esPar(n).$$

Una vez formalizada la expresión lógica se procede a la evaluación:

$$\begin{aligned} \exists \mathbf{x} \in \langle 15, 16, 17 \rangle : esPar(x) \quad &= \bigvee \mathbb{M}(esPar, \langle 15, 16, 17 \rangle) \\ &= \bigvee \langle esPar(15), esPar(16), esPar(17) \rangle \\ &= \bigvee \langle \mathbf{F}, \mathbf{V}, \mathbf{F} \rangle \\ &\mapsto \mathbf{V} \end{aligned}$$

Código 7.3: Cuantificador existencial

```
1  def existeUn(P, *L):
2      """
3      Verifica que al menos uno de los elementos de un mapeo de un predicado
4      sobre una o varias listas de argumentos sea V.
5      Ejemplos de aplicación:
6          existeUn(lambda x: x>1, [-1,0,1,2]) --> True
7          existeUn(lambda x, y: x == y, [1,3,5], [2,3,6]) --> False
8      """
9      return any(enCada(P,*L))
```

```
>>> esPar = lambda x: (x % 2) == 0
>>> existeUn(lambda x:esPar(x), [15, 16, 17])
True
>>>
```

En `Python` contamos con una función primitiva llamada `any`, que devuelve **V** si al menos uno de los ítems en un iterable es **V**, en otro caso devuelve **F**. Si el objeto iterable es vacío, la función `all()` devuelve **F**. La sintaxis es `any(iterable)`.

Ejercicios

1. Escribe en Python los funcionales que se te piden.
   a) *Ejemplo.* Escribe un funcional en Python que reciba como entrada una función F y una lista de strings. El funcional debe aplicar la función F a cada string y devolver como salida la lista con los valores obtenidos de cada aplicación.

```
def aplica_funcion(F, L):
    """
    Aplica F a cada string en la lista L
    """
    resultados = []
    for st in L:
        resultados = resultados + [F(st)]
    return resultados
```

```
>>> invertir_str = lambda texto: texto[::-1]
>>> aplica_funcion(invertir_str, ["hola", "anita lava la tina"])
["aloh", "anit al aval atina"]
>>>
```

   b) Escribe un funcional en Python llamado multiFun, que reciba como entrada un string y una lista de funciones que se puedan aplicar a cadenas de caracteres. Como resultado debes proporcionar una lista de la misma longitud que la lista de funciones que en la i-ésima posición de la lista tenga la aplicación de la i-ésima función de la lista sobre la cadena de entrada. Así por ejemplo:

```
>>> multiFun("hola mundo", [aMayuscula, empiezaConH, sonAlfanum])
['Hola mundo', False, True]
>>>
```

   El funcional multiFun recibe como entrada la cadena de caracteres "hola mundo" y aplica las funciones:
   - aMayuscula : Cambia la primera letra a mayúscula.
   - empiezaConH : Devuelve **V** si el string empieza con 'H'.
   - sonAlfanum : El valor que devuelve es **V** si todos los caracteres del string son alfanuméricos.

2. Utiliza mapeos para resolver los siguientes ejercicios:
   a) *Ejemplo.* Asigna a la variable cubos una función lambda que eleve al cubo una lista de números dada.

```
cubo = lambda n: n**3
cubos = lambda lnums: mapeo(cubo, lnums)
```

```
>>> cubos([1,3,5])
[1, 27, 125]
>>>
```

   b) Asigna a la variable aMays una función lambda que escribe en mayúsculas todas las cadenas de una lista dada.

```
>>> aMays(["hola", "mundo", "jamaica"])
["HOLA", "MUNDO", "JAMAICA"]
>>>
```

   c) Asigna a la variable aInts una función lambda que convierta a números enteros todas las cadenas de números enteros de una lista dada.

```
>>> aInts(["4", "27", "382"])
[4, 27, 382]
>>>
```

d) Asigna a la variable `longs` una función lambda que obtenga la longitud de todas las listas dentro de una lista dada.

```
>>> longs([[1, 4, 78], [3, 6,7, 8, 9], []])
[3, 5, 0]
>>>
```

3. Traduce al lenguaje simbólico de lógica y también al lenguaje `Python` las siguientes expresiones:
   a) *Ejemplo.* «Todos los estudiantes de la clase aprobaron su examen correspondiente» En este caso, se requiere un predicado llamado *aprobo* de aridad 2, que devuelve **V** cuando la calificación de *nom* en el examen *exn* es una nota aprobatoria. Claro, *nom* es el nombre del estudiante y *exn* se refiere al examen correspondiente. Como esta función es de aridad 2, se requieren 2 listas de la misma longitud:
      - $\forall nom \in LN, exn \in LE : aprobo(nom, exn)$. Donde LN es la lista de nombres y LE es la lista de exámenes.
      - En `Python` se podría escribir: `paraTodo(aprobo, LN, LE)`.
   b) «Existe un número primo mayor que 100».
   c) «Todos los pájaros vuelan y ponen huevos».
   d) «Hay al menos una fruta que es roja o amarilla».
   e) «Todas las parejas dadas tienen más de 2 hijos».

La siguiente base de datos se refiere a la información de libros en una biblioteca. Servirá para los siguientes ejercicios.

BIB

Índice	Clave	Fecha	NumPags	ClvEd	conIndice	conRef
0	AB123	2015	250	XYZ	**V**	**F**
1	CD456	2018	180	ABC	**V**	**V**
2	JF789	2020	300	LMN	**F**	**V**
3	GH012	2016	200	UVW	**F**	**F**
4	IJ345	2019	150	PQR	**V**	**F**
5	JK678	2017	220	EFG	**V**	**V**
6	LM901	2014	280	HIJ	**F**	**V**
7	NO234	2021	190	STU	**V**	**F**
8	PQ567	2013	320	RST	**F**	**V**
9	JS890	2012	210	OPQ	**V**	**V**

Clave: Es una palabra clave que enlaza con otra base de datos para obtener otra información del libro, como el título, autor, ISBN y otros.
Fecha: Es la fecha de edición del libro.
NumPags: Es el número de páginas del libro.
ClvEd: Es una palabra clave que identifica la editorial y sus datos.
conIndice: Es un valor booleano que es **V** cuando el libro cuenta con índice y es **F** cuando la obra no tiene índice.

conRef: Es un valor booleano que indica si el libro tiene o no referencias bibliográficas.

4. Con la información de la tabla anterior, escribe un cuantificador y las funciones necesarias para saber si:
   a) *Ejemplo.* Todos los libros de la base `Bib` se publicaron después del año 2000.
   Para esta actividad se requieren las siguientes funciones:
   $fecha \leftarrow \lambda lib \bullet lib_1$ ▷ *Devuelve la fecha de un registro.*
   $\forall \ell \in \texttt{BIB} : fecha(\ell) > 2000$ ▷ *Verifica que todos los libros en `BIB` se hayan publicado después del año 2000.*
   b) Todos los libros cumplen con la condición de que, si tiene más de 200 páginas, entonces tiene índice.
   c) Hay al menos un libro que tenga una clave que inicie con la letra 'J' si tiene referencias bibliográficas.
   d) Todos los libros que cuentan con referencias también se han editado después de 2015.
5. Evalúa paso a paso los siguientes cuantificadores:
   a) *Ejemplo.* $\forall x \in \langle 1,5,2,9 \rangle, y \in \langle 3,2,5,1 \rangle : \lambda x, y \bullet x - y > 0.$

$$\begin{aligned}
\forall x \in \langle 1,5,2,9 \rangle, y \in \langle 3,2,5,1 \rangle : x - y > 0 &= \bigwedge \mathbb{M}(\lambda x, y \bullet x - y > 0, \langle 1,5,2,9 \rangle, \langle 3,2,5,1 \rangle) \\
&= \bigwedge \langle (1-3) > 0,\ (5-2) > 0,\ (2-5) > 0,\ (9-1) > 0 \rangle \\
&= \bigwedge \langle -2 > 0,\ 3 > 0,\ -3 > 0,\ 8 > 0 \rangle \\
&= \bigwedge \langle \mathbf{F}, \mathbf{V}, \mathbf{F}, \mathbf{V} \rangle \\
&= \mathbf{F} \wedge \mathbf{V} \wedge \mathbf{F} \wedge \mathbf{V} \\
&\mapsto \mathbf{F} \quad \blacksquare
\end{aligned}$$

   b) $\exists y \in \langle 34,72,18,39 \rangle : esImpar(y).$
   c) $\forall st \in \langle \texttt{"Juan"}, \texttt{"Jaime"}, \texttt{"Jerusa"} \rangle, \ell \in \langle \texttt{"J"}, \texttt{"J"}, \texttt{"J"} \rangle : iniciaCon(st, \ell).$
   d) $\exists x \in \langle 4,7,2,1 \rangle, y \in \langle 7,2,9,6 \rangle, z \in \langle 34,12,52,81 \rangle : 2x + 3y < z$
6. Una función lógica `F` es *tautología* si la evaluación de la función `F` es **V** para todas las posibles combinaciones de sus argumentos. Por ejemplo:
   `F6`$\leftarrow \lambda p, q \bullet \vee(\vee(p, \neg(p)), \vee(q, \neg(q)))$ que tiene la tabla de verdad:

```
>>> F6 = lambda p,q: o(o(p,neg(p)), o(q, neg(q)))
>>> TV(F6, fnom="F6")
    p      |    q      |  F6(p,q)
-----------+-----------+-----------
   True    |   True    |   True
   True    |  False    |   True
  False    |   True    |   True
  False    |  False    |   True
>>>
```

   Se observa que `F6` es una tautología porque la función devuelve **V** para todos los argumentos.
   Escribe un predicado en `Python` que se llame `esTautologia`, que reciba como entrada una función y que devuelva **V** si la función es una tautología, o bien que devuelva **F** si no lo es.

```
>>> esTautologia(F6)
True
>>>
```

7. Una función lógica `F` es *contradicción* si la evaluación de la función `F` es **F** para todas las posibles combinaciones de sus argumentos. Por ejemplo:
`F7`$\leftarrow \lambda p,q \bullet \vee(\wedge(p,\neg(p)),\wedge(q,\neg(q)))$ que tiene la tabla de verdad:

```
>>> F7 = lambda p,q: o(y(p,neg(p)), y(q, neg(q)))
>>> TV(F7, fnom="F7")
    p      |    q      |  F7(p,q)
-----------+-----------+----------
   True    |   True    |   False
   True    |  False    |   False
  False    |   True    |   False
  False    |  False    |   False
>>>
```

Se observa que `F7` es una contradicción porque la función devuelve **F** para todos los argumentos.
Escribe un predicado en `Python` que se llame `esContradiccion`, que reciba como entrada una función y que devuelva **V** si la función es una falsedad, o bien que devuelva **F** si no lo es.

```
>>> esContradiccion(F7)
True
>>> esContradiccion(impl)
False
>>>
```

Operaciones con cuantificadores 8

8.1 Negación de los cuantificadores

Ya que los cuantificadores son predicados lógicos, una vez evaluados se pueden ver como un valor booleano, pueden ser utilizados como cualquier otra proposición lógica dentro de cualquier expresión con operadores lógicos, como la disyunción o la implicación. En esta sección estudiaremos el caso particular de la negación de los cuantificadores.

La negación del cuantificador universal, que es de la forma $\forall \alpha_0 \in L_0^k, \ldots, \alpha_a \in L_a^k : P(\alpha_0, \ldots, \alpha_a)$, se escribe:

$$\neg\Big(\forall \alpha_0 \in L_0^k, \ldots, \alpha_a \in L_a^k : P(\alpha_0, \ldots, \alpha_a)\Big),$$

que produce **F** cuando $\forall \alpha_0 \in L_0^k, \ldots, \alpha_a \in L_a^k : P(\alpha_0, \ldots, \alpha_a)$ es **V** y produce **V** en el otro caso. Observa que cuando el cuantificador universal es **F**, significa que no todas las evaluaciones del predicado sobre los argumentos han sido **V**. Consecuentemente, si alguna evaluación resultó **F**, entonces el valor del cuantificador universal sería **F** y la negación del cuantificador será **V**. Así que podemos decir:

$$\forall \alpha_0 \in L_0^k, \ldots, \alpha_a \in L_a^k : P(\alpha_0, \ldots, \alpha_a) \mapsto \begin{cases} \mathbf{F} & \text{si } \exists \alpha_0 \in L_0^k, \ldots, \alpha_a \in L_a^k : \neg(P(\alpha_0, \ldots, \alpha_a)) \\ \mathbf{V} & \textit{eoc.} \end{cases}$$

Esto hace que la negación del cuantificador universal se escriba:

$$\neg\Big(\forall \alpha_0 \in L_0^k, \ldots, \alpha_a \in L_a^k : P(\alpha_0, \ldots, \alpha_a)\Big) \mapsto \begin{cases} \mathbf{V} & \text{si } \exists \alpha_0 \in L_0^k, \ldots, \alpha_a \in L_a^k : \neg(P(\alpha_0, \ldots, \alpha_a)) \\ \mathbf{F} & \textit{eoc.} \end{cases}$$

Así se tiene la siguiente equivalencia:

$$\neg\Big(\forall\alpha_0 \in L_0^k,\ldots,\alpha_a \in L_a^k : P(\alpha_0,\ldots,\alpha_a)\Big) \equiv \exists\alpha_0 \in L_0^k,\ldots,\alpha_a \in L_a^k : \neg(P(\alpha_0,\ldots,\alpha_a))$$

Ejemplo 8.1

Las siguientes expresiones sugieren la negación de un cuantificador universal, pero se pueden expresar en términos de un cuantificador existencial:

1. «No todos los taxis son amarillos», de manera equivalente se puede decir «Hay al menos un taxi que no es amarillo».
2. «No todos los asistentes al congreso son ponentes», usando un cuantificador existencial se puede decir de manera equivalente que «Hay asistentes al congreso que no son ponentes».
3. «No siempre que se acaba el gas, es fin de semana» se puede enunciar de manera equivalente como «en ocasiones cuando se acaba el gas, no es fin de semana».
4. $\neg(\forall x \in \langle 1,2,3,4,5\rangle : x > 10)$ es equivalente a $\exists x \in \langle 1,2,3,4,5\rangle : x \leq 10$.
   Observa que $\neg(x > 10)$ es $x \leq 10$.

Por su parte, la negación del cuantificador existencial

$$\exists\alpha_0 \in L_0^k,\ldots,\alpha_a \in L_a^k : P(\alpha_0,\ldots,\alpha_a)$$

se escribe como:

$$\neg\Big(\exists\alpha_0 \in L_0^k,\ldots,\alpha_a \in L_a^k : P(\alpha_0,\ldots,\alpha_a)\Big),$$

cuando se verifica $\exists\alpha_0 \in L_0^k,\ldots,\alpha_a \in L_a^k : P(\alpha_0,\ldots,\alpha_a)$, la negación produce **F** y produce **V** en el otro caso. Observa que cuando el cuantificador existencial es **F**, significa que todas las evaluaciones del predicado sobre los argumentos han sido **F**, por lo que podemos decir:

$$\neg\Big(\exists\alpha_0 \in L_0^k,\ldots,\alpha_a \in L_a^k : P(\alpha_0,\ldots,\alpha_a)\Big) \mapsto \begin{cases} \mathbf{V} & \text{si } \forall\alpha_0 \in L_0^k,\ldots,\alpha_a \in L_a^k : \neg(P(\alpha_0,\ldots,\alpha_a)) \\ \mathbf{F} & \textit{eoc.} \end{cases}$$

y la siguiente equivalencia es correcta:

$$\neg\Big(\exists\alpha_0 \in L_0^k,\ldots,\alpha_a \in L_a^k : P(\alpha_0,\ldots,\alpha_a)\Big) \equiv \forall\alpha_0 \in L_0^k,\ldots,\alpha_a \in L_a^k : \neg(P(\alpha_0,\ldots,\alpha_a))$$

Ejemplo 8.2

Las siguientes expresiones sugieren la negación de un cuantificador existencial, pero también pueden ser escritas en términos de un cuantificador universal:

1. «No hay ni una computadora en el laboratorio que sirva». La manera equivalente es «Todas las computadoras del laboratorio no sirven».
2. «No hay alguien aquí que me ayude»; se puede expresar de manera equivalente como «Todos aquí no me ayudan».
3. «Ni un aula del edificio está disponible a esa hora»; de manera equivalente se puede expresar como «Todas las aulas del edificio no están disponibles».
4. $\neg(\exists a \in \langle 90,60,40\rangle : a < 10)$ es una expresión equivalente a $\forall a \in \langle 90,60,40\rangle : a \geq 10$.
   Observa que $\neg(a < 10)$ es $a \geq 10$.

Observamos que la negación del cuantificador universal se puede escribir en términos del cuantificador existencial, también la negación del cuantificador existencial se puede escribir en términos del universal.

Para hacer esta transliteración correctamente, se pueden seguir estos tres pasos:

① Quitar el símbolo ¬ al inicio de la expresión.
② Cambiar el símbolo ∀ por ∃; o bien, el símbolo ∃ por ∀.
③ Anteponer el símbolo ¬ al predicado de la expresión. Si es necesario, se pueden aplicar otras equivalencias para simplificar la expresión.

Ejemplo 8.3

Escribir $\neg\big(\forall x \in \langle 2,4,6,8,10\rangle : x^2 < 50\big)$ en términos del cuantificador existencial.

① $\forall x \in \langle 2,4,6,8,10\rangle : x^2 < 50$ ▹ *Quitar la negación*
② $\exists x \in \langle 2,4,6,8,10\rangle : x^2 < 50$ ▹ *Sustituir el cuantificador*
③ $\exists x \in \langle 2,4,6,8,10\rangle : \neg(x^2 < 50)$ ▹ *Anteponer la negación en el predicado*
≡ $\exists x \in \langle 2,4,6,8,10\rangle : x^2 \geq 50$

Elimina la negación al inicio de la siguiente expresión: «No hay un asistente que no haya pagado su entrada».

① «Hay un asistente que no ha pagado su entrada».▹ *Quitar la negación*
② «Todos los asistentes no han pagado su entrada».▹ *Sustituir el cuantificador*
③ «Todos los asistentes no no han pagado su entrada».▹ *Anteponer la negación en el predicado*
≡ «Todos los asistentes han pagado su entrada».

Al utilizar cuantificadores afectados por la negación, se prefiere escribirlas en términos del otro cuantificador con el objetivo de quitar las negaciones al inicio. Las negaciones al principio de una expresión lógica con cuantificadores pueden complicar la comprensión y el razonamiento. Al utilizar negaciones al principio, la lógica se vuelve más difícil de analizar y entender intuitivamente. Esto se debe a que las negaciones iniciales pueden requerir el uso de reglas adicionales, como la ley de De Morgan, para simplificar la expresión y entender su significado.

8.2 Cuantificadores anidados

Ya que un cuantificador, sin importar si es universal o existencial, se reduce a un valor booleano, es posible utilizarlo en situaciones en donde se requiere evaluar una expresión que deba resultar un valor booleano, como cualquier otro predicado. En esta sección estudiaremos el caso cuando un cuantificador ocurre dentro de otro.

Cuando hay un cuantificador que aparece dentro del ámbito de otro, se dice que hay cuantificadores anidados. Como tenemos dos cuantificadores, se pueden tener cuatro casos:

1. Cuantificador universal dentro de un cuantificador universal.
2. Cuantificador universal dentro de un cuantificador existencial.
3. Cuantificador existencial dentro de un cuantificador universal.
4. Cuantificador existencial dentro de un cuantificador existencial.

Sin embargo, la manera en que se trabaja con ellos es similar. Aquí te muestro la forma general de un cuantificador universal [cuantificador externo] que encierra un cuantificador existencial [cuantificador interno]:

$$\forall \alpha_0 \in L_0^k, \ldots, \alpha_{\mathtt{a}-1} \in L_{\mathtt{a}-1}^k : [\exists \beta_0 \in L_0^k, \ldots, \beta_{\mathtt{b}-1} \in L_{\mathtt{a}-1}^k : P(\alpha_0, \ldots, \alpha_{\mathtt{a}-1}, \beta_0, \ldots, \beta_{\mathtt{b}-1})]$$

Una regla que se debe observar a la hora de definir cuantificadores anidados es que las variables de definición de un cuantificador no pueden ser definidas como variables de definición de otro cuantificador anidado. Hacerlo significaría incurrir en ambigüedad puesto que no se podría determinar la naturaleza de tal variable, por ejemplo, en $\forall p \in \mathtt{Perros} : \forall p \in \mathtt{Paises} : origen(p, p)$, no se podría determinar si p es un perro, o bien p es un país.

Observa que el predicado $\lambda \alpha_0, \ldots, \alpha_{\mathtt{a}-1}, \beta_0, \ldots, \beta_{\mathtt{b}-1} \bullet P(\alpha_0, \ldots, \alpha_{\mathtt{a}-1}, \beta_0, \ldots, \beta_{\mathtt{b}-1})$ incluye los parámetros definidos tanto en el cuantificador externo como el interno.

Un ejemplo de una expresión que se puede modelar con un cuantificador universal–existencial es «Toda persona en el mundo tiene a alguien del mundo que la quiere». Aquí la frase «*Toda*...» se refiere al cuantificador universal, que en este caso es el cuantificador externo; la frase «... *tienen a alguien*...» se refiere a un cuantificador existencial, que en este caso es el cuantificador interno.

Podemos formalizar esta expresión para utilizar símbolos que permitan una mejor abstracción.

$$\forall a \in \mathtt{M} : [\exists b \in \mathtt{M} : quiere(b, a)]$$

Donde:

M:	Es una lista de las personas *en el mundo*.
p :	Es una instancia de persona en el mundo M.
a :	Es otra instancia de persona en M.
quiere:	Es el nombre de un predicado de aridad 2 que es **V** si la persona b quiere a la persona a y es **F** en otro caso.

Otro ejemplo donde se anidan cuantificadores, ahora en sentido inverso, es: «Existe una fábrica de ropa en México donde todos los productos de su catálogo son hechos en México». Podemos formalizar esta idea como:

$$\exists f \in \mathtt{M} : [\forall p \in \mathtt{C} : hechoEnMx(p, f)]$$

Donde:

M:	Es una lista de fábricas de ropa en México.
f :	Es una instancia de fábrica en M.
C:	Es un catálogo de productos; en cierta manera es una lista de productos que son hechos en una fábrica determinada.
p :	Es una instancia de producto en el catálogo C.
hechoEnMx:	Es el nombre de un predicado de aridad 2 que es **V** si el producto p de la fábrica f ha sido hecho en México y es **F** en otro caso.

En este otro caso, el cuantificador universal actúa como cuantificador interno, mientras que el cuantificador existencial es el externo. Observamos en ambos casos que los predicados con el que se prueban los elementos, ahora incluyen variables de ambos cuantificadores.

Cada cuantificador es definido con variables que tienen validez en su propio ámbito, por lo que es importante utilizar las variables en un ámbito en donde las variables tengan valor.

8.2.1 Ámbito de las variables

Un aspecto importante a considerar en los cuantificadores anidados es el ámbito de las variables. Cuando hay cuantificadores anidados, las variables de cada cuantificador tienen un ámbito de definición en donde su valor actual puede afectar a otras variables.

Una variable que es colocada fuera de su ámbito de validez no está ligada, es decir, asociada con un valor, por lo que utilizarla es causa de un error.

Vamos por partes. Tomemos como inicio una expresión de cuantificador, y no es importante por ahora el tipo de cuantificador porque el ámbito de la variable no depende del tipo de cuantificador. En la siguiente expresión, el ámbito del cuantificador está encerrada en un rectángulo.

$$\forall x \in \mathtt{D} : \boxed{P(x)}.$$

Así, el ámbito de definición de la variable x se encuentra después de los dos puntos y hasta el punto final. Fuera de ese rectángulo el uso de la variable no está definido. Entre la marca de dos puntos y el predicado no es posible escribir algo distinto a un predicado.

Por supuesto que el predicado $P(x)$ puede ser un predicado compuesto, por ejemplo $P \leftarrow \lambda x \bullet \rightarrow (G(x), H(x))$, donde G y H son predicados. Observa la misma expresión pero ya haciendo la sustitución de P.

$$\forall x \in \mathtt{D} : \boxed{G(x) \rightarrow H(x)}.$$

Bien, avancemos un nivel de anidamiento. En cuantificadores anidados, el ámbito de definición de las variables también es anidado.

$$\exists y \in \mathtt{E} : \boxed{\forall x \in \mathtt{D} : \boxed{P(x,y)}}.$$

La regla fundamental en el uso de las variables en cuantificadores anidados es: Las variables definidas en ámbitos más internos, no pueden ser utilizadas en ámbitos más externos. Pero las variables definidas en ámbitos más externos sí pueden ser usadas en ámbitos más internos.

Ejemplo 8.4

Descubre los ámbitos de las variables f, i y a en la siguiente expresión: «En cada fiesta del pueblo se cumple que, si todos los invitados a la fiesta están felices, entonces hay al menos una actividad en la fiesta que es emocionante».

Al formalizar la expresión, encontramos que hay tres expresiones que indican el uso de un cuantificador:

1. «*En cada fiesta...*»: Se induce la idea de un cuantificador existencial, que es el cuantificador externo.
2. «*Todos los invitados...*»: Se alude el uso de un cuantificador universal, que es el primer cuantificador interno.

3. «*Hay al menos una actividad...*»: Se refiere a un cuantificador existencial, que es el segundo cuantificador interno.

El cuantificador externo puede ser formalizado como sigue, donde también se ha marcado el ámbito de la variable f:

$$\exists f \in \texttt{Fiestas} : \boxed{P(f)}.$$

Aquí podemos suponer que cada elemento de `Fiesta` es un registro que tiene la información tanto de los invitados como de las actividades, entre otros datos que pudiera tener, por ejemplo la siguiente tabla, donde se muestra apenas la estructura básica de la información de una fiesta:

	Fiestas		
Índice	Invitados	Actividades	...
0	[ana, pedro, susana]	[comida,juego,caminata,canto]	...
⋮	⋮	⋮	⋮

El predicado P se construye con la formalización de «*... si todos los invitados a la fiesta están felices, entonces hay al menos una actividad en la fiesta que es emocionante*», que como notamos, es una expresión condicional con el cuantificador universal como antecedente y un cuantificador existencial como consecuente:

$$P \leftarrow \lambda i, a \bullet \rightarrow (Q(i,f), R(a,f)),$$

donde Q y R son los cuantificadores universal y existencial respectivamente.

El antecedente de la implicación, que es el cuantificador universal Q, es la parte «*...todos los invitados a la fiesta están felices...*», que puede ser formalizado como:

$$\forall i \in \texttt{Invs} : \boxed{estaFeliz(i,f)}$$

Que comunica la idea de que todos los invitados cumplen que, el invitado i está feliz en la fiesta f. También se ha marcado el ámbito de este cuantificador.

Por su parte, el cuantificador existencial R es la parte del texto «*... hay al menos una actividad en la fiesta que es emocionante*», que puede ser formalizado como:

$$\exists a \in \texttt{Acts} : \boxed{esEmocionante(a,f)}$$

Finalmente, la formalización de la expresión original, junto con los ámbitos de cada cuantificador es:

$$\exists f \in \texttt{Fiestas} : \boxed{\rightarrow\Big(\forall i \in \texttt{Invs} : \boxed{estaFeliz(i,f)} \ , \ \exists a \in \texttt{Acts} : \boxed{esEmocionante(a,f)}\Big)}.$$

Solamente hay que anotar que `Invs` se refiere al campo 0 de un registro de `Fiestas`, y `Acts` es el campo 1 del mismo registro. Ya sin los marcos que señalan el ámbito de las variables:

$$\exists f \in \texttt{Fiestas} : \rightarrow(\forall i \in \texttt{Invs} : estaFeliz(i,f) \ , \ \exists a \in \texttt{Acts} : esEmocionante(a,f)).$$

En el caso del ejemplo 8.4 de la página 171, se ha creado una cuantificador que tiene anidados dos cuantificadores, pero los cuantificadores anidados están al mismo nivel, por lo que la variable i del cuantificador universal interno, no debe ser utilizada en el entorno de la variable a del cuantificador existencial interno. Sin embargo, la variable f definida en el cuantificador externo, sí es válida dentro de ambos entornos internos.

8.2.2 Evaluación de cuantificadores anidados

La manera de evaluar un cuantificador es instanciar las variables definidas en el cuantificador y evaluar el predicado con cada una. Cuando hay cuantificadores anidados, la manera de proceder es instanciar la variable del cuantificador externo y evaluar el cuantificador interno del mismo modo.

Este procedimiento establece [en el peor caso], que se deben instanciar todos los valores del cuantificador externo, y por cada instancia evaluar el cuantificador interno instanciando las variables definidas en ese ámbito.

Veamos cómo se realiza la evaluación con el siguiente ejemplo que tiene solamente dos cuantificadores anidados:

$$\forall x \in \langle 1,2,3 \rangle : \forall y \in \langle 7,8,9,10 \rangle : 2x < y$$

Para la evaluación, se sustituye cada expresión por su significado, de izquierda a derecha y se van resolviendo a medida que se tengan todos los operandos en su expresión más primitiva y se tengan únicamente operaciones no postergadas [ver página 132]:

$$\begin{aligned}
&\forall x \in \langle 1,2,3 \rangle : \forall y \in \langle 7,8,9,10 \rangle : 2x < y \\
&= \bigwedge (\mathbb{m}(\lambda x \bullet \forall y \in \langle 7,8,9,10 \rangle : 2x < y, \langle 1,2,3 \rangle)) \\
&= \bigwedge \langle \forall y \in \langle 7,8,9,10 \rangle : 2(1) < y, \forall y \in \langle 7,8,9,10 \rangle : 2(2) < y, \forall y \in \langle 7,8,9,10 \rangle : 2(3) < y \rangle \\
&= [\forall y \in \langle 7,8,9,10 \rangle : 2 < y] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 4 < y] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 6 < y] \\
&= \bigwedge (\mathbb{m}(\lambda y \bullet 2 < y, \langle 7,8,9,10 \rangle)) \wedge [\forall y \in \langle 7,8,9,10 \rangle : 4 < y] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 6 < y] \\
&= \bigwedge (\langle 2 < 7, 2 < 8, 2 < 9, 2 < 10 \rangle) \wedge [\forall y \in \langle 7,8,9,10 \rangle : 4 < y] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 6 < y] \\
&= [2 < 7 \wedge 2 < 8 \wedge 2 < 9 \wedge 2 < 10] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 4 < y] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 6 < y] \\
&= [\mathbf{V} \wedge \mathbf{V} \wedge \mathbf{V} \wedge \mathbf{V}] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 4 < y] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 6 < y] \\
&= \mathbf{V} \wedge [\forall y \in \langle 7,8,9,10 \rangle : 4 < y] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 6 < y] \\
&= \mathbf{V} \wedge \left[\bigwedge \mathbb{m}(\lambda y \bullet 4 < y, \langle 7,8,9,10 \rangle)\right] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 6 < y] \\
&= \mathbf{V} \wedge [4 < 7 \wedge 4 < 8 \wedge 4 < 9 \wedge 4 < 10] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 6 < y] \\
&= \mathbf{V} \wedge [\mathbf{V} \wedge \mathbf{V} \wedge \mathbf{V} \wedge \mathbf{V}] \wedge [\forall y \in \langle 7,8,9,10 \rangle : 6 < y] \\
&= \mathbf{V} \wedge \mathbf{V} \wedge \left[\bigwedge \mathbb{m}(\lambda y \bullet 6 < y, \langle 7,8,9,10 \rangle)\right] \\
&= \mathbf{V} \wedge \mathbf{V} \wedge [6 < 7 \wedge 6 < 8 \wedge 6 < 9 \wedge 6 < 10] \\
&= \mathbf{V} \wedge \mathbf{V} \wedge [\mathbf{V} \wedge \mathbf{V} \wedge \mathbf{V} \wedge \mathbf{V}] \\
&= \mathbf{V} \wedge \mathbf{V} \wedge \mathbf{V} \\
&\mapsto \mathbf{V} \quad \blacksquare
\end{aligned}$$

Observa que para cada valor de la variable de un cuantificador más externo se resuelven todos los valores de la variable de un cuantificador más interno, observa el siguiente desarrollo:

Para $x \leftarrow 1$:
Para $y \leftarrow 7$: $2(1) < 7 = 2 < 7 \mapsto \mathbf{V}$
Para $y \leftarrow 8$: $2(1) < 8 = 2 < 8 \mapsto \mathbf{V}$
Para $y \leftarrow 9$: $2(1) < 9 = 2 < 9 \mapsto \mathbf{V}$
Para $y \leftarrow 10$: $2(1) < 10 = 2 < 10 \mapsto \mathbf{V}$
Para $x \leftarrow 2$:
Para $y \leftarrow 7$: $2(2) < 7 = 4 < 7 \mapsto \mathbf{V}$
Para $y \leftarrow 8$: $2(2) < 8 = 4 < 8 \mapsto \mathbf{V}$
Para $y \leftarrow 9$: $2(2) < 9 = 4 < 9 \mapsto \mathbf{V}$
Para $y \leftarrow 10$: $2(2) < 10 = 4 < 10 \mapsto \mathbf{V}$
Para $x \leftarrow 3$:
Para $y \leftarrow 7$: $2(3) < 7 = 6 < 7 \mapsto \mathbf{V}$
Para $y \leftarrow 8$: $2(3) < 8 = 6 < 8 \mapsto \mathbf{V}$
Para $y \leftarrow 9$: $2(3) < 9 = 6 < 9 \mapsto \mathbf{V}$
Para $y \leftarrow 10$: $2(3) < 10 = 6 < 10 \mapsto \mathbf{V}$

Como todos los casos resultaron **V**, se concluye que:

$$\forall x \in \langle 1,2,3 \rangle : \forall y \in \langle 7,8,9,10 \rangle : 2x < y \mapsto \mathbf{V}.$$

Cuando hay cuantificadores universales o existenciales, es posible que *efectivamente* [computacionalmente] no se realicen todas las pruebas, pues basta encontrar un caso en que no se cumpla el predicado para que termine un cuantificador universal, o bien encontrar un caso en que sí se cumpla el predicado para que termine el cuantificador existencial.

Ejemplo 8.5
Considerando que

Fiestas

Índice	[0] Invitados	[1] Actividades	[2] Fecha	[3] Costo
F_0	[ana, pedro, susana]	[comida,juego]	23/dic/2024	12000
F_1	[tina, mona]	[cena, lectura, paseo]	12/jun/2024	7000
F_2	[gina, fer, fiona]	[pelicula, cena]	06/feb/2024	8500

Y que
$\texttt{Invs} \leftarrow \lambda f \cdot f_0$ ▷ *f_0 es una lista de nombres.*
$\texttt{Acts} \leftarrow \lambda f \cdot f_1$ ▷ *f_1 es una lista de actividades.*

$esFlz \leftarrow \lambda i \cdot rnd(0,1) < 0.5$ ▷ ***V** o **F** en forma aleatoria.*
$esEmte \leftarrow \lambda a \cdot rnd(0,1) < 0.5$ ▷ ***V** o **F** en forma aleatoria.*

Evalúa la siguiente expresión: «Hay una fiesta en la que, si todos los invitados a esa fiesta están felices, entonces es que hay una actividad emocionante en esa fiesta». Al formalizar la expresión se llega a la siguiente proposición:

$$\exists f \in \texttt{Fiestas} : \rightarrow (\forall i \in Invs(f) : esFlz(i)\ ,\ \exists a \in Acts(f) : esEmte(a))\ .$$

Haciendo las respectivas sustituciones:

$$\exists f \in \texttt{Fiesta} : \rightarrow(\forall i \in Invs(f) : esFlz(i)\ ,\ \exists a \in Acts(f) : esEmte(a))$$

$$= \bigvee (\text{𝕄}(\lambda f \bullet \rightarrow(\forall i \in Invs(f) : esFlz(i)\ ,\ \exists a \in Acts(f) : esEmte(a)), \langle \texttt{F}_0, \texttt{F}_1, \texttt{F}_2 \rangle))$$

$$\begin{array}{l} = [\rightarrow(\forall i \in Invs(\texttt{F}_0) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_0) : esEmte(a))] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_1) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_1) : esEmte(a))] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))] \end{array}$$

$$\begin{array}{l} = [\rightarrow(\forall i \in \langle \text{ana,pedro, susana} \rangle : esFlz(i)\ ,\ \exists a \in \langle \text{comida, juego} \rangle : esEmte(a))] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_1) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_1) : esEmte(a))] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))] \end{array}$$

$$\begin{array}{l} = [\rightarrow([esFlz(\text{ana}) \land esFlz(\text{pedro}) \land esFlz(\text{susana})],\ [esEmte(\text{comida}) \lor esEmte(\text{juego})])] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_1) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_1) : esEmte(a))] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))] \end{array}$$

$$\begin{array}{l} = [\rightarrow([\mathbf{V} \land \mathbf{V} \land \mathbf{F}],\ [\mathbf{F} \lor \mathbf{V}])] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_1) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_1) : esEmte(a))] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))] \end{array}$$

$$\begin{array}{l} = [\rightarrow(\mathbf{F},\ \mathbf{V})] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_1) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_1) : esEmte(a))] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))] \end{array}$$

$$\begin{array}{l} = \mathbf{V} \lor [\rightarrow(\forall i \in Invs(\texttt{F}_1) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_1) : esEmte(a))] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))] \end{array}$$

$$\begin{array}{l} = \mathbf{V} \lor [\rightarrow(\forall i \in \langle \text{tina, mona} \rangle : esFlz(i)\ ,\ \exists a \in \langle \text{cena, lectura, paseo} \rangle : esEmte(a))] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))] \end{array}$$

$$\begin{array}{l} = \mathbf{V} \lor [\rightarrow(esFlz(\text{tina}) \land esFlz(\text{mona})\ ,\ esEmte(\text{cena}) \lor esEmte(\text{lectura}) \lor esEmte(\text{paseo}))] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))] \end{array}$$

$$\begin{array}{l} = \mathbf{V} \lor [\rightarrow(\mathbf{V} \land \mathbf{V}\ ,\ \mathbf{F} \lor \mathbf{F} \lor \mathbf{V})] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))] \end{array}$$

$$\begin{array}{l} = \mathbf{V} \lor [\rightarrow(\mathbf{V}\ ,\ \mathbf{V})] \lor \\ [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))] \end{array}$$

$$= \mathbf{V} \lor \mathbf{V} \lor [\rightarrow(\forall i \in Invs(\texttt{F}_2) : esFlz(i)\ ,\ \exists a \in Acts(\texttt{F}_2) : esEmte(a))]$$

$$= \mathbf{V} \lor \mathbf{V} \lor [\rightarrow(\forall i \in \langle \text{gina,fer,fiona} \rangle : esFlz(i)\ ,\ \exists a \in \langle \text{pelicula,cena} \rangle : esEmte(a))]$$

$$= \mathbf{V} \lor \mathbf{V} \lor [\rightarrow(esFlz(\text{gina}) \land esFlz(\text{fer}) \land esFlz(\text{fiona})\ ,\ esEmte(\text{pelicula}) \lor esEmte(\text{cena}))]$$

$$= \mathbf{V} \lor \mathbf{V} \lor [\rightarrow(\mathbf{F} \land \mathbf{V} \land \mathbf{V}\ ,\ \mathbf{F} \lor \mathbf{F})]$$

$$= \mathbf{V} \lor \mathbf{V} \lor [\rightarrow(\mathbf{F}\ ,\ \mathbf{F})]$$

$$= \mathbf{V} \lor \mathbf{V} \lor \mathbf{V}$$

$$\mapsto \mathbf{V} \quad \blacksquare$$

Nota: Los resultados de las aplicaciones para $esFlz(f)$ y $esEmte(a)$ han sido obtenidos aleatoriamente, de modo que es posible que se generen resultados diferentes en ejecuciones diferentes.

Código 8.1: Traducción a `Python` del ejemplo 8.5

```
Fiestas = [['ana', 'pedro', 'susana'], ['comida','juego'], "23/dic/2024", 12000],
          [['tina', 'mona'], ['cena', 'lectura', 'paseo'], "12/jun/2024", 7000],
          [['gina', 'fer', 'fiona'], ['pelicula', 'cena'], "06/feb/2024", 8500]]

Invs = lambda f:f[0]
Acts = lambda f:f[1]

import random

esFlz = lambda i: random.uniform(0,1) < 0.5
esEmte = lambda a:random.uniform(0,1) < 0.5
```

```
>>> print(existeUn(lambda f:
    impl(paraTodo(lambda i: esFlz(i),Invs(f)),
         existeUn(lambda a: esEmte(a), Acts(f))),Fiestas))
True
>>>
```

`import random` permite acceder a funciones relacionadas con la generación de números aleatorios y otras operaciones aleatorias. Una vez que importas el módulo `random`, puedes utilizar varias funciones y métodos que están dentro de él para realizar diferentes tareas, entre estas:

- Generar números aleatorios con distribución uniforme en un rango dado, usando `random.uniform(0,1)` como en el ejemplo.
- Revolver los elementos de una lista, usando `shuffle()`.
- Seleccionar de manera aleatoria elementos de una lista, usando `choice()`.

La negación en los cuantificadores anidados sigue las mismas reglas que la negación de cuantificadores simples. Recuerda que un cuantificador es un predicado, por lo que puede ser colocado en el lugar destinado para cualquier predicado de otra proposición, incluyendo otro cuantificador. Basta recordar la sintaxis de un cuantificador, por ejemplo el cuantificador universal:

$$\forall\langle \texttt{variables} \rangle : \langle \texttt{lista} \rangle : \boxed{\langle \texttt{Predicado} \rangle}$$

En el lugar donde va un predicado se puede colocar otro cuantificador, que a su vez requiere de otro predicado, que puede ser un nuevo cuantificador, que requiere su propio predicado; este anidamiento de cuantificadores es posible hasta que sea necesario.

Así, para lograr la negación de un cuantificador que tenga como predicado un cuantificador se realiza lo siguiente:

1. Eliminar el símbolo de negación al inicio del cuantificador más externo.
2. Sustituir el símbolo de $\exists$ por $\forall$ o bien $\forall$ por $\exists$ en el predicado más externo, según sea el caso.
3. Colocar un símbolo de negación en el inmediato cuantificador interno, luego tratar de simplificar la negación del predicado interno.

El proceso de eliminación de negaciones se repite hasta que los cuantificadores no tengan símbolos de negación.

Ejemplo 8.6

Escribe una expresión equivalente a $\neg\forall x \in \{3,5,7,10\} : \exists y \in \{3,5,7,10\} : \frac{x}{y} = 1$ que no tenga negaciones al inicio de ningún cuantificador.

Siguiendo el método de *eliminación de negaciones* descrito anteriormente:

(1) Eliminar el símbolo al inicio:

$$\boxed{\neg}\forall x \in \{3,5,7,10\} : \exists y \in \{3,5,7,10\} : \frac{x}{y} = 1.$$
$$\forall x \in \{3,5,7,10\} : \exists y \in \{3,5,7,10\} : \frac{x}{y} = 1.$$

(2) Cambiar el símbolo $\forall$ por el símbolo $\exists$.

$$\boxed{\forall} x \in \{3,5,7,10\} : \exists y \in \{3,5,7,10\} : \frac{x}{y} = 1.$$
$$\exists x \in \{3,5,7,10\} : \exists y \in \{3,5,7,10\} : \frac{x}{y} = 1.$$

(3) Negar el predicado del cuantificador y hacer las reducciones necesarias.

$$\exists x \in \{3,5,7,10\} : \boxed{\exists y \in \{3,5,7,10\} : \frac{x}{y} = 1}.$$
$$\exists x \in \{3,5,7,10\} : \neg\exists y \in \{3,5,7,10\} : \frac{x}{y} = 1.$$

Ahora tenemos la negación en el cuantificador interno, por lo que repetimos el procedimiento con este cuantificador:

(1) Eliminar el símbolo al inicio:

$$\exists x \in \{3,5,7,10\} : \boxed{\neg}\exists y \in \{3,5,7,10\} : \frac{x}{y} = 1.$$
$$\exists x \in \{3,5,7,10\} : \exists y \in \{3,5,7,10\} : \frac{x}{y} = 1.$$

(2) Cambiar el símbolo $\exists$ por el símbolo $\forall$.

$$\exists x \in \{3,5,7,10\} : \boxed{\exists} y \in \{3,5,7,10\} : \frac{x}{y} = 1.$$
$$\exists x \in \{3,5,7,10\} : \forall y \in \{3,5,7,10\} : \frac{x}{y} = 1.$$

(3) Negar el predicado del cuantificador y hacer las reducciones necesarias.

$$\exists x \in \{3,5,7,10\} : \forall y \in \{3,5,7,10\} : \boxed{\frac{x}{y} = 1}.$$
$$\exists x \in \{3,5,7,10\} : \forall y \in \{3,5,7,10\} : \neg\left(\frac{x}{y} = 1\right).$$
$$\exists x \in \{3,5,7,10\} : \forall y \in \{3,5,7,10\} : \frac{x}{y} \neq 1.$$

Así $\neg\forall x \in \{3,5,7,10\} : \exists y \in \{3,5,7,10\} : \frac{x}{y} = 1 \equiv \exists x \in \{3,5,7,10\} : \forall y \in \{3,5,7,10\} : \frac{x}{y} \neq 1$ ■

El ejemplo 8.5 [página 174] ilustra un caso interesante, porque el cuantificador externo [$\exists f \in$ `Fiestas`…] contiene una implicación compuesta por dos cuantificadores anidados al mismo nivel. La negación de esta expresión se escribe:

$$\neg\exists f \in \texttt{Fiestas} : \rightarrow(A, B) \tag{8.1}$$

Donde: A es $\forall i \in Invs(f) : esFlz(i)$
B es $\exists a \in Acts(f) : esEmte(a)$

Al escribir la expresión equivalente sin negaciones al principio se tiene:

$$\forall f \in \texttt{Fiestas} : \neg(\rightarrow(A, B)).$$

Observa que el predicado es la negación de una implicación, recuerda las expresiones lógicamente equivalentes usando la tabla de verdad [ver la página 110], y que la negación de una implicación es precisamente la q-Inhibición [ver la tabla 5.3, página 110]; desafortunadamente esta función no es estándar en los lenguajes de programación, por lo que usualmente se logra con una expresión lógica equivalente.

La expresión lógica $\neg(\rightarrow(A,B))$ se puede reducir utilizando las leyes de la lógica proposicional. La implicación lógica $\rightarrow(A,B)$ es equivalente a $\vee(\neg(A),B)$, como se observa en la siguiente tabla de verdad.

				∇	∇
A	B	$\rightarrow(A,B)$	$\vee(\neg(A),B)$	$\neg(\rightarrow(A,B))$	$\neg(\vee(\neg(A),B))$
V	**V**	**V**	**V**	**F**	**F**
V	**F**	**F**	**F**	**V**	**V**
F	**V**	**V**	**V**	**F**	**F**
F	**F**	**V**	**V**	**F**	**F**

Por lo tanto, $\neg(\rightarrow(A,B)) \equiv \neg(\vee(\neg(A),B))$. Usando la ley de De Morgan en el caso disyuntivo [ver la página 77] en la parte derecha de la equivalencia, tenemos:

$$\neg(\rightarrow(A,B)) \equiv \wedge(\neg(\neg(A)),\neg(B))$$

Usando la regla de la doble negación [página 62] en $\neg(\neg(A))$:

$$\neg(\rightarrow(A,B)) \equiv \wedge(A,\neg(B))$$

Regresando ahora a la expresión inicial (8.1):

$$\neg\exists f \in \texttt{Fiestas} : \rightarrow(A,B) \equiv \forall f \in \texttt{Fiestas} : (A \wedge \neg B). \tag{8.2}$$

Pero:

A es $\forall i \in Invs(f) : esFlz(i)$

B es $\exists a \in Acts(f) : esEmte(a)$

Por lo que

$$\forall f \in \texttt{Fiestas} : (\forall i \in Invs(f) : esFlz(i) \wedge \neg\exists a \in Acts(f) : esEmte(a))$$

Encontramos ahora la negación de un cuantificador existencial, y haciendo las operaciones correspondientes, llegamos a la expresión final:

$$\forall f \in \texttt{Fiestas} : (\forall i \in Invs(f) : esFlz(i) \wedge \forall a \in Acts(f) : \neg(esEmte(a))) \tag{8.3}$$

que ya no tiene negaciones al inicio del cuantificador.

Ejercicios

1. Expresa la negación de la siguiente proposición cuantificada:
   a) *Ejemplo.* «Todos los estudiantes aprobaron el examen».
   Negación: «No todos los estudiantes aprobaron el examen», o bien, «Algún estudiante no aprobó el examen».

   b) «Todos los estudiantes asistieron a la conferencia».
   c) «Cada pájaro puede volar».
   d) «Todos los días llueve al menos una vez».
   e) «No hay ni una persona que no le guste la navidad».
2. Utiliza cuantificadores anidados para expresar mediante símbolos la siguiente situación:
   a) *Ejemplo.* «Hay al menos un perro que disfruta de la compañía de todos los gatos».
   $$\exists p \in \texttt{Perros} : \forall g \in \texttt{Gatos} : disfruta(p, g)$$
   b) «Cada persona tiene al menos un amigo que conoce a todas sus mascotas».
   c) «En cada habitación, hay al menos un mueble que es más grande que todos los demás».
   d) «Cada empresa tiene al menos un empleado que ha trabajado en todos sus departamentos».
3. Formula una proposición que contenga un cuantificador dentro del predicado de otro cuantificador.
   a) *Ejemplo.* «Hay libros que si tienen al menos tres capítulos, entonces todos los capítulos de esos libros tienen ejercicios».

   $$\exists \ell \in \texttt{Libros} : \rightarrow (numCaps(\ell) \geq 3, \forall c \in caps(\ell) : tieneEjercicios(c))$$

   Esto afirma que hay libros $[\ell]$ con 3 capítulos o más $[c]$ que tienen ejercicios en todos sus capítulos.
   b) «Cada equipo tiene al menos dos jugadores que han ganado más de tres partidos consecutivos».
   c) «Cada clase tiene al menos un estudiante que ha participado en todas las actividades académicas».
   d) «Cada tienda tiene al menos un cliente que ha comprado en todos los departamentos».
   e) «En todos los departamentos de una tienda, si entra un cliente al departamento, entonces compra algo de ese departamento de la tienda».
4. Niega la siguiente proposición con cuantificadores anidados.
   a) *Ejemplo.* «Todos los días, a cada hora, alguien lee un libro».
   Negación: «Hay al menos un día y una hora en la que nadie lee un libro». La negación de una afirmación universal como «Todos los días, a cada hora» se convierte en una afirmación existencial negada. En este caso, se afirma que existe al menos un día y una hora en los que nadie lee, lo que contradice la proposición original.

   b) «En todas las reuniones, conoce al menos a alguien que ha viajado a todos los continentes».

c) «En cada película, para cualquier escena, hay al menos un actor que aparece en todas las tomas».

d) «En cada festival, en algún momento, todos los asistentes disfrutan de al menos una presentación musical».

5. Desarrolla la evaluación de la expresión sustituyendo los valores de los dominios. Si es necesario, primero simplifica la expresión.

a) *Ejemplo.* $\neg\forall x \in \langle 4,6,8,12\rangle : \neg\exists y \in \langle 2,3,4\rangle : (x-2y)^2 > (2x-y)^2$.

$$
\begin{aligned}
&\neg\forall x \in \langle 4,6,8,12\rangle : \neg\exists y \in \langle 2,3,4\rangle : (x-2y)^2 > (2x-y)^2\\
&= \exists x \in \langle 4,6,8,12\rangle : \exists y \in \langle 2,3,4\rangle : (x-2y)^2 > (2x-y)^2\\
&= \bigvee \mathbb{m}\Big(\lambda x \bullet \exists y \in \langle 2,3,4\rangle : (x-2y)^2 > (2x-y)^2, \langle 4,6,8,12\rangle\Big)\\
&= \langle \exists y \in \langle 2,3,4\rangle : (4-2y)^2 > (8-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (6-2y)^2 > (12-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (8-2y)^2 > (16-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (12-2y)^2 > (24-y)^2\rangle\\
&= \langle \bigvee \mathbb{m}\Big(\lambda y \bullet (4-2y)^2 > (8-y)^2, \langle 2,3,4\rangle\Big)\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (6-2y)^2 > (12-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (8-2y)^2 > (16-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (12-2y)^2 > (24-y)^2\rangle\\
&= \langle ((4-4)^2 > (8-2)^2) \vee ((4-6)^2 > (8-3)^2) \vee ((4-8)^2 > (8-4)^2)\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (6-2y)^2 > (12-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (8-2y)^2 > (16-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (12-2y)^2 > (24-y)^2\rangle\\
&= \langle \mathbf{F} \vee \mathbf{F} \vee \mathbf{F}\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (6-2y)^2 > (12-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (8-2y)^2 > (16-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (12-2y)^2 > (24-y)^2\rangle\\
&= \mathbf{F} \vee \langle \bigvee \mathbb{m}\Big(\lambda y \bullet (6-2y)^2 > (12-y)^2, \langle 2,3,4\rangle\Big)\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (8-2y)^2 > (16-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (12-2y)^2 > (24-y)^2\rangle\\
&= \mathbf{F} \vee \langle (((-2)^2 > (10)^2) \vee ((0)^2 > (9)^2) \vee ((-2)^2 > (8)^2)\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (8-2y)^2 > (16-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (12-2y)^2 > (24-y)^2\rangle\\
&= \mathbf{F} \vee \langle \mathbf{F} \vee \mathbf{F} \vee \mathbf{F}\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (8-2y)^2 > (16-y)^2\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (12-2y)^2 > (24-y)^2\rangle\\
&= \mathbf{F} \vee \mathbf{F} \vee \langle \bigvee \mathbb{m}\Big(\lambda y \bullet (8-2y)^2 > (16-y)^2, \langle 2,3,4\rangle\Big)\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (12-2y)^2 > (24-y)^2\rangle\\
&= \mathbf{F} \vee \mathbf{F} \vee \langle \mathbf{F} \vee \mathbf{F} \vee \mathbf{F}\rangle \vee\\
&\quad \langle \exists y \in \langle 2,3,4\rangle : (12-2y)^2 > (24-y)^2\rangle\\
&= \mathbf{F} \vee \mathbf{F} \vee \mathbf{F} \vee \langle \exists y \in \langle 2,3,4\rangle : (12-2y)^2 > (24-y)^2\rangle\\
&= \mathbf{F} \vee \mathbf{F} \vee \mathbf{F} \vee \langle \bigvee \mathbb{m}\Big(\lambda y \bullet (12-2y)^2 > (24-y)^2, \langle 2,3,4\rangle\Big)\rangle\\
&= \mathbf{F} \vee \mathbf{F} \vee \mathbf{F} \vee \langle \mathbf{F} \vee \mathbf{F} \vee \mathbf{F}\rangle\\
&= \mathbf{F} \vee \mathbf{F} \vee \mathbf{F} \vee \mathbf{F}\\
&\mapsto \mathbf{F} \quad \blacksquare
\end{aligned}
$$

b) $\neg\forall d \in \langle 3,6,9\rangle : [\exists h \in \langle 6,8,10\rangle : h > d]$

c) $\exists n \in \langle 5,2,7 \rangle : [\neg \forall y \in \langle 6,8,2 \rangle : |x-y| > (2x-y)]$
d) $\neg \forall \alpha \in \langle \frac{1}{2}\pi, \pi, \frac{3}{2}\pi, 2\pi \rangle : [\neg \exists y \in \langle 0.0, 0.5, 1.0 \rangle : sen(\alpha) \leq y]$

6. Escribe correctamente la negación de las siguientes expresiones que involucran cuantificadores:
   a) *Ejemplo.* $\forall x \in \langle 0,2,\ldots,100 \rangle : [\neg\neg \exists y \in \langle 2,4,6,10 \rangle : (x+y)^2 \neq (2x-y)^2]$.

$$
\begin{aligned}
&= \neg\Big(\forall x \in \langle 0,2,\ldots,100 \rangle : [\neg\neg \exists y \in \langle 2,4,6,10 \rangle : (x+y)^2 \neq (2x-y)^2\Big)] \\
&= \neg\Big(\forall x \in \langle 0,2,\ldots,100 \rangle : [\exists y \in \langle 2,4,6,10 \rangle : (x+y)^2 \neq (2x-y)^2\Big)] \\
&= \exists x \in \langle 0,2,\ldots,100 \rangle : [\neg\Big(\exists y \in \langle 2,4,6,10 \rangle : (x+y)^2 \neq (2x-y)^2\Big)] \\
&= \exists x \in \langle 0,2,\ldots,100 \rangle : [\forall y \in \langle 2,4,6,10 \rangle : \neg\Big((x+y)^2 \neq (2x-y)^2\Big)] \\
&= \exists x \in \langle 0,2,\ldots,100 \rangle : [\forall y \in \langle 2,4,6,10 \rangle : (x+y)^2 = (2x-y)^2]
\end{aligned}
$$

   b) $\neg\neg \exists x \in \mathbb{R} : [\exists y \in \mathbb{R} : x * y < x^y]$.
   c) $\forall x \in \langle$ `'a','o'` $\rangle : [\forall y \in \langle$ `'sala','solo' ` $\rangle : \neg iniciaCon(y,x)]$
   d) $\exists f \in \langle 4,8,2 \rangle : [\rightarrow\Big(2 < f^2, \forall g \in \langle 1,2,3 \rangle : 2f < 3g\Big)]$

7. Escribe una expresión en `Python` que sea equivalente a la siguiente expresión en lenguaje natural:
   a) *Ejemplo.* «Hay un paciente del hospital que no tiene ni una enfermedad tropical».

```
existeUn(lambda p: neg(paraTodo(lambda e: neg(tiene(p,e)),
                                enfermedadTrop)),
         Hospital)
```

   b) «No hay persona alguna en el mundo que no tenga ganas de salir adelante». «Para cada estudiante en la universidad, existe al menos una asignatura en la que todos los estudiantes obtienen una calificación superior a 9».
   c) «Existe un país donde para todos sus habitantes, existe al menos un deporte que practican con regularidad y en el que son campeones nacionales».
   d) «Para cada libro en la biblioteca, existe al menos un capítulo donde se menciona algún autor reconocido en el campo y todos los conceptos son explicados de manera detallada».

8. Considera la siguiente definición:

> **Definición 8.2.1 -- Contingencia.** Un predicado es una *contingencia*, si no es una tautología [ver ejercicio 6 en la página 165] y tampoco es una contradicción [ver ejercicio 7 en la página 166].

Escribe una función que se llame `esContingencia` que debe ser utilizado para determinar si un predicado `P` es una contingencia. La función debe recibir un predicado `P`, dado por su identificador o bien como una expresión lambda. La función debe devolver `True` si el predicado `P` es una contingencia y `False` si caso contrario.

```
>>> esContingencia(y)
True
>>> esContingencia(lambda p, q: True)
False
>>> esContingencia(lambda p, q: False)
False
>>>
```

9. Para los siguientes ejercicios considera las siguientes tablas de datos.

Países

Índice	[0] Nombre	[1] Continente	[2] Población
p_0	España	Europa	47.3 millones
p_1	Francia	Europa	67.1 millones
p_2	Alemania	Europa	83.2 millones
p_3	Canadá	América	60.4 millones
p_4	EUA	América	331 millones
p_5	Cuba	América	38.3 millones
p_6	Brasil	América	213 millones
p_7	México	América	126.6 millones
p_8	Argentina	América	45.4 millones
p_9	Paraguay	América	1,397 millones
p_{10}	Chile	América	1,366 millones
p_{11}	Colombia	América	126.5 millones
p_{12}	Corea del Sur	Asia	51.8 millones
p_{13}	Indonesia	Asia	276.4 millones

Viajeros

Índice	[0] Nombre	[1] Sexo	[2] Nac	[3] Pasaporte vigente	[4] Países conocidos
v_0	Vero	F	Mex	**V**	⟨Francia, España, Cuba⟩
v_1	Fer	M	Uru	**V**	⟨Paraguay, Argentina, Chile⟩
v_2	Polo	M	Mex	**V**	⟨EUA, Colombia, Canadá, Francia⟩
v_3	Fred	M	Cub	**V**	⟨México, Brasil, España, Francia⟩
v_4	Rita	F	Mex	**F**	⟨España, Alemania, Cuba⟩

Escribe una instrucción en Python que resuelva el requerimiento planteado.

a) *Ejemplo.* «Todos los viajeros conocen al menos un país de Europa».

```
paisEuropeo = lambda p: p[1] == "Europa"
paisesConocidos = lambda v: v[4]
paraTodo(lambda v: existeUn(lambda p: paisEuropeo(p),
                            paisesConocidos(v)),
         Viajeros)
```

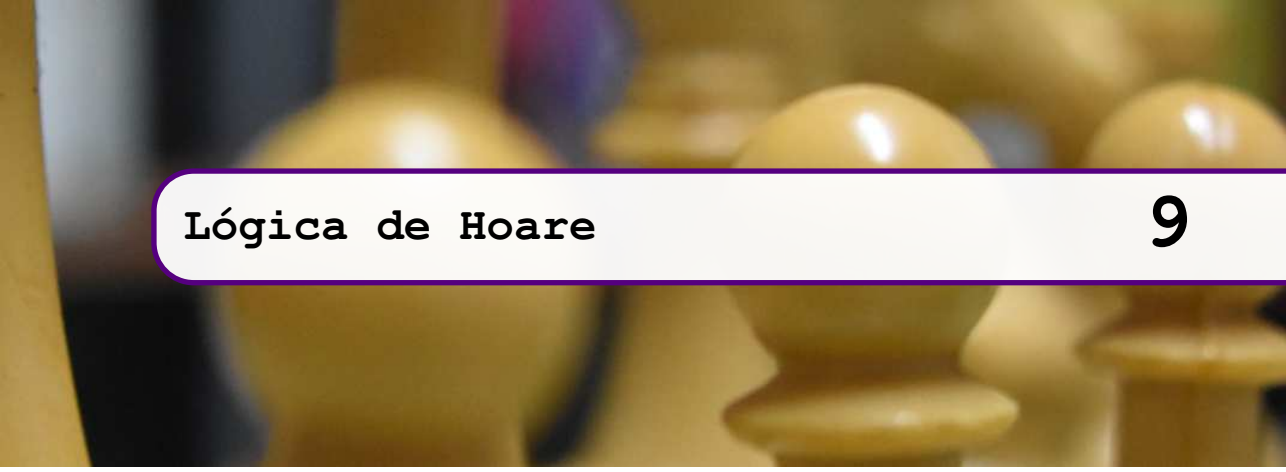
b) «Hay al menos un país europeo que es conocido por todos los viajeros».

c) «Hay viajeros que, si son mexicanos, entonces conocen algún país europeo».

d) «Ningún viajero conoce todos los países».

III

Lógica aplicada



Lógica de Hoare 9

9.1 Introducción a la lógica de Hoare

La lógica de Hoare ha sido fundamental en el desarrollo de métodos formales para la verificación de programas. Al proporcionar una base matemática y estructurada para el razonamiento sobre la corrección del software, ha permitido mejorar la confiabilidad y eficiencia en el desarrollo de software complejo y crítico.

9.1.1 Panorama histórico

La lógica de Hoare, creada por el matemático británico Sir Charles Antony Richard Hoare a finales de la década de 1960 [Hoa69], quien cambió la forma en que se abordaba la manera de garantizar un software libre de errores, ya que en ese entonces la computación se enfrentaba al desafío de producir software cada vez más complejo y confiable.

Recordemos que desde finales de la década de 1960 hasta inicios de la década de 1970, fue una época en donde se sentaron las bases del paradigma imperativo en los lenguajes de programación y han sido más influyentes en el desarrollo de software; en esa época se crearon lenguajes como `Logo` en 1967 [SHK+20, Log15]; `B` en 1969-1970 [Mac99, Tho72], que influyó en el desarrollo del lenguaje `C` ; `Pascal` en 1970 [Wir71]; `C` en 1972 [Rit96, Mac99] y `Smalltalk` en 1972 [Mac99].

Sin embargo, el entusiasmo por desarrollar software, hacía que se ignoraran errores que posteriormente tendrían que ser corregidos, por lo que era necesario establecer criterios formales que garantizaran que el producto de software estuviera libre de errores.

Hoare desarrolló esta lógica basada en parte en el trabajo de Peter Naur [Nau66] y de Floyd [Flo67], como un método formal para razonar sobre las propiedades de un programa que garantizan que cumplen con los requisitos previamente definidos, proporcionando un marco matemático para evaluar la validez y robustez de los algoritmos.

9.1.2 Principio de la lógica de Hoare

La lógica de Hoare se basa en la verificación de la correctitud de una línea de código, luego dos líneas de código, y avanzar así hasta que todo el código sea verificado. Para hacer esto, Hoare consideró, así como lo hizo Floyd, dos momentos importantes: antes de la ejecución del código y después de hacerlo.

Antes de ejecutar el código, se considera el estado del programa, luego, al ejecutar el código, se observa nuevamente el estado actual del programa. Estos tres elementos: el estado anterior, el código y el estado posterior forman la «tripleta de Hoare».

La lógica de Hoare emplea reglas de inferencia para demostrar la validez de las tripletas de Hoare. Estas reglas permiten derivar la postcondición de una secuencia de comandos basada en la precondición y la sentencia. Además, establecen las condiciones bajo las cuales una porción de código cumplirá su especificación.

La lógica de Hoare se utiliza extensamente en la verificación formal de programas. Por ejemplo, en el desarrollo de software crítico para sistemas médicos, aeronáuticos o de seguridad, se emplea para garantizar que los algoritmos funcionen correctamente en cualquier situación. Además, en la educación de Ciencias Computacionales, se utiliza para enseñar a escribir y analizar algoritmos de manera más precisa y lógica.

El uso de la lógica de Hoare proporciona un marco riguroso y formal para evaluar la corrección de los programas, mejorando la confiabilidad y la precisión en el desarrollo de software crítico y proporcionando un enfoque lógico y matemático para analizar algoritmos y estructuras de datos.

9.1.3 Correctitud y verificación de software

Se han mencionado los términos «verificación» y «correctitud»; comprender estos términos es fundamental para el estudio, análisis y práctica de la lógica de Hoare.

Correctitud: Es la propiedad de un programa de computadora que garantiza que se cumple con su especificación. Un programa se considera correcto si hace lo que se supone que debe hacer, es decir, si cumple con las expectativas y las condiciones establecidas en su diseño y especificaciones.

Verificación: Es el proceso mediante el cual se comprueba o se demuestra formalmente la correctitud de un programa. La verificación implica utilizar métodos lógicos, como las tripletas de Hoare, el cómputo de predicados de primer orden y de orden superior, para demostrar matemáticamente que un programa es correcto, satisfaciendo sus precondiciones y postcondiciones establecidas.

En general, para tener garantía de que un programa cumple con las expectativas, se hace una verificación formal. Cuando la verificación concluye, se determina si el programa es correcto o no lo es.

Para hacer la verificación se requiere una nueva operación que se llama «sustitución», además de algunas reglas que permiten el razonamiento correcto en la lógica de Hoare.

9.2 Sustitución

La sustitución [Chu56] es una acción muy importante en la lógica formal y en particular en la lógica de Hoare.

Definición 9.2.1 -- Sustitución. La sustitución es una operación sobre una expresión E y una tupla $\langle a, b \rangle$. La expresión E puede o no contener el término a. La sustitución actúa en la expresión E, reemplazando cada ocurrencia de a por el término b, esto se denota

$$\int(E, \langle a, b \rangle)$$

El resultado es una nueva expresión E' que contiene el término b en el lugar donde término a se encuentra.

Ejemplo 9.1

Obtener la sustitución de x por y en la expresión $x^2 + 4x + 7$.

Para hacer la sustitución haremos $\int\left(x^2 + 4x + 7, \langle x, y \rangle\right)$ con lo que se obtiene:

$$\begin{aligned} & x^2 + 4x + 7 \\ & (x)^2 + 4(x) + 7 \\ & (y)^2 + 4(y) + 7 \\ & y^2 + 4y + 7 \end{aligned}$$

La expresión resultante es $y^2 + 4y + 7$.

9.3 Tripletas de Hoare

El propósito de la lógica de Hoare es proporcionar las bases lógicas para la demostración de las propiedades que tiene un programa de computadoras, particularmente aquellos programas que siguen un paradigma imperativo.

☞ Paradigma imperativo. Es el modelo de programación que produce una secuencia finita de instrucciones [comandos u órdenes] bien definidas, que van cambiando el estado de las variables del programa a medida que se ejecuta cada instrucción en la secuencia. El resultado del programa es obtenido por el valor final de alguna de las variables del programa. Algunos lenguajes de programación que principalmente siguen este paradigma son `FORTRAN`, `Pascal`, `C` o `Python`. Aunque `Python` se considera multiparadigma, en esencia es imperativo.

La lógica de Hoare se basa en la suposición de que la ejecución de un programa de computadora transforma el estado inicial de las variables del sistema, y después de la ejecución del programa, las variables del sistema pueden haber cambiado de valor. Este marco de trabajo se puede sintetizar en forma de una tripleta, conocida como «tripleta de Hoare». Una tripleta de Hoare es de la forma $\{P\}\ C\ \{Q\}$, donde:

Precondición P: Esta es una afirmación lógica que describe el estado que debe existir antes de la ejecución de una porción de código.

Instrucción C: Representa la acción que se ejecuta, como una asignación, una instrucción condicional o un bucle.

Postcondición Q: Es la afirmación lógica que describe el estado que debe existir después de que la sentencia se haya ejecutado, suponiendo que la precondición inicial era verdadera.

9.3.1 El estado de un programa

El «estado» de un programa se refiere a la situación o condición en la que se encuentra el programa en un momento específico durante su ejecución. Generalmente el estado incluye información sobre los valores de las variables en ese punto.

Se puede representar el estado de un programa utilizando una asignación de valores a las variables del programa en un momento dado. Por ejemplo, si tienes un programa con variables x, y, z, el estado del programa en un punto particular podría ser algo como:

$$\{x = 3, y = 5, z = 0\}$$

Esto significa que en ese punto del programa, la variable x tiene el valor 3, la variable y tiene el valor 5 y la variable z tiene el valor 0.

Ejemplo 9.2

Considera el siguiente segmento de programa:

```
: # {y = 2, z = 3}
x ↤ y**2 + z**2
: # {x = 13}
```

Antes de la instrucción, es posible considerar las variable y y z para determinar el estado actual del programa, pero no x. En `Python` es posible crear nuevas variables en cualquier lugar del programa. Después de la instrucción, el valor de x, así como el valor de y y de z ya son conocidos, por lo que es posible crear un estado que incluya la variable x.

9.3.2 Precondiciones y postcondiciones

La lógica de Hoare utiliza precondiciones y postcondiciones para hacer afirmaciones sobre cómo cambia el estado del programa antes y después de la ejecución de un comando. Esto permite construir razonamientos sobre la correctitud del código.

La expresión $\{P\}\ C\ \{Q\}$ se puede interpretar como «Si P es cierto antes de ejecutar el código C, entonces Q será cierto después de ejecutar C», con lo que podemos decir que el programa C es parcialmente correcto [Mey90, p. 315]. Al inicio, si no hay precondiciones que deban ser cumplidas, entonces se puede escribir: $\{\mathbf{V}\}\ C\ \{Q\}$.

```
def suma_cuadrados(num1, num2):
    # precondición ------------- {P}
    suma = num1**2 + num2**2 # <-- C
    # postcondición ------------ {Q}
    return suma
```

Una tripleta $\{\mathbf{F}\}\ C\ \{Q\}$ significa que la precondición es **F**, implica que la condición inicial es una afirmación que siempre se evalúa como falsa, lo que indica que el programa no puede comenzar desde ese estado. Es una afirmación que nunca se cumple, así la instrucción C no se puede ejecutar correctamente o no tiene sentido ejecutarla porque la precondición nunca se cumple. Si la precondición es **F**, la ejecución del comando se considera indefinida o imposible de alcanzar, ya que el programa no puede comenzar desde un estado donde la precondición no se cumple.

En el otro extremo, una tripleta de la forma $\{P\}\ C\ \{\mathbf{F}\}$ representa una afirmación sobre un programa C donde P es la precondición y la postcondición es **F**. Este caso, significa que bajo la precondición P, al ejecutar el comando C, se garantiza que la postcondición siempre será **F**. Esto implica que el programa no puede finalizar en un

estado donde la postcondición sea **V**, sin importar lo que haga el comando C durante la ejecución. Esta no es una situación deseable, porque indica que, independientemente del estado inicial del programa y de lo que haga C, el resultado siempre contradice la postcondición, lo que sugiere un comportamiento no deseado o un problema en el diseño del programa.

Así, la precondición P y la postcondición Q, son proposiciones que deben ser verdaderas antes y después del segmento de código C que es analizado.

Para hacer las verificaciones, la lógica de Hoare ha provisto axiomas y reglas de inferencia, que pueden ser utilizadas en el orden que se requiera por el código a ser verificado.

9.4 Axiomas y reglas en lógica de Hoare

Los axiomas son proposiciones que son tan claros y evidentes que deben ser considerados como verdaderos; las reglas de inferencia establecen cómo se puede razonar sobre la corrección de un programa. Algunas reglas básicas de inferencia y axiomas en la lógica de Hoare se describen en las siguientes secciones.

9.4.1 Axioma del paso

El «paso» se refiere a una instrucción que no hace nada, en `Python` es la instrucción `pass`. La instrucción `pass` no debe modificar el estado del programa, ni antes de la instrucción, ni después, por lo que se tiene el siguiente:

Axioma 9.4.1 -- Axioma del paso. Si el predicado P es cierto antes de ejecutarse la sentencia `pass`, seguirá siendo cierta después de la ejecución.

$$\vdash \{P\} \text{ pass } \{P\}$$

9.4.2 Axioma de asignación

La asignación es la instrucción más representativa de un lenguaje imperativo. En otros documentos relacionados con lógica de Hoare, se emplea el símbolo '$:=$' para denotar la asignación [Hoa69, FdC+04, Pla18], esto es por influencia de `Pascal` [Jen91] y `Oberon` [RW94], que eran lenguajes de programación muy usados a principios de la década de 1990; pero aquí se usará el símbolo de asignación de `Python`, que es '$=$', para indicar en el código fuente que un valor es asignado a una variable; mientras que el símbolo '$\equiv$' será utilizado para denotar que dos proposiciones son lógicamente equivalentes.

Axioma 9.4.2 -- Axioma de asignación. Establece que, si una proposición verdadera P declara el estado del programa en el momento previo a la ejecución de una asignación de la forma `x = E`, entonces el predicado P sigue siendo verdadero después de la asignación, considerando que la variable x tiene ahora el valor de la evaluación de la expresión E.

$$\vdash \{P[E/x]\} \texttt{ x = } E \ \{P\}$$

En el axioma de la asignación:
$P[E/x] \equiv \int(P, \langle x, E \rangle)$
x es una variable.
E es una expresión válida en el código fuente.
P es una expresión lógica.

Ejemplo 9.3

Supongamos que se debe verificar la instrucción `b = a**2`. En el código fuente esta instrucción podría lucir como:

```
def areaCuadrado(a:int)->int:
    # <------ Punto de precondición ........... P
    b = a ** 2  # <- código a ser verificado ... {C}
    # <------ Punto de postcondición ........... Q
    return a
```

Al hacer `areaCuadrado(10)`, el primer punto de verificación $P \leftarrowtail (a \equiv 10)$. Esperamos que en el segundo punto de verificación $Q \leftarrowtail (b \equiv 100)$, por lo que escribimos:

$$\{a \equiv 10\} \texttt{ b = a**2 } \{b \equiv 100\}.$$

Para hacerlo hemos de suponer que $a \equiv 10$ es equivalente a $b \equiv 100$, pero sustituyendo b por la expresión a^2:

$$P = \boxed{Q[a/b]}$$
$$(a \equiv 10) = \boxed{Q[a/b]}$$
$$(a \equiv 10) = \boxed{\int\left(b \equiv 100, \langle b, a^2 \rangle\right)}$$
$$(a \equiv 10) = \boxed{a^2 \equiv 100}$$
$$(a \equiv 10) = \boxed{a \equiv 10}$$

Con lo que escribimos la conclusión:

$$\vdash \{a \equiv 10\} \texttt{ b = a**2 } \{b \equiv 100\}$$

9.4.3 Reglas de la consecuencia

En ocasiones las condiciones pueden ser «ajustadas», fortaleciendo la precondición o bien debilitando la postcondición. El término «fortaleciendo» se refiere a hacer más restrictiva la precondición. Al fortalecer la precondición, se agregan condiciones que el programa debe garantizar antes de la ejecución del código.

Regla 9.4.3 -- Fortalecer la precondición. Siendo P, Q y R tres proposiciones verdaderas y C una expresión de código fuente, se puede fortalecer la precondición P agregando la precondición R con:

$$\frac{\vdash \{P\}\ C\ \{Q\}, \vdash \rightarrow (R, P)}{\vdash \{R\}\ C\ \{Q\}}$$

Esta regla establece que, si tenemos una precondición P verdadera y una expresión de código C tal que, al ejecutar C, se verifica la postcondición Q; además, existe una

condición verdadera R que implica P, entonces se puede afirmar que R también es una precondición del código C que verifica la misma postcondición Q.

Observa que R no debe ser **F**, porque a pesar de que $\rightarrow(R,P) \mapsto \mathbf{V}$, no tendría sentido $\{\mathbf{F}\}\ C\ \{Q\}$ [ver la página 188].

La implicación $\rightarrow(R,P)$ involucra la precondición P y una precondición más fuerte R. Se dice que R es más fuerte porque los elementos que cumplen esta condición es seguro que también cumplen la condición más débil, por ejemplo, considera la condición $P \leftarrowtail \lambda x \bullet x > 0$. Una condición más fuerte que P puede ser $R \leftarrowtail \lambda x \bullet x > 5$, porque un elemento que cumple R, también cumple P.

Ejemplo 9.4

Considera la siguiente función que calcula la suma de los primeros números enteros:

```
def sumPE(n):
    y = (n * (n+1)) // 2
    return y
```

Antes de la ejecución la condición es $P \leftarrowtail \lambda n \bullet n \in \mathbb{Z}$, para indicar que n debe ser un número entero. Sin embargo se está permitiendo que n pueda tomar valores negativos, lo que seguramente causará comportamientos no deseados en el programa.

```
>>> sumPE(-2)
1
>>> sumPE(-2.5)
1.0
>>>
```

Observamos entonces que si $R \leftarrowtail \lambda n \bullet \wedge (n \in \mathbb{Z}, n > 0)$, se tiene claramente que $\vdash \rightarrow(R,P)$, es decir, un número que es entero mayor que cero implica que es entero.

```
def sumPE(n):
    assert y(isinstance(n, int), n>0)
    r = (n * (n+1)) // 2
    return r
```

El comando `assert` se utiliza par depurar programas. `assert` permite verificar que una condición se cumple. Si la condición no se cumple, el programa genera un error llamado `AssertionError` y despliega un mensaje.

```
>>> sumPE(-2)
Traceback (most recent call last):
    assert y(isinstance(n,int), n>0), "n debe ser un entero positivo"
AssertionError: n debe ser un entero positivo
>>> sumPE(2.5)
Traceback (most recent call last):
    assert y(isinstance(n,int), n>0), "n debe ser un entero positivo"
AssertionError: n debe ser un entero positivo
>>>
```

Por su parte, al considerar una condición Q, el término «debilitando la postcondición Q» significa encontrar una nueva condición Q' que sea más general o menos restrictiva que Q. Matemáticamente, esto es que $\rightarrow(Q,Q')$ sea verdadera, lo que quiere decir que si Q' es verdadera, también Q lo será, aunque no necesariamente al revés. En otras palabras, Q' cubre más casos que Q.

Regla 9.4.4 -- Debilitar la postcondición. Si un segmento de código C garantiza una postcondición Q y esta implica otra condición R se debilita la postcondición.

$$\frac{\vdash \{P\}\ C\ \{Q\}, \vdash \rightarrow(Q,R)}{\vdash \{P\}\ C\ \{R\}}$$

Esta regla establece que, si tenemos una precondición P verdadera y una expresión de código C tal que, al ejecutar C, se verifica la postcondición Q; además, existe una condición verdadera R que es implicada por Q, entonces se puede afirmar que también P es una precondición verdadera que al ejecutar el código C también verifica la postcondición R.

Ejemplo 9.5

Considera el siguiente código que aumenta el peso actual [línea 192]. Tanto el `pesoActual` como `pesoAgregado` son valores numéricos mayores que cero:

```
def aumentaPeso(pesoActual, pesoAgregado):
    assert y(pesoActual>0, pesoAgregado>0), "Se requieren dos valores positivos."
    pesoAumentado = pesoActual + pesoAgregado
    assert pesoAumentado > pesoActual
    return pesoAumentado
```

El predicado $P \leftarrow (y(\texttt{pesoActual} > 0, \texttt{pesoAgregado} > 0))$ debe ser **V** antes de ejecutar la línea `pesoAumentado = pesoActual + pesoAgregado` y se ha previsto que al terminar, cumpla el predicado $Q \leftarrow (\texttt{pesoAumentado} > \texttt{pesoActual})$.

Sin embargo $\rightarrow(\texttt{pesoAumentado} > \texttt{pesoActual}, \texttt{pesoAumentado} > 0)$ porque ambos valores, tanto `pesoActual` como `pesoAgregado` son valores positivos. Entonces, si hacemos $R \leftarrow (\texttt{pesoAumentado} > 0)$ estaremos seguros que también se cumplirá al terminar el código ejecutado.

Ya que $\vdash \rightarrow(Q,R), \{P\}\ C\ \{Q\}$, podemos asegurar:

$$\vdash (y(\texttt{pesoActual} > 0, \texttt{pesoAgregado} > 0))$$
$$\texttt{pesoAumentado = pesoActual + pesoAgregado}$$
$$\boxed{(\texttt{pesoAumentado} > 0)}$$

Lo cual es un debilitamiento de la postcondición.

9.4.4 Regla de composición

La regla de composición permite combinar dos afirmaciones sobre programas para formar una afirmación compuesta que describe la ejecución secuencial de esos programas.

Regla 9.4.5 -- Composición de instrucciones. Si tenemos dos afirmaciones $\{P\}\ C_1\ \{Q\}$ y $\{Q\}\ C_2\ \{R\}$, donde P, Q y R son predicados lógicos, y C_1 y C_2 son segmentos de código [programas], entonces podemos combinar estas afirmaciones para formar una nueva afirmación compuesta:

$$\frac{\vdash \{P\}\ C_1\ \{Q\}, \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}$$

En la regla el símbolo «;» representa la secuencia de ejecución, lo que significa que primero se ejecuta C_1 y luego C_2, en ese orden. Esta regla de composición permite razonar sobre la ejecución secuencial de programas.

Ejemplo 9.6

Considera el siguiente programa que calcula el precio total de un producto con valor inicial a, agregando al precio el impuesto al valor agregado.

```
def precioIVA(a):
    assert a >= 10.0
    # P ↤ (a ≥ 10.0)
    IVA = a * (16/100)
    # Q ↤ (IVA ≥ 1.6)
    precio = a + IVA
    # R ↤ (precio ≥ 11.6)
    return precio
```

Vamos a considerar las siguientes afirmaciones:

$$\{a \geq 10.0\} \texttt{ IVA = a * (16/100) } \{IVA \geq 1.6\}$$
$$\{IVA \geq 1.6\} \texttt{ precio = a + IVA } \{precio \geq 11.6\}$$

Podemos establecer usando la regla de la consecuencia que si a es un número real mayor o igual a 10.0 [precondición P], entonces `precio` será siempre mayor o igual a 11.6 al finalizar el código [postcondición R] porque `IVA` siempre será mayor o igual a 1.6 [condición intermedia Q].

$$\{a \geq 10.0\} \texttt{ IVA = a * (16/100); precio = a + IVA } \{precio \geq 11.6\}$$

9.4.5 Regla de la condición `if – then – else`

Debido a que en los trabajos originales las instrucciones se basaron en lenguajes imperativos como `Pascal`, la instrucción condicional se escribe como:

```
if <cond> then
  begin
  :
  end
else
  begin
  :
  end;
```

Sin embargo, la sintaxis de `Python` omite la palabra `then` y los ámbitos `begin-end` son expresados al identar bloques de código; por lo que en este texto se hará una ligera modificación a la regla de la condición, para ajustar la expresión a la sintaxis de `Python`, que es la sintaxis que ya se ha comentado antes [página 101 y siguientes].

```
P # <-- precondición P
if B : # <-- precondición B
  C1 # <-- Código cuando B es verdadero
else:
  C2 # <-- Código cuando B no es verdadero
Q # <-- postcondición Q
```

Esta regla permite especificar condiciones antes y después de la ejecución de una bifurcación condicional, asegurando que ciertas propiedades se mantengan independientemente de la rama que se ejecute.

Regla 9.4.6 -- Expresión condicional. Si la precondición P es verdadera y la condición B es verdadera, entonces el bloque de código C_1 debe llevar a la postcondición Q. Por otro lado, si la precondición P es verdadera pero la condición B es falsa $[\neg B]$, entonces el bloque de código C_2 debe llevar a la postcondición Q. En cualquier caso, P es una precondición para que o bien se ejecute C_1 o se ejecute C_2, lo que debe llevar a que la condición Q sea verdadera.

$$\frac{\vdash \{\wedge(P,B)\}\ C_1\ \{Q\},\ \vdash \{\wedge(P,\neg B)\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1\ \texttt{if}\ B\ \texttt{else}\ C_2\ \{Q\}}$$

Observa las precondiciones $\wedge(P,B)$ y $\wedge(P,\neg B)$, ambas coinciden en $\vdash P$ y difieren en que en una de ellas $\vdash B$ y en la otra $\vdash \neg B$. Esto significa que la precondición común P es la condición *invariante* de la expresión.

La otra expresión B, se conoce como la *guarda*. Es una expresión booleana que determina cuál de los dos bloques de código $[C_1$ o $C_2]$ se ejecutará, basado en su evaluación a **V** o **F**. En una instrucción condicional como el `C1 if B else C2`, B es la condición que se evalúa y determina la rama del código que será ejecutada.

El término *guarda* se utiliza frecuentemente en sistemas de modelado y diseño de software [OMG17, GBB05], para establecer las condiciones que deben ser cumplidas para que cierta parte del modelo se ejecute.

Ejemplo 9.7

Considera el código siguiente, que muestra la función `incDec`$(base,k)$ que modifica el valor de *base* que aumentará si k es positivo o bien decrementará el valor de *base* en caso de k no sea positivo.

```
def aumento(sueldo, ant):
    assert isinstance(sueldo, int)
    assert isinstance(ant, int)
    assert sueldo >= 0 and ant >= 0, "Se requieren valores enteros no negativos"
    # P ↤ ⋀(sueldo ≥ 0, sueldo ∈ ℤ, ant ∈ ℤ)
    # B ↤ ant > 24
    if ant>24: # ∧(P,B)
        res = sueldo + (sueldo // 2)   # C1
    else: # ∧(P,¬B)
        res = sueldo   # C2 "
    # Q ↤ ⊕(res ≡ sueldo, res ≡ sueldo + ⌊sueldo/2⌋)
    return res
```

En este ejemplo, la condición invariante es $P \leftarrowtail \bigwedge(sueldo \geq 0, sueldo \in \mathbb{Z}, ant \in \mathbb{Z})$, para establecer que el sueldo es un número entero positivo y *ant* es un número entero. La condición guarda es $ant \geq 24$ para determinar si se ha trabajado por al menos 24 meses o no.

Después de la ejecución de la instrucción `if`, en donde se ha ejecutado ya sea la instrucción `res=sueldo+(sueldo // 2)` o bien `res=sueldo`, la condición que se cumple es una de las dos, ya sea $res \equiv sueldo$ o bien $res \equiv sueldo + \lfloor \frac{sueldo}{2} \rfloor$.

9.4.6 Regla del ciclo while

Esta regla se centra en demostrar la validez de un programa que contiene un ciclo `while`. Si una afirmación es verdadera antes de entrar al ciclo y se mantiene cada vez que se completa el ciclo, entonces cierta propiedad será verdadera al salir del ciclo.

Regla 9.4.7 -- Expresión condicional. Si dos afirmaciones P y R son verdaderas antes del ciclo, y si P se mantiene cada vez que se ejecuta el cuerpo del ciclo C, entonces la negación de R ocasiona que el ciclo se interrumpa y deje de ejecutar el código C.

$$\frac{\vdash \{\wedge(P,R)\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{while}\ R : C\ \{\wedge(P,\neg R)\}}$$

La declaración $\{\wedge(P,R)\}\ C\ \{P\}$ indica que hay dos condiciones P y R que son verdaderas antes del código C, pero después del código solo se verifica P. Esto significa que mientras $\vdash R$ se ejecutará el código.

Como en la regla condicional, la condición P es la condición invariante del ciclo `while`, y la condición R es la condición guarda. La condición guarda es la que determina cuándo salir del ciclo, que es *mientras* que R sea verdadero.

Ejemplo 9.8

Considera la función factorial. Esta versión obtiene el factorial de un número entero positivo n mediante un ciclo `while` que itera el bloque de código C mientras el valor de n siga siendo mayor que cero. Cuando el valor de n llega a ser cero, la condición guarda $n > 0$ deja de ser verdadera, lo que ocasiona que el ciclo ya no se realice.

```
def factorial(n):
    assert isinstance(n, int), "Se requiere un valor entero"
    # P ↤ n ∈ ℤ
    assert n>0, "se requiere un valor mayor que cero"
    # R ↤ n > 0
    fact = 1
    # ⊢ ∧(n ∈ ℤ, n > 0)
    assert isinstance(n, int) and n>0
    while n > 0:       # ------------+
        fact = fact * n #            C
        n = n - 1      # ------------+
    # ⊢ ¬(n > 0)
    assert isinstance(n, int) and neg(n>0)
    return fact
```

Debido a que $\vdash \{\wedge(P,R)\}\ C\ \{P\}$, donde:

$P \leftarrow\!\shortmid \wedge(n \in \mathbb{Z}, n > 0)$,

$R \leftarrow\!\shortmid (n > 0)$,

$C \leftarrow\!\shortmid$ `while R: fact=fact*n; n=n + 1`

se puede concluir que $\{P\}$ `while R: fact=fact*n; n=n + 1` $\{\wedge(P,\neg R)\}$.

Para tener un marco de verificación formal completo, hay que crear una regla de verificación para cada expresión en el lenguaje que se desee verificar. En `Python`, aunque hay otras formas de crear ciclos, y otras maneras de crear expresiones condicionales, las aquí mostradas son suficientes para dar una idea general de los elementos que están involucrados en la lógica de Hoare.

9.5 Verificación de programas

La verificación de programas, desde este enfoque axiomático, es una manera sistemática para demostrar la corrección de un programa computacional. Se basa en el uso de las denominadas «tripletas de Hoare», que son declaraciones que describen

las precondiciones, las instrucciones y las postcondiciones de un fragmento de código, que puede ir desde una sola línea de código hasta todo el programa por completo.

El objetivo es demostrar lógicamente que, si la precondición es verdadera antes de la ejecución del programa, entonces la postcondición será verdadera después de la ejecución del programa.

Otro modo de entender esto es, si P y Q son proposiciones verdaderas, sabemos que $\vdash \rightarrow (P, Q)$, significa que el estado que representa P lleva [implica] a un estado Q, y el código que se ejecuta después de la verificación de P, es lo que hace que Q ocurra.

Este enfoque ayuda a los programadores a razonar formalmente sobre la corrección de sus programas, identificar posibles errores y garantizar que ciertas propiedades sean válidas antes y después de la ejecución del código.

9.5.1 Correctitud parcial y total de un programa

La correctitud parcial y total se relacionan con la verificación de programas mediante las tripletas de Hoare, pero con un enfoque más específico:

Correctitud parcial: Se refiere a la capacidad de un programa para cumplir las condiciones especificadas [precondiciones y postcondiciones] en un contexto particular o para un conjunto específico de instrucciones, como se ha mencionado anteriormente.

Correctitud total: Se relaciona con la propiedad de terminación de un programa. Implica demostrar que el programa finaliza su ejecución en un tiempo finito para cualquier entrada o condición inicial válida. La correctitud total se refiere a garantizar que el programa no entre en ciclos infinitos o no termine de manera impredecible, sino que siempre finalice su ejecución en un tiempo razonable. A diferencia de la correctitud parcial, la correctitud total suele denotarse como:

$$[P]\ C\ [Q]$$

Donde:

P, C, Q son como antes,

Los corchetes marcan la diferencia entre ambas aproximaciones.

Para esta sección vamos a considerar un programa en `Python` que simplemente obtiene el promedio de los números almacenados en una lista.

El proceso de la verificación formal de un programa para garantizar la correctitud total es necesario conseguir dos objetivos:

- Verificar la correctitud parcial.
- Verificar la terminación del programa.

El algoritmo para calcular el promedio sigue los siguientes pasos:

1. Se requiere una lista de números enteros `v`. El resultado debe ser un número real.
2. El valor `suma` debe empezar con `0`.
3. Para cada valor `vi` en la lista `v`, se debe hacer:
   a) actualizar el valor de `suma` con el valor de `suma` incrementado en `vi` unidades.
4. Si la lista no es vacía, entonces:
   a) El promedio `p` será `suma / len(v)`

   En otro caso:

a) El promedio `p` será `0`.

5. Se debe devolver el valor `p`.

El programa objetivo, al cual llamaremos `promedio` es el siguiente:

```
def promedio(v: list) -> float:
    """
    Calcula el promedio de los números en la lista.
    """
    # ⊕(v ≡ ⟨ ⟩, ∀v_i ∈ ⟨v_1,...,v_n⟩ : v_i ∈ ℤ^≥0)
    suma = 0
    # ∧(⊕(v ≡ ⟨ ⟩, ∀v_i ∈ ⟨v_1,...,v_n⟩ : v_i ∈ ℤ^≥0), suma ≡ 0)
    for vi in v:
        suma = suma + vi
    # ∧(⊕(v ≡ ⟨ ⟩, ∀v_i ∈ ⟨v_1,...,v_n⟩ : v_i ∈ ℤ^≥0), suma ≡ Σ_{i=0}^{|v|} v_i)
    if v != []:
        p = suma / len(v)
    else:
        p = 0
    # ⊕(p ≡ (Σ_{i=0}^{|v|} v_i)/|v|, p ≡ 0)
    return p
```

```
# Ejemplo de uso
>>> listaNoVacia = [1, 2, 3, 4, 5]
>>> listaVacia = []
>>> promedio(listaNoVacia)
3
>>> promedio(listaVacia)
3
>>>
```

El programa `promedio` toma como entrada una lista de números enteros. Al inicio, la lista puede ser una lista vacía o una lista de números enteros no negativos.

Al finalizar el programa, se garantiza que $\oplus\left(p \equiv \frac{\sum_{i=0}^{|v|} v_i}{|v|}, p \equiv 0\right)$, esto es que, o bien el resultado es 0 o bien es la suma de los números en la lista dividido por la cantidad de números. Con esto se verifica la correctitud parcial de programa.

La correctitud total se concluye al garantizar que el programa termina correctamente. Hay dos situaciones que deben ser estudiadas para ser evitadas:

1. Que el programa termine de manera incorrecta.
2. Que el programa no termine.

Un programa puede terminar de manera incorrecta en dos situaciones:

Terminación equivocada. Es cuando el programa termina pero con un resultado equivocado. Esta situación debería haber sido resuelta en la correctitud parcial.

Terminación causada por un error. Es cuando surge un error de cualquier índole que impide la continuación normal del programa. Por ejemplo una división por cero o la lectura de un dato en un lugar inexistente, hay una inmensa cantidad de posibles errores que pueden ocasionar la terminación abrupta a causa de un error.

La segunda situación que debe ser evitada es aún más delicada, es decir, debemos asegurar que el programa termine.

Esto ocurre frecuentemente cuando se crean ciclos que son controlados manualmente, como los ciclos `while`, donde a diferencia de los ciclos `for`, la continuidad

del ciclo depende de una expresión booleana cuya variable es modificada dentro del cuerpo del ciclo.

En los ciclos `for` ocurre diferente. Un poco depende del diseño del lenguaje, por ejemplo en `Python`, un ciclo `for` itera una variable dentro de un rango o bien dentro de una colección de elementos. En `Python` un generador proporciona el siguiente valor que la variable de control debe tener. El mismo generador proporciona el final del ciclo, es por eso que un programa como:

```
for i in range(0,10):
    print(i, end=', ')
    i = i - 1
```

no tiene problema en desplegar solamente los diez dígitos:

```
>>> %Run -c $EDITOR_CONTENT
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
>>>
```

En el caso del programa `promedio`, la situación es simple, ya que se ha resuelto el problema mediante un ciclo `for`:

```
for vi in v:
    suma = suma + vi
```

`Python` crea un generador que proporciona uno a uno los valores en la lista `v` y los va asignando a la variable de control `vi`, de modo que en cada iteración la variable `suma` actualiza su valor agregando el valor de `vi`.

Como una de las condiciones previas es que, suponiendo que `v` no es vacía, todos sus elementos son números enteros no negativos, por lo que el valor de `suma` nunca podrá ser menor que 0.

El mismo generador del ciclo `for` se detiene al proporcionar el último elemento de la lista, dando por terminado el ciclo.

9.5.2 Metodología para la verificación formal

Aunque no hay una única manera de hacer la verificación formal de un programa, aquí te sugiero estos nueve pasos para hacer una verificación formal. Estos pasos no están asociados con ningún lenguaje de programación imperativa, por lo que pueden ser utilizados en la verificación de programas en cualquier lenguaje.

1. **Comprensión del programa**: Entiende el programa en cuestión: su propósito, las estructuras de control [ciclos, condicionales], las variables involucradas y el conjunto de operaciones que se realizan.
2. **Establecer precondiciones**: Define las condiciones que deben cumplirse antes de que el programa comience a ejecutarse. Esto incluye el estado inicial de las variables y estructuras de datos.
3. **Identificar invariantes de ciclo**: Identifica las propiedades que deben mantenerse verdaderas antes, durante y después de cada iteración de los ciclos. Estas invariantes son cruciales para asegurar que el algoritmo se comporta como se espera.
4. **Determinar la postcondición**: Define las condiciones que deben cumplirse al finalizar la ejecución del programa. Esto describe el estado final deseado de las variables y estructuras de datos.

5. **Verificación paso a paso**: Sigue cada paso del programa, verificando que las invariantes de ciclo se mantienen antes, durante y después de cada iteración. Asegúrate de que las operaciones realizadas por el programa no violen las invariantes definidas y se alineen con la lógica del algoritmo.
6. **Verificar la postcondición**: Al finalizar la ejecución del programa, analiza las condiciones establecidas en la postcondición y verifica que se cumplan. Esto confirma que el programa ha completado su tarea correctamente.
7. **Demostración formal**: Para cada etapa de la verificación, documenta claramente los pasos seguidos y las razones detrás de cada afirmación.

Cada paso puede hacerse en cualquier momento de la verificación, considerando que una instrucción puede ser la invocación de un subprograma, lo que significaría la verificación de ese subprograma.

La verificación formal utilizando la lógica de Hoare puede ser detallada y requiere un análisis cuidadoso de cada paso del programa para garantizar su corrección. Es importante ser sistemático y metódico en el seguimiento de cada variable y estructura de control para demostrar que el programa se comporta como se espera bajo todas las condiciones posibles.

Ejercicios

1. Encierra en un círculo la opción que creas que mejor responde a la siguiente pregunta: ¿Cuál es el principio fundamental de la lógica de Hoare?
   a) Demostrar la eficiencia de los algoritmos en términos de tiempo de ejecución y espacio en memoria requerido.
   b) Verificar la seguridad de los programas mediante el análisis de posibles vulnerabilidades.
   c) Proporcionar un procedimiento sistemático y formal para razonar sobre la corrección de programas mediante tripletas que relacionan precondiciones, comandos y postcondiciones.
   d) Establecer un estándar de codificación para la programación en diferentes lenguajes de programación.
2. Encierra en un círculo la opción que creas que mejor responde a la siguiente pregunta: ¿Cómo se aplica la verificación de correctitud de los programas?
   a) Comparando el rendimiento de los programas con conjuntos de pruebas pre-establecidos, para comparar los resultados obtenidos experimentalmente, con los resultados reportados.
   b) Utilizando tripletas de Hoare, que relacionan precondiciones, código fuente y postcondiciones, para razonar formalmente sobre el estado correcto de los programas, y eventualmente sabiendo que el programa termina correctamente.
   c) Analizando el código fuente del programa en busca de errores sintácticos y semánticos.
   d) Ejecutando pruebas unitarias y de integración con el ecosistema de implementación del software, para identificar fallas en tiempo de ejecución.
3. En cada una de las siguientes expresiones de sustitución [página 187] , escribe la expresión resultante de efectuar la sustitución.
   a) *Ejemplo.* $\int(a+b, \langle a, x \rangle) \mapsto x + b$
   b) $\int(3x + 2y + 5, \langle y, g \rangle)$
   c) $\int\left(x^2 + 2xy + y^2 + 3x + 4y + 6, \langle y, u \rangle\right)$
   d) $\int\left(2x^3 + 3y^3 - 5xy + 4x - 6y + 7, \langle x, u \rangle\right)$
   e) $\int\left(x^4 - 4x^3y + 6x^2y^2 - 4xy^3 + y^4 + 2x^2 - 3xy + 5y^2 + 8, \langle x, w \rangle\right)$
4. La definición de sustitución de la página 187, aplica únicamente un par $\langle a, b \rangle$ sustitutivo a una expresión E. Considera la siguiente definición que extiende el concepto de sustitución:

> **Definición 9.5.1 -- Sustitución extendida.** La función extendida $\int^*(E, [\langle a, b \rangle \ldots])$ toma como entrada una expresión E y una lista de pares de sustituciones $[\langle a, b \rangle, \ldots]$. Cada par en la lista contiene como primera entrada una literal que aparece en la expresión y como segunda entrada, una literal que se usará para reemplazar la primera literal en la expresión. El resultado de esta función es una nueva expresión en donde se han sustituido, para cada par de la lista, cada ocurrencia de la literal dada en el primer elemento del par, por la literal que aparece en el segundo elemento del par, aplicando las sustituciones de manera secuencial según el orden en que aparecen en la lista.

Aplica la sustitución extendida para las siguientes expresiones:

a) *Ejemplo.* $\int^*(a+b,[\langle a,x\rangle,\langle b,y\rangle])$

$$\begin{aligned}\int^*(a+b,[\langle a,x\rangle,\langle b,y\rangle]) &= \int^*(\int(a+b,\langle a,x\rangle),[\langle b,y\rangle])\\ &= \int^*(x+b,[\langle b,y\rangle])\\ &= \int^*(\int(x+b,\langle b,y\rangle),[])\\ &= \int^*(x+y,[])\\ &\mapsto x+y\end{aligned}$$

b) $\int^*\left(2x^3+3y^2+k,[\langle x,a\rangle,\langle y,b\rangle]\right)$

c) $\int^*\left(\alpha\frac{u}{2}+\beta\frac{s}{4},[\langle u,x\rangle,\langle s,y\rangle]\right)$

d) $\int^*(\texttt{triple(z) / sumaCuadrados(a,b)},[\langle \texttt{z,c}\rangle,\langle \texttt{a,x}\rangle,\langle \texttt{b,f}\rangle])$

5. Escribe una función en `Python` que implemente la función de sustitución $\int$. La función se debe llamar `sust`, debe recibir como entrada una cadena de caracteres `expr` como la expresión y un par `v` que contenga la variable actual y la variable sustituta. El programa debe generar como salida una nueva expresión con la variable sustituida.

```
def sust(expr:str, v:list) -> str:
    """
    Sustituye el segundo elemento de v en cada ocurrencia
    del primer elemento de v en la expr.
    """
    pass
```

```
>>> sust("a+b", ["a", "x"])
'x+b'
>>>
```

6. Considera la siguiente función en `Python`:

```
1 def sumar(a:int, b:int) -> int:
2     """
3     Suma los enteros a y b
4     """
5     # Precondición
6     c = a + b
7     # Postcondición
8     return c
```

Escribe la tripleta de Hoare correspondiente para verificar la línea 6.

7. Considera la siguiente función en `Python`:

```
1 def sumaCuadrados(a:int, b:int) -> int:
2     """
3     Suma el cuadrado de a con el cuadrado de b
4     """
5     c = cuadrado(a) # no implementado, pero devuelve a^2.
6     d = cuadrado(b) # no implementado, pero devuelve b^2.
7     e = sumar(c,d)
8     return e
```

Ahora considera la siguiente tripleta de Hoare para verificar la línea 7:

$$\{c \in \mathbb{Z} \land d \in \mathbb{Z}\}\ \texttt{e = sumar(c,d)}\ \{e \in \mathbb{Z}\}$$

Critica las precondiciones y postcondiciones establecidas. Reescribe la tripleta de Hoare con una versión que atienda las críticas que has encontrado.

8. Diseña tripletas de Hoare para un algoritmo de búsqueda (como la búsqueda binaria), incluyendo las precondiciones y postcondiciones.

```python
from typing import List

def busquedaBinaria(target: int, nums: List[int]) -> bool:
    """
    Realiza una búsqueda binaria para encontrar el target
    en una lista ordenada de enteros.

    Parámetros:
    - target (int): El número entero que se busca en la lista.
    - nums (List[int]): Una lista de números enteros ordenada
          de menor a mayor.
    Devuelve:
    - bool: True si target se encuentra en la lista,
            False en caso contrario.
    """
    low = 0
    high = len(nums) - 1

    while low <= high:
        mid = (low + high) // 2
        guess = nums[mid]
        if guess == target:
            return True
        if guess > target:
            high = mid - 1
        else:
            low = mid + 1
    return False
```

   a) Escribe una tripleta de Hoare para verificar la línea 21.
   b) Escribe una tripleta de Hoare para verificar la línea 23.
   c) Escribe una tripleta de Hoare para verificar la línea 25.

9. Explica cómo las tripletas de Hoare pueden ayudar a identificar errores en el estado de un programa antes y después de su ejecución.

10. Explica qué sentido tienen las líneas 5 y 7 del siguiente código.

```python
def inversoMult(n: float) -> float:
    """
    Calcula el inverso multiplicativo de un número dado.
    """
    assert n != 0, "Precondición fallida:__________________."
    inverso = 1 / n
    assert n * inverso == 1, "Postcondición fallida: _____________________."
    return inverso
```

   a) Escribe en forma de string una descripción textual de la precondición en la línea 5.
   b) Escribe en forma de string una descripción textual de la postcondición en la línea 7.

11. Explica el axioma de asignación utilizando las siguientes tripletas de Hoare:
   a) *Ejemplo.* $\{x+1>0\}$ `x = x + 1` $\{x>0\}$

   > "Supongamos que tenemos la postcondición x >0 después de una asignación, y queremos verificar que esta postcondición sea verdadera. La asignación en `Python` es `x = x + 1`. Usando el Axioma del Paso, la precondición necesaria es $x+1>0$. Si esta precondición es verdadera antes de la asignación, entonces la postcondición $x>0$ será verdadera después".

   b) $\{count \equiv n\}$ `count = count + 1` $\{count \equiv n+1\}$

c) $\{x \equiv n\}$ `x = x * x` $\left\{x \equiv n^2\right\}$

d) $\{a \equiv A \land b \equiv B\}$

```
temp = a
a = b
b = temp
```

$\{a \equiv b \land b \equiv A\}$

12. En la lógica de Hoare, la regla de la consecuencia permite ajustar las precondiciones y postcondiciones de las tripletas de Hoare para facilitar la demostración de la correctitud de los programas. En los siguientes casos, explica cómo la regla de la consecuencia puede ser utilizada para simplificar la verificación de la correctitud del programa.

a) *Ejemplo.* Verificación de una suma acumulativa. Considera una tripleta de Hoare con el fragmento de código que suma los números de 1 a n:

$\{n \geq 1\}$

```
total = 0
for i in range(1, n+1):
    total += i
```

$\{total \equiv \frac{n(n+1)}{2}\}$.

Para simplificar la verificación, podemos utilizar una precondición más fuerte:

Precondición más fuerte: $\{n \in \mathbb{Z}^+\}$

Postcondición: $\{total \equiv \frac{n(n+1)}{2}\}$ (la misma que antes).

Con la regla de la consecuencia, si n es un entero positivo, necesariamente $n \geq 1$, lo que simplifica la justificación de la precondición original.

b) Verificación de una función de búsqueda. La siguiente tripleta de Hoare incluye una función que verifica si un elemento está en una lista:

$\{lst \neq []\}$ ▷ *La lista no está vacía.*

```
def enLista(elem, lst : list) -> bool:
    return elem in lst
```

$\{elem \in lst\}$.

c) Inicialización de una variable. La siguiente tripleta de Hoare inicializa una variable: $\{\mathbf{V}\}$ `x = 0` $\{x \equiv 0\}$.

Circuitos digitales 10

10.1 Lógica digital

La lógica digital es una rama de la lógica que se emplea en la electrónica y las ciencias computacionales que se ocupa del diseño y gestión de sistemas que procesan información codificada en señales digitales. En la lógica digital, las señales tienen estados discretos, típicamente dos estados: alto [1] y bajo [0, en ocasiones -1]. Estos estados representan los valores binarios en un sistema de numeración base 2 y fundamentan el procesamiento de datos en dispositivos electrónicos digitales, como las computadoras, teléfonos móviles, sistemas de GPS, entre muchos otros.

10.1.1 Sistema binario

La base de la lógica digital es el sistema binario, que utiliza dos estados o valores: 0 y 1. Toda la información en los dispositivos digitales se representa mediante combinaciones de estos dígitos binarios. Cada dígito se conoce como *bit* [abreviatura de Binary digIT].

En este sistema, cada dígito en un número representa una potencia de 2, con el dígito menos significativo colocado más a la derecha, que representa 2^0; mientras que a la izquierda se coloca el más significativo. Un número binario se escribe colocando un 2 como subíndice delante del dígito menos significativo, por ejemplo 1011_2, esto es para hacerlo diferente al número 1011 [mil once] en el sistema decimal.

Así, el número binario 1011_2 representa el número 11 en decimal:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11_{10}.$$

Lo contrario, es decir, para expresar un número en decimal en el sistema binario, se puede utilizar el método de la división sucesiva. Esto es, dividir el número decimal entre 2 repetidamente hasta que el cociente sea 0, registrando el residuo de cada división. Para formar el número decimal equivalente, se escribe el resto de cada división, ya sea 0 o 1, colocados desde el último hasta el primero.

Algoritmo de divisiones sucesivas:

Se requiere un número en sistema decimal:

1. Divide el número decimal entre 2.
2. Anota el cociente y el residuo.
3. Usa el cociente obtenido como el nuevo número a dividir por 2.
4. Anota el residuo en una palabra, escribiendo cada resto a la izquierda del anterior.
5. Repite los pasos 1 a 5 hasta que el cociente sea 0.

El número binario es la secuencia de residuos. Veamos un ejemplo con el número decimal 11:

1. 11 dividido entre 2 es igual a 5 con un resto de 1. Por lo que se escribe el resto 1:

   1

2. 5 dividido entre 2 es igual a 2 con un resto de 1. Por lo que se escribe 1 a la izquierda de 1:

   11

3. 2 dividido entre 2 es igual a 1 con un resto de 0. Por lo que se escribe 0 a la izquierda de 11:

   011

4. 1 dividido entre 2 es igual a 0 con un resto de 1. Por lo que se escribe 1 a la izquierda de 011:

   1011

Como el cociente ya es 0, el procedimiento termina, y el número en binario equivalente a 11_{10} es 1011_2.

El sistema binario es importante en lógica digital, porque es posible relacionar el valor 1 con el valor booleano **V** y el 0 con **F**. Además, estos números binarios pueden representar diferentes niveles de voltaje en un circuito electrónico, por ejemplo 0 puede indicar que el circuito tiene 0V, mientras que el 1 puede representar 5V [puede ser diferente, en dependencia de las características del componente electrónico].

De este modo, la secuencia de valores binarios $\langle 1, 0, 1, 1 \rangle$, que es representada por el número binario 1011_2, representa la secuencia de niveles de voltaje $\langle 5, 0, 5, 5 \rangle$, lo que indica que el dispositivo electrónico recibe 5V en un primer momento, luego 0, luego 5 y finalmente se mantiene en 5, todos ellos de igual duración, que es gobernada por un pulso llamado reloj y cada tic del reloj marca el inicio de una nueva señal.

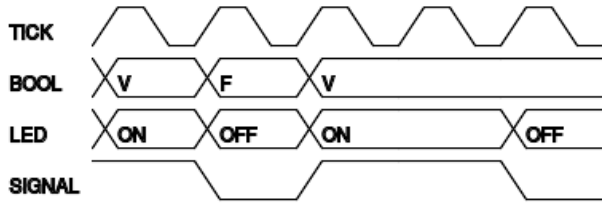


*Figura 10.1: Diagrama de tiempos para la palabra $\langle 1,0,1,1 \rangle$ que incluye la señal de un reloj. Observa que el valor **V** de más a la derecha no aparece en el diagrama, esto es porque el valor se ha mantenido los últimos dos tics. Cada tic incluye cuatro eventos: sube el voltaje, se mantiene un momento, baja el voltaje y se mantiene un momento.*

10.1.2 Compuertas lógicas

Considerando un 1 como indicador de voltaje alto en un circuito electrónico equivalente a **V**, y un 0 como indicador de un voltaje bajo en el circuito equivalente a **F**, una compuerta lógica es un componente electrónico que recibe una o más señales binarias y devuelve una señal binaria como salida.

Al igual que las funciones lógicas primitivas [página 97], las compuertas lógicas modelan el comportamiento de las principales funciones lógicas como la negación [página 113], la conjunción [página 115] y la disyunción [página 114], entre otras.

Las compuertas lógicas son elementos pictóricos que ilustran la entrada de señales lógicas y producen una salida. En la tabla 10.1 se muestran las principales compuertas lógicas y su significado.

Nombre	Símbolo	Nombre en lógica digital	Referencia
Negación		NOT	página 113
Conjunción		AND	página 115
Barra de Sheffer		NAND	página 126
Disyunción		OR	página 114
Flecha de Peirce		NOR	página 125
Disyunción exclusiva		XOR	página 121
Doble implicación		XNOR	página 119

Tabla 10.1: Principales compuertas lógicas. Cada compuerta está asociada con una función lógica primitiva, generalmente se dibujan con las señales de entrada a la izquierda y la salida a la derecha.

Antes de diseñar cualquier circuito electrónico utilizando compuertas lógicas, se debe entender el comportamiento de cada compuerta:

NOT: La salida es 1 cuando la entrada es 0 y la salida es 0 cuando la entrada es 1. En lógica digital, si p representa una proposición lógica, la negación de la proposición se representa como $\bar{p}$. Colocar una barra sobre el símbolo permite identificar el o los términos que están siendo afectados por la negación, por ejemplo $\overline{ABC}$ equivale a escribir $\neg(A \wedge B \wedge C)$ y en forma prefija es equivalente

a $\neg(\wedge(A, \wedge(B, C)))$.

AND: La salida es 1 si todas las entradas son 1. Es la forma digital de las proposiciones conjuntivas [ver la página 39]

NAND: Es el inverso de AND; la salida es 0 solo si todas las entradas son 1.

OR: La salida es 1 si al menos una de las entradas es 1. Es la forma digital de las proposiciones disyuntivas [ver la página 38].

NOR: Es el inverso de OR; la salida es 1 solo si todas las entradas son 0.

XOR: La salida es 1 si las entradas son diferentes.

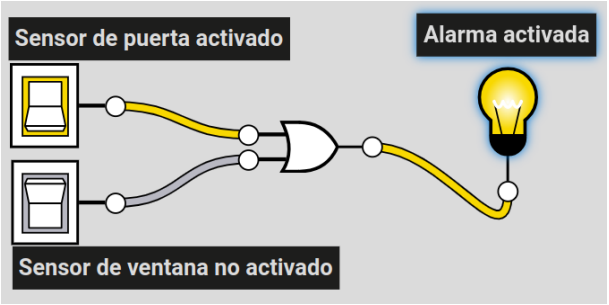
XNOR: La salida es 1 si las entradas son iguales.

La manera de utilizar las compuertas es tratarlas como bloques de construcción de circuitos digitales, se usan para realizar operaciones lógicas básicas. Cada compuerta tiene una o más entradas y produce una salida que se basa en alguna función lógica primitiva [ver el capítulo 5 en las páginas 109 y 110]. Un procedimiento general que se puede implementar para diseñar un circuito digital es el siguiente:

1. *Definición del problema.* En este paso se identifica claramente qué función debe realizar el circuito. Esto incluye definir las entradas, las salidas y cómo deben relacionarse las salidas con las entradas.
2. *Especificación de la tabla de verdad.* Para cada combinación posible de entradas, se debe especificar el estado deseado de la salida. La tabla de verdad es una representación sistemática de esta relación [ver el capítulo 5 en la página 107].
3. *Simplificación de la función lógica.* Se deben utilizar métodos de simplificación como los mapas de Karnaugh [Wak06, MC07, BV90] o el método algebraico de Boole [RK04, HH12, KB05] para reducir la función lógica a su forma más simple. Esto ayuda a minimizar el número de compuertas necesarias y, por tanto, el costo y la complejidad del circuito.
4. *Diseño del esquema del circuito.* Se dibuja un esquema del circuito con las compuertas sugeridas por la función reducida.
5. *Verificación.* Comprobar el diseño del circuito contra la tabla de verdad original para asegurar de que todas las combinaciones de entrada producen las salidas esperadas. Para esto es útil usar software de simulación de circuitos para facilitar este paso. Como ejemplos de software de simulación de uso libre están:
   - **LogiSim** que es una herramienta educativa para diseñar y simular circuitos lógicos digitales, se puede descargar en `http://www.cburch.com/logisim/es/index.html`
   - **KiCad EDA** es un software de código abierto para la automatización del diseño electrónico, disponible para muchas plataformas y se puede descargar desde `https://www.kicad.org/`
   - **logic.ly**, que es un simulador de lógica digital en línea, se puede visitar en `https://logic.ly/`.

Ejemplo 10.1

Se desea construir un sistema de alarma simple en una casa, que se activa cuando se abre una ventana o una puerta. Puedes usar una compuerta OR para este propósito. Dos sensores [uno en la puerta y otro en la ventana] se conectan a las entradas de la compuerta OR, de modo que la alarma [conectada a la salida] se activará si alguno de los sensores detecta una apertura.



Circuito creado en `logic.ly`

10.2 Especificación de la tabla de verdad

Después de haber especificado el problema, el cual incluye la descripción de cada una de las condiciones en las que se debe presentar el efecto deseado en el circuito, se tomas esas mismas condiciones y deben ser formalizadas.

La formalización de las condiciones de un circuito lógico es una descripción proposicional [capítulo 2, página 33] para formar proposiciones atómicas y con ellas, construir otras proposiciones moleculares, las cuales utilizan términos de enlace.

Cada proposición atómica como «El sensor de la puerta está encendido» se asocia con una literal, luego se coleccionan todas las literales y se forma una tabla con los valores booleanos digitales.

Ejemplo 10.2

Considera la siguiente declaración «El sensor de la puerta está activado», que permite obtener las siguientes dos proposiciones:

- ⟨*El sensor de la puerta está encendido*⟩
- NO ⟨*El sensor de la puerta está encendido*⟩ [página 37], en este caso la proposición opositiva se puede reescribir de manera equivalente como ⟨*El sensor de la puerta está desactivado*⟩.

Consideremos el caso que se presenta al tener dos sensores, digamos A y B; ambos sensores pueden estar ya sea activados o desactivados. Todos los posibles escenarios en los que se pueden observar ambos sensores producen una tabla como la que sigue:

A	B	Significado	Interpretación lógica	Álgebra booleana
1	1	Ambos sensores están activados	$A \wedge B$	AB
1	0	A está activado y B no está activado	$A \wedge \overline{B}$	$A\overline{B}$
0	1	A no está activado y B está activado	$\overline{A} \wedge B$	$\overline{A}B$
0	0	Ninguno de los sensores está activado	$\overline{A} \wedge \overline{B}$	$\overline{A}\overline{B}$

Cada renglón es la conjunción de las proposiciones, en su estado verdadero o como su negación. Para determinar el comportamiento esperado del sistema, se escribe un 1 en los renglones que presentan el comportamiento deseado, por ejemplo, si se desea que el sistema se active cuando A no esté activado, pero B sí, entonces se escribe un 1 en el renglón $\overline{A} \wedge B$. Cuando hay más de dos variables, cada conjunción es la conjunción generalizada [página 142] de todas las veriables.

Un **minitérmino** [*minterm* en inglés] es una expresión conjuntiva del álgebra booleana que contiene todas las variables incluídas en el sistema. Es evaluada como 1 [**V**] para una única combinación de esas variables, por ejemplo $pq\bar{r} \mapsto 1$, si el sistema considera las variables p, q y r.

Un **maxitérmino** [*maxiterm* en inglés], es una expresión disyuntiva del álgebra booleana que involucra todas las variables del sistema. Se evalúa como 0 [**F**] para una combinación específica del estado de las variables, por ejemplo $p + q + \bar{r} \mapsto 0$, si el sistema considera las variables p, q y r.

La función de la tabla de verdad queda determinada al crear una disyunción generalizada [página 144] con todas las conjunciones de aquellas combinaciones en donde se ha colocado un 1, esto es, de todos los minitérminos.

Supongamos que deseamos que el sistema se active cuando solamente uno de los dos sensores se activen, así la tabla de verdad queda conformada de la siguiente manera:

A	B	$F(A,B)$
1	1	0
1	0	1
0	1	1
0	0	0

El nombre F de la función es completamente arbitraria. Ya con la tabla de verdad, se puede especificar la función F como la disyunción generalizada de las conjunciones, que también se expresa como suma de productos o suma de minitérminos [MC07], la función se puede escribir como:

$$F \leftarrow \lambda A, B \bullet (A \wedge \overline{B}) \vee (\overline{A} \wedge B),$$

en términos de álgebra booleana se escribe como suma de productos, recordando que el operador de conjunción [producto] usualmente se omite al no haber confusión alguna:

$$F \leftarrow \lambda A, B \bullet (A\overline{B} + \overline{A}B).$$

Podemos utilizar nuevamente la función `TV` [página 134] y comprobar que la función $F(A,B)$ se comporta de la manera esperada, computando la función [páginas 130 y siguientes].

```
>>> TV(lambda A, B: o(y(A,neg(B)), y(neg(A), B)), "F")
    A     |    B     |  F(A,B)
----------+----------+----------
   True   |   True   |  False
   True   |  False   |   True
  False   |   True   |   True
  False   |  False   |  False
>>>
```

Como cada variable en un minitérmino se puede presentar en forma directa con un 1 o en forma de su complemento con un 0, un número binario puede representar un minitérmino si antes se determina el orden de las variables, ya como número binario, se puede representar como un número decimal.

Observa los siguientes ejemplos para un sistema de cuatro variables:

Expresión lógica	Álgebra booleana	Expresión binaria	Expresión decimal
$\overline{A} \wedge B \wedge \overline{C} \wedge D$	$\overline{A}B\overline{C}D$	0101	5
$A \wedge B \wedge C \wedge D$	$ABCD$	1111	15
$A \wedge \overline{B} \wedge \overline{C} \wedge D$	$A\overline{B}\overline{C}D$	0110	6

Se ha aprovechado la notación decimal para especificar una función lógica expresada como suma de minitérminos como:

$$F \leftarrow \lambda A, B, C, D \bullet \sum m(d, \dots)$$

donde $d, \dots$ representa una lista de números decimales que representan números binarios y a su vez representan minitérminos.

Ejemplo 10.3

La función lógica $F \leftarrow \lambda A, B, C, D \bullet (A\overline{B}\overline{C}D + \overline{A}\overline{B}CD)$ incluye a los minitérminos 1001 y 0011 que corresponden a los números decimales 9 y 3, por lo que la función lógica se puede expresar como la suma de los minitérminos:

$$F \leftarrow \lambda A, B, C, D \bullet \sum m(9, 3)$$

10.3 Simplificar una función lógica

Simplificar la función lógica significa crear una función lógica que sea equivalente a la original, pero que se exprese con menos conectores lógicos.

Se puede simplificar una función lógica utilizando las reglas de equivalencia lógica dadas por el álgebra booleana o también usando mapas de Karnaugh [MC07, HH12].

Ejemplo 10.4

La función $F \leftarrow \lambda A, B \bullet (A\overline{B} + \overline{A}B)$ tiene 5 compuertas: 2 negaciones, dos conjunciones y una disyunción. Pero se puede reducir a una sola compuerta $\oplus$: $F \leftarrow \lambda A, B \bullet (A\overline{B} + \overline{A}B)$

A	B	$\lambda A, B \bullet (A\overline{B} + \overline{A}B)$	$\lambda A, B \bullet (A \oplus B)$
1	1	0	0
1	0	1	1
0	1	1	1
0	0	0	0

10.3.1 Álgebra booleana

Es un sistema matemático que describe las operaciones lógicas y las relaciones entre los bits. Fue creada por George Boole en 1854 [Boo48] y se utiliza para analizar y simplificar circuitos lógicos. Utiliza operaciones como AND, OR, NOT, entre otras, para manipular y combinar valores binarios. Con el álgebra booleana, se pueden representar y resolver problemas lógicos de manera formal con el objetivo de llegar a una forma más simple de la función y que mantenga el mismo comportamiento lógico. Para utilizar álgebra booleana se pueden seguir los siguientes pasos:

1. **Identificar la función lógica**. Este paso se logra creando una función lógica conformada por la suma de los minitérminos o bien por el producto de los maxitérminos.
2. **Aplicar las leyes del Álgebra de Boole**. Se usan las identidades booleanas [páginas 61 y siguientes] para simplificar la función. Enseguida se muestran las leyes más utilizadas:
   - **Identidad:** $A + 0 \equiv 0 + A \equiv A$; $A \cdot 1 \equiv 1 \cdot A \equiv A$.
   - **Complemento:** $A + \overline{A} \equiv 1$; $A \cdot \overline{A} \equiv 0$.
   - **Idempotencia:** $A + A \equiv A$; $A \cdot A \equiv A$.
   - **Distributiva:** $A + (BC) \equiv (A + B) \cdot (A + C)$; $A(B + C) \equiv AB + AC$
3. **Simplificación paso a paso:** Se procede aplicando las leyes que claramente reduzcan el número de términos o la cantidad de variables en cada término. Se pueden localizar los términos que tengan expresiones comunes que se puedan factorizar. Usualmente se puede aplicar la ley de la absorción $[(A + A \cdot B \equiv A)]$ para eliminar términos redundantes. También, al aplicar las leyes de De Morgan se obtienen suma de productos por producto de sumas, o al contrario, con el fin de poder aplicar alguna otra regla que pueda reducir la expresión.

10.3.2 Mapas de Karnaugh

Los mapas de Karnaugh son herramientas esenciales en la lógica digital y el diseño de circuitos. Ideados por Maurice Karnaugh, un científico e ingeniero que trabajaba en los Laboratorios Bell. En 1953, Maurice Karnaugh introdujo este método gráfico para simplificar las expresiones booleanas y así facilitar el diseño de circuitos digitales [Kar53].

Estos mapas permiten identificar las simplificaciones mediante la agrupación de términos adyacentes en una tabla bidimensional o, en casos de más de cuatro variables, una forma multidimensional. Este método se basa utiliza el álgebra booleana para simplificar las expresiones, minimizando el número de términos y, por tanto, la complejidad de los circuitos lógicos resultantes. Este método se basa en los siguientes conceptos:

Representación de Funciones Booleanas: Los mapas de Karnaugh permiten representar funciones booleanas de forma visual. Cada celda del mapa representa una combinación de valores de entrada [variables], y el valor dentro de la celda representa el resultado de la función para esa combinación de entrada.

Agrupación de Términos: La clave del método de Karnaugh es la agrupación de celdas adyacentes que contienen unos [o ceros, dependiendo de si se está minimizando la función o su complemento]. Estas agrupaciones representan términos simplificados de la función original. La adyacencia incluye el borde del mapa, donde las celdas se consideran adyacentes en un sentido toroidal, reflejando las propiedades del álgebra de Boole.

Minimización: Las agrupaciones de celdas se traducen de nuevo a expresiones booleanas simplificadas. La simplificación se basa en las leyes del álgebra de Boole, como la ley de la idempotencia, la ley de la absorción, y otras. El objetivo es reducir el número de términos y de variables en cada término al mínimo posible.

Principio de Dualidad: Los mapas de Karnaugh utilizan el principio de dualidad del álgebra de Boole, que establece que todas las operaciones lógicas tienen una

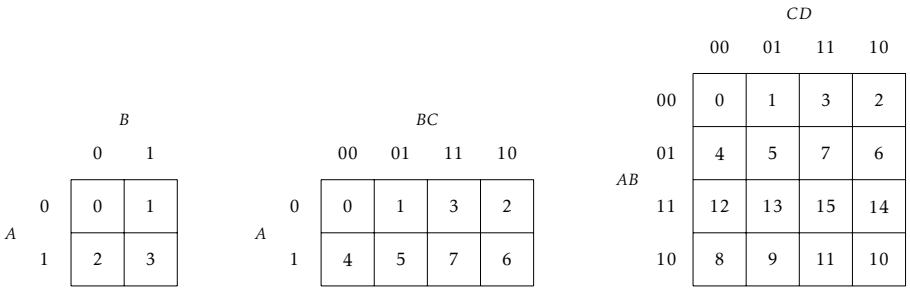


Figura 10.2: *Mapas de Karnaugh para 2, 3 y 4 variables. Dentro de las casillas se ha colocado el número en decimal de la construcción binaria hecha con los encabezados de fila y columna.*

operación dual que se puede obtener invirtiendo los operadores y los valores de identidad. Esto permite trabajar tanto con la suma de productos como con el producto de sumas de una manera eficiente.

Para representar una función lógica en un mapa de Karnaugh se pueden seguir los siguientes pasos:

1. *Preparar el mapa adecuado*. La aridad [ver la página 96] determina el número de variables que utiliza. El tamaño del mapa de Karnaugh depende de la aridad ya que habrán 2^n celdas para una función de aridad n.
   El mapa que se debe dibujar es una matriz en la que cada fila/columna representan diferentes combinaciones de variables de entrada. Cada fila o columna varía un solo bit de la fila/columna adyacente. En la figura 10.2 se muestran los mapas de Karnaugh para 2, 3 y 4 variables.
2. *Llenar el mapa con los valores de la función*. Considerando los minitérminos, que son aquellas combinaciones de valores de las variables en donde la función es 1, se coloca también un 1 en la casilla correspondiente, por ejemplo al considerar un mapa de 4 variables [que genera una matriz 4x4 de 16 casillas], si $\overline{A}BC\overline{D}$ es un minitérmino, entonces habrá un 1 en el renglón donde AB vale 01 y la columna CD vale 10.
   Por ejemplo, si la función es

$$F(A,B,C,D) \leftrightarrow \lambda A,B,C,D \bullet (\overline{A}B\overline{C}\overline{D} + AB\overline{C}\overline{D} + \overline{A}BC\overline{D} + ABC\overline{D} + \overline{A}\overline{B}CD + \overline{A}BCD),$$

podemos ver que los minitérminos traducidos a 1's y 0's respectivamente son:

$$0100, 1100, 0101, 1110, 0011, 0111.$$

Eso hace que el mapa de Karnaugh sea como el siguiente:

AB \ CD	00	01	11	10
00	·	·	1	·
01	1	·	1	1
11	1	·	·	1
10	·	·	·	·

③ *Detectar implicantes*. Se identifican y agrupan los unos adyacentes en el mapa [los implicantes]. Estas agrupaciones deben formarse con 1, 2, 4, etc., celdas, siguiendo potencias de 2, para garantizar la simplificación máxima. Como los bordes del mapa están conectados, las celdas de un borde conectan con el borde opuesto.

AB \ CD	00	01	11	10
00	·	·	1	·
01	1	·	1	1
11	1	·	·	1
10	·	·	·	·

④ *Obtener las expresiones simplificadas*.

- Para cada grupo, identifica las variables que no cambian dentro del grupo. Estas son las que definen un término en la expresión simplificada.
- Escribe un término para cada grupo, donde las variables que permanecen constantes representan el término en su forma directa [si es un 1] o complementada [si es un 0].
- Combina todos los términos mediante operaciones OR para obtener la función simplificada.

Se lograron dos grupos, uno de cuatro unos y el otro de dos. El grupo de cuatro lo constituyen los términos $\overline{A}B\overline{C}\overline{D}, AB\overline{C}\overline{D}, \overline{A}BC\overline{D}$ y $ABC\overline{D}$ y el grupo de dos lo conforman los términos $\overline{A}\overline{B}CD$ y $\overline{A}BCD$.
Para el grupo de cuatro, los valores fijos son $B\overline{D}$, y para el grupo de dos, lo que no cambia es $\overline{A}CD$. De modo que la función reducida es:

$$F^{\mathrm{R}}(A,B,C,D) \leftarrow\!\shortmid \lambda A,B,C,D \bullet (B\overline{D} + \overline{A}CD)$$

⑤ *Verificar la función simplificada*. Una vez terminado el proceso, es conveniente verificar que la función original y la función simplificada son lógicamente equivalentes [ver la página 110].
Para verificar la función reducida F^{R} podemos utilizar la computadora con la herramienta TV [ver página 134] para generar las tablas de verdad y comparar las respectivas columnas.

```
>>> F = lambda a,b,c,d:O(Y(neg(a),b,neg(c),neg(d)), Y(a,b,neg(c),neg(d)), Y(
    neg(a),b,c,neg(d)), Y(a,b,c,neg(d)), Y(neg(a),neg(b),c,d), Y(neg(a),b,c,d
    ))
>>> TV(F)
    a     |    b     |    c     |    d     |F(a,b,c,d)
----------+----------+----------+----------+----------
   True   |   True   |   True   |   True   |  False
   True   |   True   |   True   |  False   |   True
   True   |   True   |  False   |   True   |  False
   True   |   True   |  False   |  False   |   True
   True   |  False   |   True   |   True   |  False
   True   |  False   |   True   |  False   |  False
   True   |  False   |  False   |   True   |  False
   True   |  False   |  False   |  False   |  False
  False   |   True   |   True   |   True   |   True
  False   |   True   |   True   |  False   |   True
  False   |   True   |  False   |   True   |  False
  False   |   True   |  False   |  False   |   True
  False   |  False   |   True   |   True   |   True
  False   |  False   |   True   |  False   |  False
  False   |  False   |  False   |   True   |  False
  False   |  False   |  False   |  False   |  False
>>>
```

y para el caso de la función reducida:

```
>>> >>> Fred = lambda a,b,c,d:O(Y(b,neg(d)),Y(neg(a),c,d))
>>> TV(Fred)
    a     |    b     |    c     |    d     |Fred(a,b,c,d)
----------+----------+----------+----------+----------
   True   |   True   |   True   |   True   |  False
   True   |   True   |   True   |  False   |   True
   True   |   True   |  False   |   True   |  False
   True   |   True   |  False   |  False   |   True
   True   |  False   |   True   |   True   |  False
   True   |  False   |   True   |  False   |  False
   True   |  False   |  False   |   True   |  False
   True   |  False   |  False   |  False   |  False
  False   |   True   |   True   |   True   |   True
  False   |   True   |   True   |  False   |   True
  False   |   True   |  False   |   True   |  False
  False   |   True   |  False   |  False   |   True
  False   |  False   |   True   |   True   |   True
  False   |  False   |   True   |  False   |  False
  False   |  False   |  False   |   True   |  False
  False   |  False   |  False   |  False   |  False
>>>
```

10.3.3 Compuertas lógicas en la práctica

Considera un sistema de control de iluminación *inteligente* para una habitación que tiene dos modos de operación: uno basado en las condiciones de luz externas y el otro modo basado en las preferencias del usuario. El sistema tiene tres entradas:

Entrada A: Es un sensor de luminosidad externa. Cuando $A \equiv 1$ significa que en el exterior hay suficiente luz. Si $A \equiv 0$ entonces afuera está oscuro.

Entrada B: Representa la preferencia del usuario. Si $B \equiv 1$ significa que el usuario activa la luz artificial; pero si $B \equiv 0$ quiere decir que se desactiva la luz artificial.

Entrada C: Es un sensor de presencia en la habitación. Cuando $C \equiv 1$ es que hay alguien en la habitación; mientras que cuando $C \equiv 0$ no hay nadie.

La habitación debe estar iluminada artificialmente en las siguientes condiciones:

ⓘ Cuando está oscuro afuera y hay alguien en la habitación, sin importar la preferencia del usuario sobre la luz artificial.

(ii) Cuando el usuario prefiere la luz artificial y hay alguien en la habitación, sin importar si afuera hay suficiente luz.

(iii) Si afuera hay suficiente luz, el usuario prefiere la luz artificial, y hay alguien en la habitación.

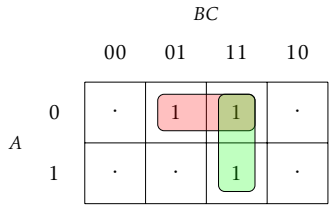
La tabla de verdad muestra el comportamiento general:

Condición	A	B	C	$F(A,B,C)$
ii, iii	1	1	1	1
	1	1	0	0
	1	0	1	0
	1	0	0	0
i, ii	0	1	1	1
	0	1	0	0
i	0	0	1	1
	0	0	0	0

La tabla de verdad permite crear la función lógica:

$$\lambda A,B,C \bullet (ABC + \overline{A}BC + \overline{A}\overline{B}C)$$

Al hacer el mapa de Karnaugh para tres variables y agrupando los unos:



Al considerar cada uno de los implicantes [grupos de unos], se obtiene la función:

$$\lambda A,B,C \bullet (\overline{A}C + BC)$$

Pero se observa que ambos términos comparten el término C, por lo que la función aún se puede reducir considerando C como factor común:

$$\lambda A,B,C \bullet [C(\overline{A} + B)]$$

Al verificar la función se corrobora que tiene el mismo comportamiento que la tabla de verdad deseada.

```
>>> F = lambda A,B,C:Y(C, O(neg(A),B)
>>> TV(F)
    A      |    B     |    C     | F(A,B,C)
----------+----------+----------+----------
   True    |   True   |   True   |   True
   True    |   True   |  False   |  False
   True    |  False   |   True   |  False
   True    |  False   |  False   |  False
  False    |   True   |   True   |   True
  False    |   True   |  False   |  False
  False    |  False   |   True   |   True
  False    |  False   |  False   |  False
>>>
```

Con la función reducida podemos dibujar un diagrama de circuitos digitales.

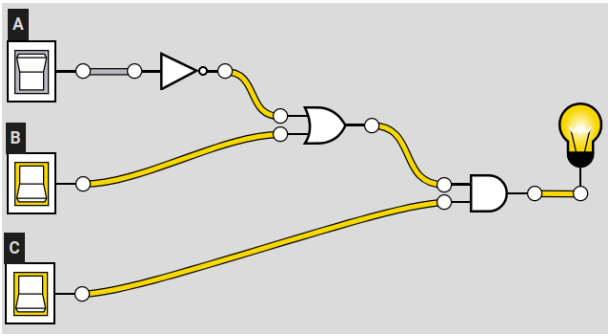


Figura 10.3: *Circuito digital de la función* $\lambda A, B, C \bullet Y(C, O(neg(A), B)$. *Se muestra el caso* $(\lambda A, B, C : Y(C, O(neg(A), B))(\boldsymbol{F}, \boldsymbol{V}, \boldsymbol{V})$. Circuito creado en `logic.ly`

Ejercicios

1. Convierte a binario los siguientes números decimales:
   a) *Ejemplo.* Convierte el número decimal 45 a binario. Para convertir el número decimal 45 a binario, se debe dividir el número entre 2. El cociente se toma como nuevo número y el residuo se guarda para construir el número decimal. Continúa dividiendo cada cociente entre 2 y cada residuo se conserva, hasta que el cociente sea 0.

Número	Cociente [÷2]	Residuo [÷2]
45	22	1
22	11	0
11	5	1
5	2	1
2	1	0
1	0	1

   Así el número binario se construye colocando cada residuo desde el último al primero: 101101_2.
   b) Convierte el número decimal 373 a binario.
   c) Convierte el número decimal 2947 a binario.
2. Convierte los siguientes números binarios a decimal.
   a) *Ejemplo.* Convierte el 101001_2 a decimal. Para convertir un número binario a decimal, se debe obtener la suma de las potencias de 2 de los 1's, considerando la potencia de acuerdo a la posición [de derecha a izquierda iniciando en 0]. Así 101001 se convierte al obtener:

$$1 \times 2^5 + 1 \times 2^3 + 1 \times 2^0 = 32 + 8 + 1 \mapsto 41$$

   b) Convierte el número binario 101100 a decimal.
   c) Convierte el número binario 110011 a decimal.
3. Describe formalmente los siguientes problemas dados en lenguaje natural.
   a) *Ejemplo.* **Sistema de control de acceso**. Un edificio requiere un sistema de control de acceso que se activa mediante un código binario de 3 bits. El acceso se concede solo si el código cumple con ciertas condiciones específicas: El primer número binario es 1, o el segundo y tercer bits son ambos 0.
   Si se determinan 3 variables A, B y C para cada bit, A el primero, B el segundo y C el tercero, las condiciones se cumplen cuando $A \equiv 1 \vee (B \equiv 0 \wedge C \equiv 0)$, lo que se expresa como $A + \overline{B}\overline{C}$.
   b) **Detección de paridad**. Una empresa necesita verificar la paridad de los datos transmitidos en un sistema de 3 bits para detectar errores básicos en la transmisión. Se requiere un sistema que determine si el número de bits con valor 1 en la entrada de 3 bits es par o impar. Si es par, la salida debe ser 1; de lo contrario, debe ser 0.
   c) **Sistema contra intrusos**. Un sistema de alarma doméstica contra intrusos se configura con sensores en puertas y ventanas. La alarma se activa si al menos tres de los cuatro sensores detectan una apertura. Si dos o menos sensores se activan, la alarma permanece desactivada.

d) **Control de iluminación**. Un sistema de control de iluminación *inteligente* para una sala de conferencias utiliza 4 interruptores. Las luces deben encenderse solo bajo las siguientes condiciones: todos los interruptores están apagados, o exactamente dos interruptores están encendidos. En cualquier otro caso, las luces deben permanecer apagadas.

4. Especifica la tabla de verdad de los problemas de diseño digital de los ejercicios del punto 3.
   a) *Ejemplo.* Para el **sistema de control de acceso**, la tabla de verdad de la función $\lambda A, B, C \bullet (A + \overline{B}\overline{C})$ es la siguiente:

A	B	C	$F(A,B,C)$
1	1	1	1
1	1	0	1
1	0	1	1
1	0	0	1
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	1

   b) Determina la tabla de verdad para el problema de la **detección de paridad**.
   c) Determina la tabla de verdad para el problema del **sistema contra intrusos**.
   d) Determina la tabla de verdad para el problema del **control de iluminación**.
5. Escribe la función de minitérminos en forma de suma de productos.
   a) *Ejemplo.* Reescribe la función $\lambda a, b, c \bullet \sum m(0, 5, 6)$ a una forma de suma de productos. Los números 0, 5, 6 son las expresiones en decimal de los números binarios que representan el estado **V** o **F** de cada variable, en el orden en que aparecen.

d_{10}	a	b	c	Producto
0	0	0	0	$\bar{a}\bar{b}\bar{c}$
5	1	0	1	$a\bar{b}c$
6	1	1	0	$ab\bar{c}$

   Así la función en la forma de suma de productos es $\lambda a, b, c \bullet (\bar{a}\bar{b}\bar{c} + a\bar{b}c + ab\bar{c})$.
   b) Reescribe $\lambda w, x, y, z \bullet \sum m(1, 3, 5, 6, 12)$ como suma de productos.
   c) Reescribe $\lambda p, q, r, s \bullet \sum m(2, 4, 6, 9, 14, 15)$ como suma de productos.
   d) Reescribe $\lambda a, b, c \bullet \sum m(1, 2, 5, 6, 7)$ como suma de productos.
6. Aplica las reglas del álgebra booleana para reducir expresiones.
   a) *Ejemplo.* Reduce la expresión $A\overline{B}C + \overline{A}B\overline{C} + ABC$.
   Podemos ver que los términos $A\overline{B}C$ y ABC tienen en común AC, por lo que se puede factorizar de esos términos:

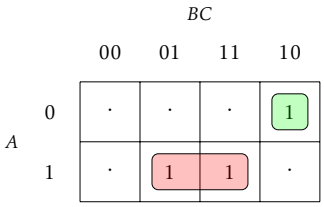
$$AC(\overline{B} + B) + \overline{A}B\overline{C}$$

   Sabiendo que $B + \overline{B} = 1$ porque siempre será verdadero [la suma de una variable con su complemento siempre es 1], podemos simplificar un poco más.

$$AC + \overline{A}B\overline{C}.$$

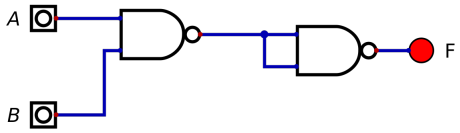
b) Reduce la expresión $AB\overline{C}D + \overline{A}BC\overline{D} + A\overline{B}\overline{C}D + \overline{A}\overline{B}\overline{C}D + AB\overline{C}\overline{D} + ABCD$.
c) Reduce la expresión $ABC + \overline{A}\overline{B}C + A\overline{B}\overline{C} + \overline{A}B\overline{C}$.

7. Utiliza mapas de Karnaugh para reducir expresiones:
   a) *Ejemplo.* Reduce la expresión $A\overline{B}C + \overline{A}B\overline{C} + ABC$.
   Como la expresión tiene tres variables el mapa ya con los implicantes es:



   Lo que no cambia en el implicante de dos unos es AC y en el implicante de 1 solo uno no hay nada que reducir $\overline{A}B\overline{C}$, por lo que la simplificación es la suma $AC + \overline{A}B\overline{C}$.
   b) Reduce la expresión $AB\overline{C}D + \overline{A}BC\overline{D} + A\overline{B}\overline{C}D + \overline{A}\overline{B}\overline{C}D + AB\overline{C}\overline{D} + ABCD$.
   c) Reduce la expresión $ABC + \overline{A}\overline{B}C + A\overline{B}\overline{C} + \overline{A}B\overline{C}$.

8. Diseño de circuitos.
   a) *Ejemplo.* Diseña un circuito usando solo compuertas NAND que realice la función de una compuerta AND. Explica el funcionamiento del circuito.
   Para lograr el comportamiento de AND con compuertas NAND, primero observamos que una compuerta NAND tiene el efecto 0 cuando ambas entradas son 1 y en los demás casos la salida es 1. Conectando la salida de la primera NAND a ambas entradas de una segunda NAND tiene el efecto de inversor, como se muestra en el siguiente diagrama:



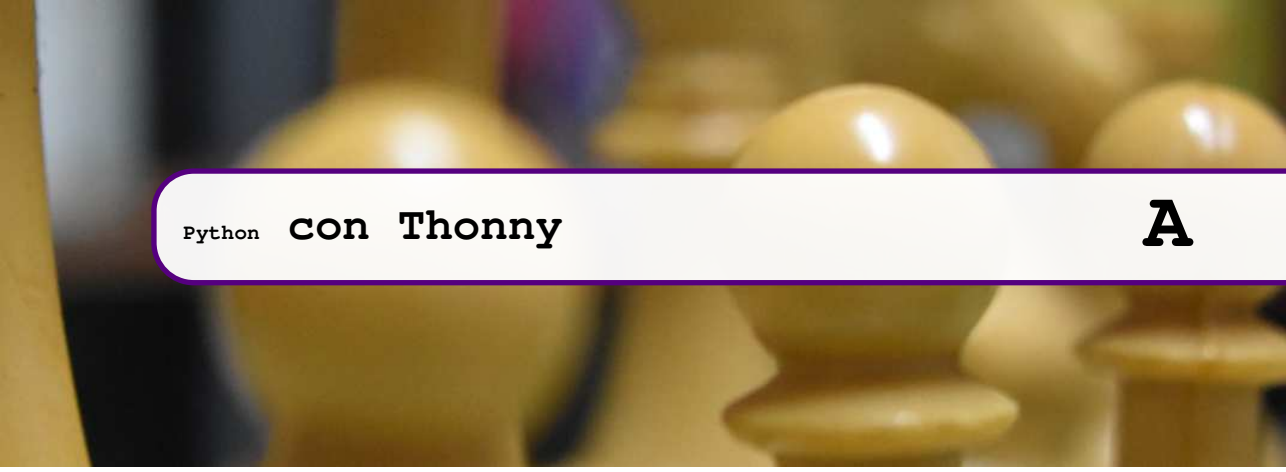
   La tabla de verdad muestra que esta construcción es equivalente a la compuerta AND:

A	B	$\text{AND}(A,B)$	$\text{NAND}(\text{NAND}(A,B),\text{NAND}(A,B))$
1	1	1	1
1	0	0	0
0	1	0	0
0	0	0	0

   b) Diseña un circuito que utilice compuertas XOR para comparar dos bits de entrada y producir un 1 si son diferentes y un 0 si son iguales.
   c) Diseña un circuito que se active cuando exactamente 2 de los cuatro sensores de entrada se activen.

IV

Apéndices



Python con Thonny

A

El código fuente que se encuentra en este libro de lógica computacional se ha escrito en el lenguaje de programación `Python`. Pero con un poco de paciencia se pueden escribir para casi cualquier otro lenguaje de programación.

Los lenguajes de programación son muchos y muy diversos. Los lenguajes de programación se pueden clasificar en dos grandes grupos:

1. Lenguajes de propósito general como `Racket`, `C`, `Python`, `Pascal`, etc., son aquellos lenguajes que puedes ser utilizados para resolver cualquier problema computable de cualquier área, es decir, aquellos problemas que se pueden resolver mediante un algoritmo.
2. Lenguajes de propósito específico como `Mathematica`, `SPSS`, `Octave`, `Prolog`, `SQL`, etc., son aquellos que se especializan en resolver problemas de un ámbito muy reducido. Estos lenguajes a menudo proporcionan abstracciones y constructos que son altamente optimizados para su dominio particular, pero pueden ser limitados o menos eficientes para tareas fuera de ese dominio.

En este libro se utiliza `Python` como lenguaje de programación base, ya que se presta muy bien para construir razonamientos lógicos, es de fácil aprendizaje y como tiene relativamente pocas palabras clave y su sintaxis es muy sencilla, se puede dedicar más tiempo en el desarrollo lógico que en resolver problemas propios del uso del lenguaje, como corregir los errores de sintaxis.

Este libro utiliza la versión 3.9.16 de `Python`, pero a partir de la versión 3.4, las expresiones del lenguaje no varían mucho, y para los objetivos de este libro no hay diferencia en el uso de cualquier versión de `Pythonv3.4` y posteriores.

Para utilizar `Python` en la computadora se requiere de manera obligatoria el intérprete de `Python` y de manera opcional un `IDE`, que es un programa que facilita la

escritura del código fuente de Python y la interacción del programador con el sistema. El IDE que utilizaremos es Thonny, que es un programa con dos ventajas [a mi parecer] sobre otros IDEs:

1. Ya incluye un intérprete de Python, por lo que no será necesario instalar Python por un lado y el IDE por otro.
2. Tiene muy pocas funcionalidades en comparación con otros, lo que permite que el programa ocupe poca memoria en la computadora y se aprenda a utilizar rápido. Entre esas funcionalidades, Thonny tiene una manera de seguir el rastro del programa paso a paso, que permite descubrir errores y ver qué sucede con la información en cada momento.

Thonny puede ser obtenido de manera gratuita desde su sitio web en

https://thonny.org/

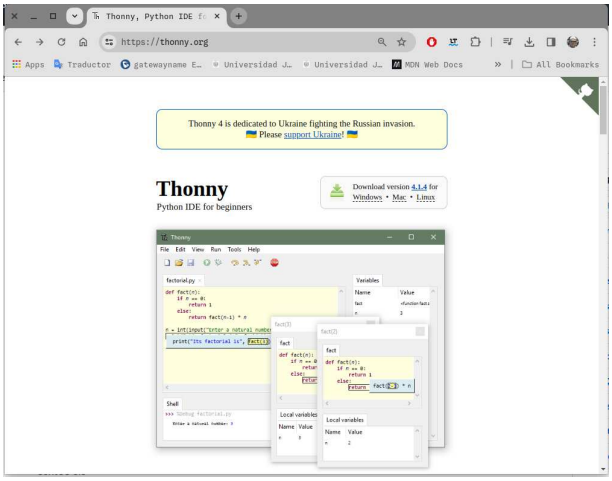


Figura A.1: *Thonny. Un IDE para Python pensado para principiantes.*

A la fecha, la versión de Thonny que se instala es la versión 4.1.4. Las instrucciones para instalar el programa se encuentran también en el mismo sitio. Las versiones para Windows y MacOS tienen instaladores que son fáciles de seguir. La versión para Linux también es simple, pues hay que correr un comando en la terminal, siempre que se tenga acceso a Internet. Sin embargo, aquí algunas notas acerca de la instalación:

- Instalación en Windows:
  - Hay cinco opciones para instalar en Windows:
    1. Windows en sistemas de 64 bits.
    2. Windows en sistemas de 32 bits.
    3. Versión portable en sistemas de 64 bits.
    4. Versión portable en sistemas de 32 bits.
    5. Cuando ya hay una instalación de Python.

    En las primeras dos opciones se descarga un archivo ejecutable [.exe]; en las siguientes dos opciones se descarga un paquete [.zip] que contiene los

archivos necesarios; y en la última opción se debe ejecutar el comando `pip install thonny` desde una terminal de comandos.

- Instalación en MacOS:
  - Hay al menos una advertencia en la instalación de `Thonny` en sistemas con MacOS 10.15 Catalina, particularmente dirigida a aquellos usuarios que han actualizado su sistema, ya que al parecer, esa nueva versión ha cambiado la manera en que se otorgan permisos para acceder a ciertos lugares del árbol de directorios. En https://github.com/thonny/thonny/wiki/MacOSX se encuentran las instrucciones sobre qué hacer en ese caso.
- Instalación en Linux. El caso de Linux siempre es particular, pues generalmente no hay una única manera de instalar programas, pero en general, `Thonny` y todos los programas pueden instalar de dos maneras:

  ① *Instalación con el instalador oficial:* Aquí de nuevo hay dos opciones,

  (a) Con el instalador oficial. El siguiente comando descarga e instala `Thonny` junto con una versión privada de `Python` 3.10 en sistemas `x86_64` o usa la versión de `Python` existente:

  ```
  bash <(wget -O - https://thonny.org/installer-for-linux)
  ```

  (b) Si ya eres experto en Linux, puedes intentar la instalación mediante `pip`, si prefieres reutilizar una instalación existente de Python:

  ```
  pip3 install thonny
  ```

  ② *Instalación desde repositorios de la distribución:* Aquí hay varias opciones dependiendo de la distribución que estés usando.

  - *Debian, Ubuntu, Mint y derivados:*

  ```
  sudo apt install thonny
  ```

  - *Fedora:*

  ```
  sudo dnf install thonny
  ```

  - *Flatpack:*

  ```
  flatpak install org.thonny.Thonny
  ```

  - *Snap:*

  ```
  sudo snap install thonny
  ```

Cada método garantiza la instalación de `Thonny`, aunque las versiones pueden variar según la fuente utilizada.

A.1 IDE de Thonny

La ventana principal de `Python` se divide principalmente en dos: la parte superior es la sección de definiciones y la inferior es la sección de interacciones [figura A.2].

- La sección de definiciones sirve para escribir el código fuente de los programas que quedarán grabados. Para que `Python` tome en cuenta las definiciones hechas, debes cargar el programa y ejecutarlo, para que `Python` verifique que no tiene errores e inicie la interpretación del código.

- La sección de interacciones sirve para interactuar con Python, invocando los conceptos definidos. En ocasiones se dice que la sección de interacciones funciona como una calculadora, ya que se ingresan las operaciones y al aceptar con ENTER, el sistema evalúa la solicitud y emite su respuesta.

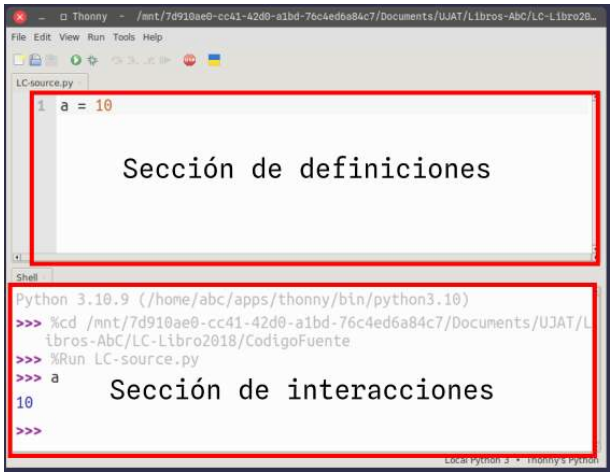


Figura A.2: El IDE de Python está dividido en dos partes: La sección en donde se escriben las definiciones en la parte superior y la sección donde se interactúa con el programa en la parte inferior.

Una interacción en Python es como solicitar a Python el significado o el valor de algo.

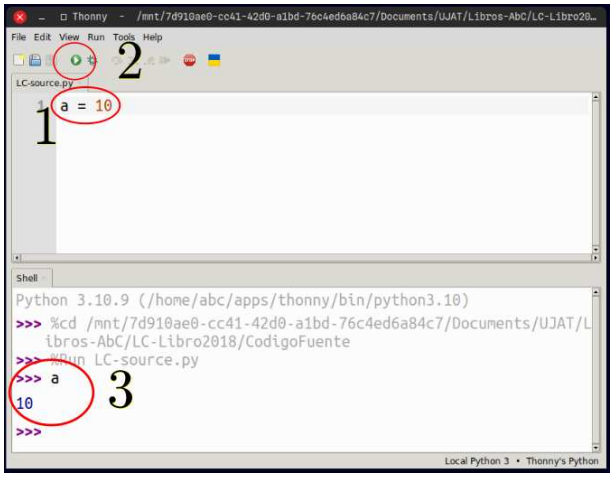


Figura A.3: Para usar el IDE de Python, usualmente primero se escribe el código [1] en la sección de definiciones, luego se analiza y ejecuta [2] el código escrito y finalmente se prueba [3] el código desde la sección de interacciones.

El uso de Thonny consiste en 3 pasos [figura A.3]:

① El código fuente se escribe en la sección de «definiciones», lo que se escribe en esa parte se puede grabar en un archivo de texto con la extensión .py que es relativa al nombre del lenguaje.

② Para que el sistema pueda utilizar las instrucciones escritas en la parte de definiciones, es necesario ejecutar el programa. Esto puede hacerse activando el ícono RUN, en la parte superior de la ventana principal. Al activar la ejecución del código, Python lee y analiza el código fuente informando de cualquier error que hubiera; de no haber error, en la parte de interacciones aparece un nuevo prompt [>>>] que indica que el sistema tiene todas las definiciones cargadas en memoria y está listo para recibir consultas.

③ Ya con las definiciones cargadas en la memoria, las interacciones se escriben en donde señala el prompt, señalado con [>>>] en la parte de interacciones.
Una vez escrita la consulta, se debe accionar la tecla ENTER o RET en el teclado de su computadora, a veces marcada con el símbolo [↵].
En ese momento, el sistema lee la entrada, la evalúa y finalmente imprime la salida justo debajo de la consulta. Estas tres acciones se repiten hasta que se hace algún cambio en el código y se ejecuta de nuevo el programa. El ciclo de leer-evaluar-imprimir se conoce como REPL, por sus siglas en inglés [*Read-Evaluate-Print-Loop*].

A.2 El modo depurador de Thonny

El modo depurador (debugger) en el IDE de Thonny para Python es una herramienta poderosa diseñada para ayudar a los desarrolladores a entender mejor su código, encontrar y corregir errores más eficientemente. El funcionamiento es como sigue:

1. *Inicio del modo depurador*: El modo depurador se inicia seleccionando la opción correspondiente en el menú o usando un atajo de teclado. Esto prepara el código para ser ejecutado paso a paso.
2. *Puntos de interrupción [*breakpoints*]*: Thonny permite colocar puntos de interrupción en las líneas de código donde desees que la ejecución se pause. Esto es útil para inspeccionar el estado de tu programa en momentos específicos.
3. *Ejecución* paso a paso: Una vez en modo depurador y con los puntos de interrupción establecidos, puedes ejecutar el código paso a paso. Thonny ofrece diferentes opciones para esto, como avanzar a la siguiente línea de código, entrar en funciones para ver su ejecución detalladamente, o saltar sobre ellas.
4. *Inspección de variables*: Mientras el código se ejecuta en modo depurador, Thonny muestra el valor actual de las variables en cada paso. Esto permite ver cómo cambian los valores a lo largo de la ejecución y puede ayudar a identificar dónde las cosas no van según lo previsto.
5. *Visualización de la pila de llamadas*: Thonny proporciona una vista de la pila de llamadas, mostrando qué funciones se han invocado y actualmente se encuentran en estado de ejecución. Esto es especialmente útil para entender el flujo de ejecución en programas con múltiples llamadas a funciones y métodos.
6. *Control de la ejecución*: Puedes controlar la ejecución del programa con botones que permiten continuar la ejecución hasta el siguiente punto de interrupción, salir del modo depurador, o detener la ejecución completamente.

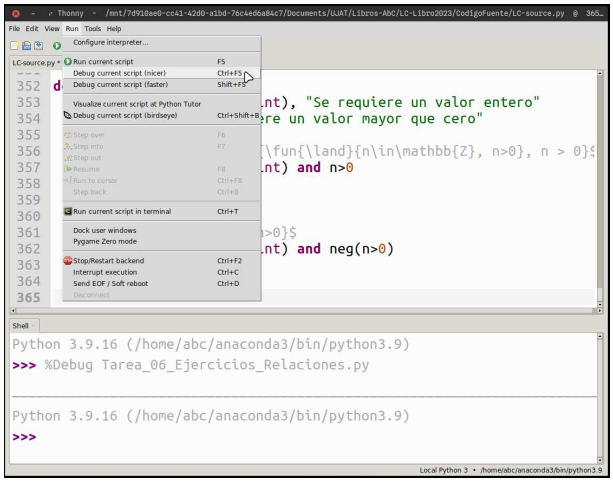


Figura A.4: *Inicialización del modo depurador. Opcionalmente se inicia activando el ícono con forma de insecto en el menú.*

7. *Mensajes y errores*: Si el programa genera errores, Thonny los muestra de manera que se pueda determinar con facilidad qué ha salido mal y en qué parte del código.

`Ventajas del modo depurador en Thonny`

Al ser `Thonny` un `IDE` diseñado con un enfoque en principiantes, su modo depurador es intuitivo y fácil de usar, lo que lo hace ideal para quienes están aprendiendo a programar. Mejora la eficiencia en la solución de problemas. Al permitirte ver exactamente cómo se ejecuta tu código y cómo cambian los datos, te ayuda a identificar y solucionar errores de manera más eficiente. Usar el modo depurador en Thonny te ofrece una manera interactiva y detallada de entender el comportamiento de tu programa, lo que lo convierte en una herramienta invaluable tanto para principiantes como para desarrolladores más experimentados.

Bibliografía

[AA06] Joaquín Aranda Almansa. *Fundamentos de Lógica Matemática y Computación*. Sanz y Torres, 2006.

[Arn89] José Antonio Arnaz. *Iniciación a la Lógica Simbólica*. Temas básicos: Metodología de la Ciencia. Trillas, tercera ed., 1989.

[BA09] Alfonso Bustamante Arias. *Lógica y Argumentación: De los Argumentos Inductivos a las Álgebras de Boole*. Pearson Educación México, 2009.

[Boc68] Jozef María Bocheńsky. *Historia de la Lógica Formal*. Gredos Ed., 1968.

[Boo48] George Boole. *The Mathematical Analysis of Logic: Being an Essay Towards A Calculus of Deductive Reasoning*. Philosophical Library Inc., 1948.

[BV90] Stephen Brown and Zvonko Vranesic. *Fundamentals of Digital Logic with VHDL Design*. McGraw-Hill, 3rd ed., 1990.

[CG15] Willem Conradie and Valentin Goranko. *Logic and Discrete Mathematics: A Concise Introduction*. Wiley & Sons, Limited, John, 2015.

[Chu32] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics, Second Series*, 33(2):346 – 366, 1932.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ, 1941.

[Chu56] Alonzo Church. *Introduction To Mathematical Logic*, volume 1. Princeton, New Jersey Princeton University Press, 1956.

[CN02] Luis Camacho Naranjo. *Introducción a la Lógica*. Libro Universitario Regional, 2002.

[Cur29] Haskell B. Curry. An Analysis of Logical Substitution. *American Journal of Mathematics*, 51:363, 1929.

[Cur34] Haskell B. Curry. Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, 1934.

[Cur77] Haskell B. Curry. *Foundations of Mathematical Logic*. Dover Publications, Incorporated, dover ed., 1977.

[Dav73] Martin Davis. Hilbert's Tenth Problem is Unsolvable. *The American Mathematical Monthly*, 80(3):233–269, mar 1973.

[DG06] María Noemí Domínguez García. *Conectores Discursivos en Textos Argumentativos Breves*. Arco Libros - La Muralla, S.L., 2006.

[DM47] Augustus De Morgan. *Formal Logic: The Calculus of Inference, Necessary and Probable*. Taylor & Walton Eds., 1947.

[DO82] Ezequiel De Olaso, editor. *G. W. Leibniz, Escritos Filosóficos*. Biblioteca de Filosofía. Charcas Ed., 1982.

[DP17] Vilnis Detlovs and Karlis Podnieks. *Introduction to Mathematical Logic*. OpenLibra, 2017.

[FdC+04] Max Fernández de Castro et al. *Lógica Elemental*. Universidad Autónoma Metropolitana. U. Iztapalapa, México, 2004.

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. In American Mathematical Society Jacob T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume XIX, pages 19–32. American Mathematical Society, Providence, Rhode Island, 1967.

[FR09] Catalina Fuentes Rodríguez. *Diccionario de Conectores y Operadores del Español*. Editorial Arco Libros, 2009.

[GBB05] Patrick Grässle, Henriette Baumann, and Philippe Baumann. *UML 2.0 in Action: A Project Based Tutorial. A Detailed and Practical Walk-Through Showing how to apply UML to Real Worl Development Projects*. PACKT Publishing, September 2005.

[Gen33] Gerhard Gentzen. Über die Existenz unabhängiger Axiomensysteme zu unendlichen Satzsystemen. *Mathematische Annalen*, 107(1):329–350, 1933.

[Gen64] Gerhard Gentzen. Investigations into Logical Deduction. *American Philosophical Quarterly*, 1(4):288–306, 1964.

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[HH12] David Money Harris and Sarah L. Harris. *Digital Design and Computer Architecture*. Elsevier, 2nd ed., 2012.

[Hil02] David Hilbert. Mathematical Problems. *Bulletin of the American Mathematical Society*, 8(10):437 – 479, 1902.

[Hoa69] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, page 576–580, 1969.

[Jen91] Kathleen Jensen. *Pascal User Manual and Report*. Springer New York, 4 ed., 1991. Description based on publisher supplied metadata and other sources.

[Kar53] Maurice Karnaugh. The Map Method for Synthesis of Combinational Logic Circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.

[KB05] Randy H. Katz and Gaetano Borriello. *Contemporary Logic Design*. Pearson, 2nd ed., 2005.

[Kle36] Stephen Cole Kleene. λ-Definability and Recursiveness. *Duke Mathematical Journal*, 2(2):340 – 353, 1936.

[Kle81] Stephen Cole Kleene. The Theory of Recursive Functions, Approaching its Centennial. *Bulletin (New Series) of the American Mathematical Society*, 5(1):43 – 61, July 1981.

[Log15] Logo Foundation. Logo history. *Logo Foundation Web Page* https://el.media.mit.edu/logo-foundation/what_is_logo/history.html, 2015.

[Mac99] Bruce J. MacLennan. *Principles of programming languages. Design, evaluation, and implementation*. Oxford University Press, 3 ed., 1999. Includes bibliographical references and index.

[Mat93] Jurij V. Matijasevič. *Hilbert's Tenth Problem*. Foundations of Computing. The MIT Press, 3 ed., 1993. Aus dem Russ. übers.

[MC07] M. Morris Mano and Michael D. Ciletti. *Digital Design*. Pearson Prentice Hall, 4th ed., 2007.

[Mey90] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice-hall International Series in Computer Science. Prentice Hall, 1990.

[MN88] Andrei Andreevich Markov and Nikolai Makarovich Nagorny. *The Theory of Algorithms*. Number 23 in Mathematics and Its Applications. Kluwer Academic Publishers, 1988.

[MnR18] Dora Luz Muñoz Rincón. ¿Qué son, cuál es el uso y cómo se clasifican los conectores? *Palabras en Orden (Blog)* https://www.upb.edu.co/es/central-blogs/ortografia/como-se-clasifican-conectores, 2018.

[Nau66] Peter Naur. Proof of Algorithms by General Snapshots. *BIT*, 6:310–316, 1966.

[New00] Monty Newborn. *Automated Theorem Proving*. Springer, 2000.

[OMG17] OMG. *Unified Modeling Language (UML), Version 2.5.1*. Object Management Group, 2017.

[Pla18] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer International Publishing, 2018. Literaturangaben.

[RAE10] RAE. *Manual de la Nueva Gramática de la lengua española*. Nuevas Obras Real Academia. Espasa, 2010.

[RCS+22] Markus Roggenbach, Antonio Cerone, Bernd-Holger Schlingloff, Gerardo Schneider, and Siraj Ahmed Shaikh. *Formal Methods for Software Engineering : Languages, Methods, Application Domains*. Springer, 2022.

[Rit96] Dennis M. Ritchie. *The Development of the C Programming Language*, page 671–698. Association for Computing Machinery, New York, NY, USA, 1996.

[RK04] Charles H. Roth and Larry L. Kinney. *Fundamentals of Logic Design*. Thomson Learning, 5th ed., 2004.

[RND17] Mohit Raj and Bhaskar N. Das. *Learn Python in 7 days*. Packt Publishing, 2017.

[Ros04] Kenneth H. Rosen. *Matemática Discreta y sus Aplicaciones*. McGraw-Hill/Interamericana de España, S.A. U., quinta ed., 2004.

[RW94] Martin Reiser and Niklaus Wirth. *Programming in Oberon*. ACM Pr. [u.a.], reprinted ed., 1994. Literaturangaben.

[SH86] Patrick Suppes and Shirley Hill. *Introducción a la Lógica Matemática*. Editorial Reverté, 1986.

[SHK+20] Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark Miller, Margaret Minsky, Artemis Papert, and Brian Silverman. History of logo. *Proceedings of the ACM on Programming Languages*, 4:1–66, 06 2020.

[Sim21] Peter Simons. Jan Łukasiewicz >Łukasiewicz's Parenthesis-Free or Polish Notation. On line entry at https://plato.stanford.edu/entries/lukasiewicz/polish-notation.html, 2021.

[SK95] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages. A Laboratory Based Approach*. Addison-Wesley Publishing Company, Inc., 1995. Literaturverz. S. 611 - 624.

[Tho72] Ken Thompson. Users' Reference to B. *Bell Telephone Laboratories*, 39199(11), 1972.

[TMRP00] Óscar Trelles Montero and Diógenes Rosales Papa. *Introducción a la Lógica*. Pontificia Universidad Católica del Perú. Fondo Editorial 2000, 2 ed., 2000.

[Tur36] Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Tur50] Alan Mathison Turing. Computing Machinery and Intelligence. *Mind*, 59(236):433–460, 1950.

[VN58] John Von Neumman. *The Computer and the Brain*. New Haven: Yale University Press, 1958.

[Wak06] John F. Wakerly. *Digital Design: Principles and Practices*. Prentice Hall, 4th ed., 2006.

[Wie61] Norbert Wiener. *Cybernetics: Or Control and Communication in the Animal and the Machine*. Number 25 in MIT Paperback series. MIT University Press, Cambridge, MA, second ed., 1961.

[Wir71] Niclaus Wirth. The Programming Language Pascal. *Acta Informatica*, 1(1):35–63, mar 1971.

[WR63] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge at the University Press, 1963.

[WTT+17] Karn Wijarnpreecha, Charat Thongprayoon, Natanong Thamcharoen, Panadeekarn Panjawatanan, and Wisit Cheungpasitporn. Association of Coffee Consumption and Chronic Kidney Disease: A Meta-analysis. *WILEY International Journal of Clinical Practice*, 71(e12919):e12919, 2017.

[ZM16] Martín Zubiria and Juan José Moral, editors. *El poema doctrinal de Paraménides*. Universidad Nacional de Cuyo, 2016. Libro digital.

Índice alfabético

Símbolos

G

H

I

J

K

L

M

N

O

P

R

S

T

V

W

Y

Wilfrido Miguel Contreras Sánchez
Secretario de Investigación, Posgrado y Vinculación

Pablo Marín Olán
Director de Difusión, Divulgación Científica y Tecnológica

Analuisa Kú Ortíz
Jefa del Departamento Editorial de Publicaciones No Periódicas

Lógica Computacional:
Una perspectiva funcional con Python

Lógica Computacional es una disciplina obligatoria en las Ciencias Computacionales. En este libro se ofrece un amplio conjunto de definiciones que cubren diversos temas de cálculo proposicional; cómputo de predicados y algunas de las aplicaciones de la lógica computacional. Todos los conceptos incluidos en el libro cuentan con una definición, ya sea en el lenguaje de matemáticas, en el lenguaje de programación `Python`*, o incluso en ambos. Para estudiar con este libro se requiere un nivel mínimo de conocimientos de programación, por lo que es accesible para personas con el interés de aplicar este lenguaje en temas matemáticos.*

Se ha seleccionado `Python` *por diversas razones, entre ellas, porque es un lenguaje de propósito general y multiparadigma. Escribir las definiciones matemáticas usando* `Python`*, ayuda a cerrar la brecha entre programadores y matemáticos, acercando a los programadores a la formalidad y abstracción matemática; por otro lado, acerca a los matemáticos a la efectividad y eficiencia computacional.*

Abdiel E. Cáceres González
Doctor en Ciencias con especialidad en
Ingeniería Eléctrica en el área de Computación.
Área de interés fundamentos teóricos de computación.
Autor de:
Lenguajes y Autómatas: Una perspectiva funcional con Racket (2019) UJAT. DOI https://doi.org/10.19136/book.152
Matemáticas Discretas: Una perspectiva funcional con Python 3.x (2023) UJAT. DOI https://doi.org/10.19136/mg7g4lk2
Nació en Ciudad de México, México.

ISBN 978-6-0726285-6-4