Matemáticas Discretas

Una perspectiva funcional con Python 3.x

Abdiel E. Cáceres González

Universidad Juárez Autónoma de Tabasco

"Estudio en la duda. Acción en la fe"

Matemáticas Discretas

Una perspectiva funcional con Python 3.x

COLECCIÓN
HÉCTOR OCHOA BACELIS
Textos de enseñanza de ciencias básicas

Guillermo Narváez Osorio

Rector

Hermicenda Pérez Vidal

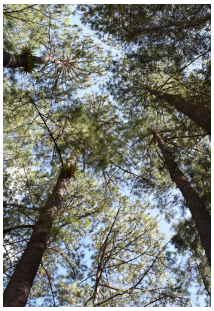
Directora de la División Académica de Ciencias Básicas

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

Matemáticas Discretas

Una perspectiva funcional con Python 3.x

Abdiel E. Cáceres González



Fotografía en portada y título de capítulos:
«*Árboles chiapanecos*»
por: Sara Camila Cáceres Pastrana,

Primera edición, 2023

www.ujat.mx
ISBN 978-607-606-632-4
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.

Maquetación: Abdiel E. Cáceres González
Fotografía: Sara Camila Cáceres Pastrana
Corrección de estilo: Abdiel E. Cáceres González
Hecho en Cunduacán, Tabasco, México

Para Amparo y Sara Camila

Índice general

Prefacio

Este libro es el resultado de algunos años de coleccionar y mejorar el material que empleo en el curso de Matemáticas Discretas, en la Universidad Juárez Autónoma de Tabasco, en México; también he incluido algunas notas realizadas para un curso de Profundización, en la Universidad Surcolombiana en Neiva, Huila, Colombia.

Este libro cubre el material de estudio que generalmente se atiende en los cursos básicos de matemáticas discretas. Los temas que son tratados en este material van desde lógica proposicional, hasta teoría de grafos. El enfoque del estudio propuesto es mayormente práctico, haciendo (es decir, programando) casi todas las definiciones matemáticas que usualmente se estudian de manera teórica.

He planeado este documento pensando principalmente en aquellas personas de habla hispana que participen en un primer curso de matemáticas discretas, por lo que en principio no se requiere algún conocimiento previo de matemáticas, excepto la aritmética básica que se aprende en los cursos de educación elemental. La intención es que se logre un conocimiento práctico y profundo de los temas fundamentales de la teoría de Matemáticas Discretas, y al mismo tiempo mejore las habilidades de programación con un lenguaje multiparadigma como lo es `Python`.

La decisión de utilizar el lenguaje de programación `Python` para la implementación de las funciones matemáticas es porque se trata de un lenguaje que ocupa una sintaxis muy sencilla por lo que se aprende rápidamente, lo cual es sumamente útil cuando se desea expresar en código los conceptos y procedimientos involucrados. A lo largo de este texto, trato de conducir al lector en la escritura en sintaxis de `Python` de las expresiones en lenguaje matemático, para definir de manera efectiva desde los conceptos más simples y fundamentales, hasta las expresiones que son necesarias para describir procedimientos más complejos; sin embargo aún en los últimos temas, se mantiene un nivel de simpleza bastante aceptable para todo nivel de usuarios.

El contenido del libro lo he dividido en cuatro partes:

1. Lógica proposicional,
2. Operaciones con conjuntos,
3. Relaciones y Funciones,
4. Grafos dirigidos.

Al principio, en la parte de lógica proposicional, se tratan principalmente los temas de valores de verdad; proposiciones lógicas, combinaciones con proposiciones; predicados; combinaciones con predicados. Así como las nociones fundamentales de conjuntos para terminar con cuantificadores.

La segunda parte se trata acerca de conjuntos. Una vez que se tiene la noción de conjuntos, el tema fundamental aquí es diseñar procedimientos eficientes que permitan obtener nuevos conjuntos a partir de la operación con conjuntos. Estudiaremos aquí temas como la unión, la intersección entre otras operaciones. Hacia el final de esta parte, se analizan dos conjuntos importantes: el conjunto potencia de un conjunto y el producto cartesiano entre conjuntos, lo que nos permitirá trabajar con procedimientos de combinatoria computacional para generar los elementos de estos conjuntos.

La tercera parte incluye los temas de relaciones y funciones. En los capítulos que comprenden esta parte estudiaremos cómo modelar las relaciones en el lenguaje de programación `Python`; podremos identificar los elementos conceptuales que definen las relaciones y analizaremos las propiedades de las relaciones. Como las funciones son un caso particular de las relaciones, estudiaremos y programaremos procedimientos especiales para manipular relaciones y funciones; y podremos clasificar las funciones de acuerdo a las propiedades que presenten.

Finalmente en la cuarta parte del libro escribiremos procedimientos efectivos para definir y operar grafos. En esta parte se incluyen las listas y árboles, ya que son casos particulares de los grafos. Tendremos la oportunidad de hacer programas para graficar y visualizar los resultados de algunas consultas características de los grafos, como los caminos, ciclos e itinerarios.

Como una de las metas de este libro ha sido proporcionar al lector del conocimiento necesario, para obtener la habilidad de definir funciones que le permitan un conocimiento práctico del tema, se ha incluido más de 90 programas en `Python` que han sido probados y que sustentan las bases de los conceptos descritos en el libro. Al final del libro se encuentran estas funciones juntas para que el lector pueda seguir al detalle el texto.

Como el código fuente es una parte muy importante del texto, se incluyen las definiciones y expresiones que se han tipografiado tal y como aparecerían en el IDE de `Python`:

- La tipografía general del código fuente se escribe en letra `TrueType`, como al definir la variable `edad = 45`.
- El resultado de las evaluaciones se escribe en un recuadro separado del código, cada evaluación aparece después del prompt **>>>**, que es el símbolo indicativo de la consola de `Python` para interactuar con el programa.
- Los comentarios en el código se escriben en itálicas y anteponiendo un símbolo #, como en: *# Esto es un comentario en el código*.
- Al interactuar con `Python`, en las evaluaciones ocasionalmente surgen errores. Con el fin de que el programador observe este comportamiento, he escrito en el texto algunas interacciones equivocadas, donde el error se escribe con el texto que aparecería en la consola de `Python`, como en el siguiente mensaje: *NameError: name 'r' is not defined*

Los programas en Python se han escrito en recuadros de texto que incluyen una enumeración a la izquierda como el siguiente:

```
1 def mayoresDe(m:float, L:list)-> list:
2     """
3     Selecciona aquellos elementos de L que son mayores a m
4     """
5     res = list(filter(lambda j: j>m, L))
6     return res
```

Cada programa cuenta con números de línea a la izquierda para una mejor referencia, así como comentarios de ayuda al inicio de cada función, lo que permite establecer un texto de asistencia para el usuario de la función.

Casi siempre después de un segmento de código que define algún concepto en Python, se muestra el uso del concepto mediante una interacción. 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:

```
>>> mayoresDe(4, [2, 3, 8, 4, 9]) # esto es una interacción
[8, 9] # esto es la evaluación de la interacción
>>>
```

Las interacciones se escriben con la finalidad de que el lector, al escribir su propio código, compare la salida que le ofrece su interacción con la salida ofrecida en este texto. Tanto las interacciones como el código fuente se han tipografiado de una manera muy similar a la que se presenta en un IDE de Python, con el fin de que la experiencia de aprendizaje del propio lector respecto al lenguaje de programación sea lo más amable posible.

Ocasionalmente se hace uso de alguna característica especial que es propia del lenguaje de programación, como el uso de una nueva función primitiva, o el uso de sintaxis particular, o incluso algún comentario que se relaciona específicamente con el lenguaje; este tipo de comentarios se señalan con el logotipo del lenguaje, como el siguiente:

Python es un lenguaje de programación de alto nivel y multiparadigma. Python fue creado con el propósito de generar soluciones de cómputo para una muy amplia variedad de aplicaciones, como programación de sistemas, aplicaciones WEB, la ciencia de datos, aprendizaje automático y muchas otras más.

En algunas ocasiones se hace una nota, señalada con un ícono de una mano apuntando, para aclarar cierto texto, por ejemplo, relacionar alguna definición escrita en Python, con la misma definición escrita en un modo convencional utilizando símbolos y notación matemática convencional como en:

El símbolo ∀ se emplea en expresiones lógicas de orden superior, se escribe como

$$\forall(x \in A) : P(x)$$

donde:

A es un conjunto.
P es un predicado.

En Python se puede representar como `paraTodo(P:callable, A:list)`

Este libro cuenta con toda la colección de definiciones definidas a lo largo del texto, sin embargo, también se ofrecen todas juntas en el apéndice A a partir de la página 251.

Aunque el lector puede copiar el código, insertarlo y ejecutarlo utilizando el IDE de su computadora, es más recomendable leerlo y escribirlo a medida que vaya avanzando en su estudio.

Todas las definiciones hechas en Python han sido probadas mediante ejemplos, que se han enmarcado en un recuadro para simular una interacción, como en el siguiente ejemplo:

Ejemplo 0.1

En este ejemplo se muestra una interacción típica en Python

```
>>> A = [1, 2, 3, 4, 5]
>>> A[0]
1
>>> A[1:]
[2, 3, 4, 5]
>>>
```

He puesto especial atención en ofrecer una bibliografía que combina textos clásicos como textos de más reciente publicación, con el fin de complementar el conocimiento, combinando la teoría y la práctica.

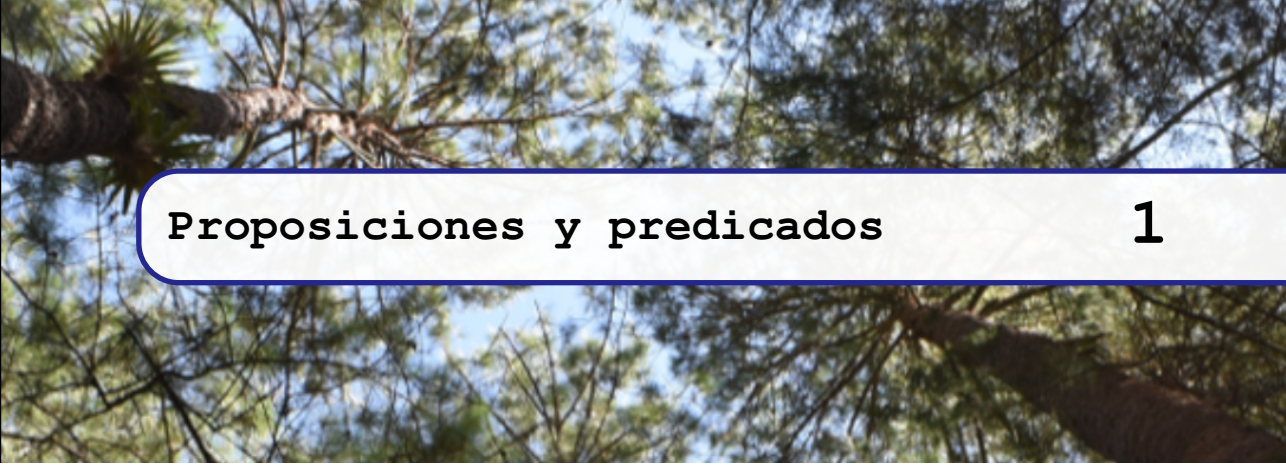
En el índice alfabético se enlistan también las palabras clave que han sido definidas como esVacio o pCart; así como las que son propias de Python, como filter, o list. Puedes consultar este índice cuando lo estimes oportuno.

Este texto se ha escrito en LaTeX [https://www.latex-project.org/], utilizando el paquete listings para escribir el código fuente. La versión de Python que se utilizó es la versión 3.10 de Python (https://www.python.org/). Las imágenes para mostrar los grafos se realizaron con Graphviz [http://www.graphviz.org/], otras imágenes se realizaron con Inkscape y GIMP [GNU Image Manipulation Program]. Todos estos programas utilizados tienen licencia de libre uso.

Abdiel Emilio Cáceres González
Abril 2023

I

Lógica computacional



Proposiciones y predicados 1

1.1 Presentación

«Lógica» es una palabra de origen griego: *Logos* ($\lambda o \gamma o \varsigma$), que significa razón, reflexión, pensamiento. Actualmente se utiliza la palabra «lógica» para designar el estudio formal de los razonamientos y las conclusiones a las que conducen tales razonamientos. En matemáticas ha resultado extremadamente útil tener una manera formal de establecer conclusiones a partir de hechos y combinaciones de hechos, porque se evitan así muchas ambigüedades y malos entendidos.

Se dice que la lógica es un «sistema formal» porque está constituido por un lenguaje formal, reglas gramaticales y reglas de inferencia. Cada uno de los constituyentes del sistema se debe establecer de antemano, para construir razonamientos a partir de los hechos más básicos y fundamentales, hasta aquellos que se hayan construido al considerar las reglas de inferencia.

Particularmente estaremos interesados en el estudio de la lógica simbólica, que hace uso de símbolos para representar tanto los hechos como las reglas de inferencia.

1.2 Los valores de verdad

1.2.1 Valores booleanos

En la lógica *clásica*, también conocida como *dura*, o *bivalente*, hemos de considerar únicamente dos valores de verdad, el **falso** y el **cierto**. Cualquier aseveración lógica tiene exactamente una de esas dos propiedades. El valor falso lo escribiremos con el símbolo `False` , y el cierto con el símbolo `True`. Aunque en otras áreas de la lógica, como la lógica multivalente o la lógica difusa se emplean otra gama de valores de verdad, aquí consideraremos solamente `False` y `True` como los únicos valores de verdad.

Hay muchos otros sistemas bivalentes que pueden ser útiles para representar los valores falso y cierto, como el `0` para representar falso y `1` para representar el cierto. Otros sistemas bivalentes pueden ser abajo-arriba; izquierda-derecha, en fin, cualquier conjunto de dos símbolos diferentes entre sí puede servir para representar los valores de verdad `False` y `True`.

☞ La lógica matemática es un campo de estudio sumamente extenso, lo que se presenta aquí es un breve repositorio de definiciones que deben alentar al lector a indagar más sobre el tema. Excelentes libros para empezar un estudio diligente en la lógica matemática son «A Beginner's Guide to Mathematical Logic» [Smu14] y «Lógica Elemental» [FdPSTF04] pero sin duda, de una rápida búsqueda en internet es posible obtener muchos otros títulos más.

Los valores de verdad también se conocen como «valores booleanos» en honor al matemático británico George Boole (1815 - 1864), quien nos ha legado lo que ahora conocemos como álgebra de Boole [Boo09], donde se utilizan técnicas algebraicas para operar con proposiciones lógicas. Así los valores booleanos son `False` y `True`, y nos referiremos a ambos con el símbolo $\mathbb{B}$, y diremos que representa el dominio de los valores booleanos.

1.2.2 Valores booleanos en Python

En `Python` se dedican las constantes `True` para el valor cierto y `False` para el valor falso. Pero `Python` es un poco menos restrictivo, pues cualquier valor tiene una representación como `True` o `False`. Veamos por ejemplo:

- `0` es `False`. Cualquier valor numérico diferente que `0` es `True`.
- "" es `False`. Cualquier string diferente que "" es `True`.
- `[]` es `False`. Cualquier lista diferente que `[]` es `True`.
- `()` es `False`. Cualquier tupla diferente que `()` es `True`.

Ejemplo 1.1

Al utilizar el IDE de `Python`, en la sección de interacciones escribiremos los valores de verdad para que el intérprete pueda leerlos, evaluarlos y ofrecer una impresión del resultado.

```
>>> True # El símbolo para cierto
True
>>> False # El símbolo para falso
False
>>> bool("")
False
>>> bool("hola")
True
>>> bool([])
False
>>> bool([1])
True
>>>
```

1.3 Expresiones simbólicas

En `Python` hay expresiones simbólicas **primitivas** que son las expresiones que no requieren programación por parte del usuario, ya que han sido programadas previamente y cargadas al sistema de manera automática al iniciar `Python`. Las expresiones primitivas ya tienen un valor establecido y no requieren cálculo alguno ni para determinar su valor, ni para utilizarlas.

En Python al igual que en matemáticas, se utilizan símbolos para escribir expresiones, estas se llaman **expresiones simbólicas** o bien, **S-expresiones**. En adelante consideraremos sinónimos los términos «S-expresión» y «expresión»; y usaremos el término «S-expresión» para hacer énfasis en que la expresión del lenguaje utiliza símbolos.

El entorno de desarrollo integrado (IDE) de Python se maneja en un ciclo interactivo denominado **REPL**, (del inglés *Read*, *Evaluate*, *Print*, *Loop)*; cada expresión debe ser leída por el intérprete, luego debe evaluarse y el resultado se imprime, o bien se utiliza como entrada para un nuevo ciclo.

Una expresión que al ser analizada es finalmente evaluada como `True` o `False` se llama **expresión lógica**, o bien **expresión booleana** porque su valor es uno del conjunto de valores booleanos.

1.3.1 Expresiones simples

En Python hay principalmente dos tipos de S-expresiones, las expresiones simples y las expresiones compuestas. Las expresiones simples son expresiones cuyo valor es la misma expresión, como en el ejemplo 1.1 con los símbolos `True` o `False`, para el caso de las expresiones booleanas; pero también los números son expresiones simples como `10`, `24.48` o `3+2j`; las cadenas de caracteres entre comillas como `"hola mundo"` también son expresiones simples. Las expresiones simples también se conocen como expresiones atómicas.

Ejemplo 1.2

Expresiones simples

- `True` *Es un valor de verdad*
- `"falso"` *Es una cadena de texto*
- `3.14159` *Es un número*
- `3j+2` *Es un número*
- `['w', 'x', 'y', 'z']` *Es una lista de cadenas de texto*

1.3.2 Expresiones compuestas

Son expresiones que incluyen un símbolo que representa a un **operador** junto con una o más expresiones, ya sean simples o compuestas. Cada elemento que no es el operador, se llama **argumento**. Por ejemplo `5+(2*2)` es una S-expresión compuesta, de un operador `+`, con dos argumentos, el `5` que es expresión simple y `(2*2)` es una expresión compuesta.

En el lenguaje cotidiano, frecuentemente podemos encontrar expresiones lógicas compuestas, las expresiones compuestas se escriben agrupando expresiones utilizando operadores lógicos, que se conocen también como **conectores** o **términos de enlace**. En Python escribimos expresiones compuestas utilizando paréntesis cuando es necesario agrupar expresiones.

Ejemplo 1.3

Las siguientes sentencias son expresiones compuestas en el lenguaje cotidiano.

- «Los niños son juguetones y traviesos».
- «La medicina se puede tomar antes o después de la comida».
- «El usuario o bien es administrador o es un usuario regular».
- «Si llovió, entonces hay humedad».

Como regla general una expresión compuesta **empieza con un operador** y sus **operandos** se escriben como una lista entre paréntesis, siguiendo el siguiente patrón

```
operador(operando[...])
```

Separado cada operador con una coma. El símbolo `...` significa *y así sucesivamente, con cero o más elementos*; de paso, hay otro símbolo que utilizaremos de vez en cuando, es `...+` que significa: *y así sucesivamente, con uno o más elementos*.

Ejemplo 1.4

Las siguientes son expresiones compuestas:

- `sumar(4, 5, 6)`
- `map(lambda x: F(x), D)`
- `5*(3*y+x**2)`
- `45*q >= L[i]`

El nombre en `Python` para los identificadores de las funciones, se escribirán siguiendo un estilo `minusculaMayuscula`; mientras que los nombres para las clases y objetos con un estilo `PalabrasConMayuscula`.

1.3.3 Expresiones proposicionales

Una **expresión proposicional**, como lo dicen Copi y Cohen en [CC13, p. 5], «son el material de nuestro razonamiento». Una proposición es una expresión lógica, por lo que es evaluada o bien como `False` o como `True`, pero no puede adquirir ambos valores al mismo tiempo y tampoco es posible que carezca de valor lógico.

Las proposiciones también pueden ser simples o compuestas. Las proposiciones simples, también llamadas **atómicas**, son expresiones lógicas que no pueden seccionarse sin perder el valor de verdad.

Ejemplo 1.5

Los siguientes ejemplos son proposiciones escritas en el lenguaje cotidiano o lenguaje natural:

1. «Mentira» [significa *falso*].
2. «Es cierto» [significa *cierto*].
3. «El número de sesiones ha sobrepasado el límite».

Observa que las proposiciones en lenguaje natural están escritas en tiempo presente o en pasado, porque son hechos que ya han sido evaluados.

Las siguientes son expresiones simples en `Python`:

1. `True`. *Es primitiva, significa cierto*
2. `False`. *Es primitiva, significa falso*
3. `4 == 4`. *produce verdadero*
4. `14 == 5`. *es evaluado como falso*

En `Python`, una expresión primitiva ya cuenta con el valor de todos sus argumentos, por lo que no requiere la evaluación de alguna subexpresión para determinar su valor.

Ejemplo 1.6

Las siguientes expresiones NO son proposiciones lógicas:

- «¿Cómo se fabrica el pan?». *Es una pregunta, no un valor de verdad.*
- «Sirve la mesa». *Es una orden, no un valor booleano.*
- $3x > 27$. *No es posible determinar el valor de verdad.*

Las proposiciones compuestas son proposiciones que incluyen otras expresiones simples o compuestas.

Ejemplo 1.7

Las siguientes expresiones son proposiciones compuestas:

- «El usuario lleva 3 horas de uso continuo o el usuario no ha hecho modificaciones».
- «Si Roberto es constante en sus pagos y paga puntualmente, entonces Roberto es recompensado como cliente distinguido».
- $3*4 = 4*3$.
- $2 > (2+2)$

Al evaluar las expresiones se obtiene un valor, por ejemplo la expresión $2 > (2+2)$ al ser evaluada debe ser interpretada como `False`, por lo que se puede escribir

$$2 > (2+2) \mapsto \texttt{False}$$

Utilizaremos el símbolo $\mapsto$ para decir «es evaluado como», o «produce el valor», o simplemente «produce».

En general, al escribir

$$\langle \textit{expresión} \rangle \mapsto \langle \textit{valor} | \textit{tipo} \rangle,$$

se debe entender que se está evaluando la expresión a la izquierda de la flecha y se devuelve el valor o el tipo de dato que aparece a la derecha de la flecha. Esta práctica es común al utilizar `Python` con anotaciones de tipos [1].

1.3.4 Definiendo proposiciones en Python

En `Python` es posible crear proposiciones al definir conceptos con un valor de verdad. El lenguaje `Python` es un lenguaje funcional, esto significa que se deben crear segmentos de código con alguna funcionalidad específica. Este es un método de **definición**, donde el identificador es el significante y la funcionalidad es el significado. Particularmente en el caso de las proposiciones, el significado siempre es un valor de verdad, o bien una expresión primitiva o compuesta que finalmente sea evaluada con un valor de verdad.

```
>>> elPatoEsAve = True
>>> elPatoEsAveyElGatoEsMamifero = False
>>>
```

`Python` asocia un valor a un identificador. El concepto definido es el identificador y su significado es el valor asociado. En lo sucesivo, cuando se solicite el significado de ese nuevo concepto y mientras la memoria de la computadora no se restablezca, ya se conocerá su significado:

```
>>> elPatoEsAve
True
>>>
```

Una notación que también se utilizará frecuentemente en este texto, es la notación para asignar un valor a una variable, siguiendo el ejemplo, podemos escribir: `elPatoEsAve=True`, que significa literalmente «se le asigna el valor `True` a la variable `elPatoEsAve`», una traducción alternativa es «elPatoEsAve$\hookleftarrow$`True` ». El símbolo $\hookleftarrow$, se usa al escribir algoritmos o en el lenguaje convencional de matemáticas para asignar un valor a una variable.

[1] `https://docs.python.org/3/library/typing.html`

Ejemplo 1.8

Supongamos que se tiene la expresión «Gustavo es usuario autorizado», la cual queremos modelar en Python. Podemos crear un identificador gustavoEsUsuarioAutorizado y definirlo como una proposición con valor de verdad True. Un primer acercamiento es escribir:

$$gustavoEsUsuarioAutorizado \leftarrow\!\shortmid \texttt{True}$$

lo que puede ser modelado en Python como:

```
gustavoEsUsuarioAutorizado = True.
```

Recordemos que en Python se utiliza un símbolo = para la asignación y el símbolo compuesto == para la comparación. Una vez que se ha definido el nuevo concepto, es posible utilizarlo sabiendo el valor de verdad con el cual ha sido definido:

```
>>> gustavoEsUsuarioAutorizado
True
>>>
```

Utilizar un concepto que no ha sido definido previamente ocasiona un error.

```
>>> hoyEsSoleado
NameError: name 'hoyEsSoleado' is not defined
>>> hoyEsSoleado = True
>>> hoyEsSoleado
True
>>>
```

Al definir proposiciones en Python, el significante [el identificador] no necesariamente debe tener sentido para las personas; por ejemplo xy345 es un identificador válido, también lo son home o casa, que no necesariamente tienen relación de significado alguno entre sí.

Preferiblemente se deben escribir identificadores con sentido, al menos para el programador y que tengan relación con un significado ya conocido, principalmente es para beneficio de las personas que revisen el código fuente.

Ejemplo 1.9

Las siguientes proposiciones están correctamente definidas, pero son muy largas o confusas.

```
>>> elLunesEsDomingo = True
>>> losPerrosSonPeces = True
>>> losPerrosSonPeces
True
>>>
```

Las siguientes proposiciones son correctamente definidas y *más amigables* para el lector humano.

```
>>> llueve = True
>>> haceFrio = False
>>> haceFrio
False
>>>
```

1.4 Predicados

Un **predicado** es un procedimiento que contiene expresiones de valor desconocido, que al proporcionar tales valores y ser evaluado, genera un valor de verdad. Los identificadores cuyo valor es desconocido se conocen como **parámetros**, también son conocidos como parámetros formales o argumentos ficticios [Ben78].

En el lenguaje natural, los predicados están asociados con expresiones en tiempo gramatical futuro, pues aún no se conoce con certeza el valor de al menos uno de los componentes de la expresión. En matemáticas, las **incógnitas** frecuentemente se escriben con los símbolos x, y o z; en lógica es frecuente usar símbolos p, q o r.

El valor de verdad de un predicado depende del valor de cada uno de sus parámetros y de cómo se combinan. Los parámetros no siempre son expresiones lógicas, también pueden constituirse de expresiones construidas con números, cadenas de caracteres o cualquier otro objeto; cuando una expresión tiene elementos que no son booleanos, debe haber operaciones que permitan que la evaluación final de la expresión tenga un valor booleano.

Ejemplo 1.10

Las siguientes expresiones son el mismo predicado. El primero está escrito como λ-expresión; el segundo en la notación de `Python` [ver la nota siguiente sobre expresiones lambda]; y el tercero en lenguaje natural:

- $\lambda x \cdot 3x = 12$.
- `lambda x: 3*x == 12`.
- «El triple de un número es doce».

Una **expresión lambda**, escrita también como λ-expresión, sirve para escribir procedimientos. Un procedimiento toma algunos datos de entrada, los procesa y emite un resultado. Los datos de entrada se establecen por medio de variables declaradas al principio de la expresión, luego se escribe una expresión que considera esos datos y el resultado de la evaluación de esa expresión es lo que se emite como resultado. Un procedimiento que emite un valor booleano es un predicado. Por ejemplo la λ-expresión $\lambda x \cdot 3x = 12$ es un procedimiento que toma como dato de entrada un valor x, con el que se debe verificar que $3x = 12$; el resultado emitido será `True` o `False`, dependiendo del valor de la variable de entrada x. La marca $\cdot$, sirve para separar las variables de la expresión a evaluar. En `Python` las λ-expresiones se escriben con el siguiente formato:

```
lambda [params]: expr
```

Que puede leerse como «El predicado [el procedimiento] que requiere los parámetros `params`, con los que se debe evaluar la expresión `expr`». Como en $\lambda x, y \cdot x + y > 15$ y en `Python` se traduce `lambda x,y:x+y>15`.

El símbolo λ, proviene del λ-cálculo, que es un formalismo dado a conocer por Alonzo Church [Chu36, Chu41], como una manera formal y rigurosa de expresar las nociones de *función* y *aplicación de una función*.

Cuando se establece el valor para un parámetro, se dice que este ha sido instanciado con el valor dado. El predicado puede ser evaluado una vez que todos sus términos variables han sido instanciados. Un predicado que ya ha sido evaluado se convierte en una proposición.

En el momento en que se invoca el predicado se establecen los valores que deben tomar los parámetros del predicado, estos valores específicos se llaman **argumentos**. Los argumentos también reciben el nombre de argumentos actuales o parámetros actuales [Ben78].

Ejemplo 1.11

Considera nuevamente el predicado $\lambda x \cdot 3x = 12$. Para evaluar este predicado es necesario asociar la variable x con algún valor adecuado; luego realizar la operación escrita para finalmente emitir el resultado. La evaluación se hace encerrando entre paréntesis el predicado e indicando el valor que debe tomar la variable:

$$(\lambda x \cdot 3x = 12)(5)$$

El número de parámetros en un procedimiento, establece la **aridad** del mismo [CF08, p. 121]. Así un procedimiento de aridad 0, está definido sin parámetros, por lo que en las invocaciones a un procedimiento de aridad 0, no requiere argumentos. Los procedimientos de aridad 1 requieren un solo argumento en la invocación.

Ejemplo 1.12

El predicado $\lambda x, y \bullet x+2 > y^3$ es de aridad dos. Ahora se muestra el predicado y los argumentos con los que se instancían los parámetros del predicado, junto con su evaluación. $(\lambda x, y \bullet x+2 > y^3)(3,4)$. Aquí `x` $\leftarrow\!\!\!\shortmid$ `3`, y `y` $\leftarrow\!\!\!\shortmid$ `4`. En `Python` podemos escribir:

```
>>> (lambda x,y:2+x > y**3)(3,4)
False
>>>
```

1.4.1 Operadores

Un **operador** es un identificador al cual se le ha asociado un procedimiento como significado. Los parámetros del procedimiento deben ser instanciados con argumentos adecuados, especificados en la declaración de los parámetros. Los términos «operador» y «función» usualmente son considerados como sinónimos; inclusive el término «procedimiento» suele referirse a la función que realiza.

El nombre de los operadores se escriben mediante una cadena de caracteres usualmente de longitud muy pequeña, puede ser un único caracter como el operador de raíz cuadrada [p. ej. $\sqrt{4}$], o bien un símbolo compuesto de varios caracteres, como el operador de módulo [p. ej. 5 mód 2], pero debe ser suficientemente expresivo como para ser asociado con la función que realiza el predicado. El nombre del operador sirve para hacer referencia al procedimiento que se realiza.

Un **operador lógico** es un operador [*procedimiento que genera un valor booleano*] definido con parámetros formales de tipo booleano. Un operador lógico suele tratarse de la misma manera que una **función lógica**.

Ejemplo 1.13

El operador lógico aparece primero como expresión lambda, por lo que al ser evaluado se obtiene `<function <lambda>at 0x7f5e43d6e820>` [el número puede cambiar]. Al definir el símbolo `neg` con la expresión lambda anterior, se crea una asociación de un procedimiento con un nombre, ya entonces puede ser utilizado [invocado] con argumentos actuales para sus formales, en esta caso el argumento formal es `x`:

```
>>> lambda x: False if x else True
<function <lambda> at 0x7f5e43d6e820>
>>> neg = lambda x: False if x else True
>>> neg(True)
False
>>>
```

Los operadores lógicos funciones que reciben valores booleanos como argumentos y devuelven como resultado un valor booleano. Esto permite construir expresiones booleanas más complejas a partir de expresiones booleanas más simples.

Un operador puede ser **unario**, **binario**, de **aridad múltiple** o de **aridad indeterminada**. Los operadores unarios requieren exactamente un parámetro; los binarios requieren exactamente dos; los operadores de aridad múltiple requieren uno o más parámetros, pero deben definir la cantidad de parámetros que requieren; por último los

operadores de aridad indeterminada, requieren cero o más parámetros, sin determinar de antemano la cantidad de ellos.

La siguiente función sirve para determinar la aridad de una función. Requiere el método `signature` de la biblioteca `inspect`:

Código 1.1: *Aridad de una función*

```
from inspect import signature

def aridad(fun):
    """
    Obtiene la aridad de una función.
    """
    return len(signature(fun).parameters)
```

Ejemplo 1.14

Una vez escrito y ejecutado el código, revisemos la aridad de un par de funciones, una de aridad 1 y la otra de aridad 2:

```
>>> aridad(lambda x:x+1)
1
>>> aridad(lambda x,y:x>y)
2
>>>
```

1.5 Operadores lógicos unarios

1.5.1 Negación

Definición 1.5.1 -- Negación. Si p es una expresión booleana, la negación de p se escribe $\neg p$ y es otra expresión booleana con el valor de verdad diferente que p, así

$$\neg p \mapsto \begin{cases} \texttt{False} & \text{si } p. \\ \texttt{True} & \textit{eoc.} \end{cases}$$

Otros símbolos que aparecen en textos de lógica con el mismo efecto son $\not p$, $\sim p$, p' o incluso $\bar{p}$. También puede aparecer como la negación del resultado de una operación, como en $x \not\leq y$ para decir que no es verdad que $x \leq y$, lo que tiene el mismo efecto que $\neg(x \leq y)$. El término *eoc* significa literalmente «en otro caso».

En `Python` definimos el predicado `neg` con el comportamiento de la negación:

Código 1.2: *La negación*

```
def neg(p:bool)-> bool:
    """ La negación de una proposición
    Devuelve el valor de verdad opuesto a p.
    """
    if p:
        return False
    else:
        return True
```

Probemos nuestro código con los valores de verdad.

```
>>> neg(True)
False
>>> neg(False)
True
>>>
```

Las definiciones de funciones en `Python` tienen cuatro partes:

1. Los comentarios sirven de ayuda para el usuario de la función. En el programa del código 1.2 los comentarios son:

```
2       """ La negación de una proposición
3       Devuelve el valor de verdad opuesto a p.
4       """
```

   Aunque los comentarios son opcionales, es muy importante escribirlos, con el fin de que el código sea más fácil de leer para las personas.
2. La palabra clave `def`, indica la creación de un nuevo concepto o nueva definición.
3. El identificador, es una palabra con la que se conocerá la **función** del procedimiento que se define. En el código 1.2, el identificador es el nemónico `neg`.
4. La expresión o expresiones que conforman el procedimiento. En estas expresiones se incluye, si se requiere, al menos una expresión `return` que devuelve un valor.

```
7       if p:
8          return False
9       else:
10         return True
```

En `Python`, la expresión `if` tiene diferentes formatos. El siguiente formato corresponde al uso tradicional:

`if` ⟨*expr-lógica*⟩:
   ⟨*caso-verdadero*⟩
`else:`
   ⟨*alternativa*⟩

El valor de ⟨*expr-lógica*⟩ determina cuál es la siguiente expresión a evaluar. Si ⟨*expr-lógica*⟩↦`True`, entonces se evalúa la expresión ⟨*caso-verdadero*⟩; en caso contrario, se evalúa ⟨*alternativa*⟩. En expresiones condicional cortas, se puede escribir en un solo renglón como:

⟨*caso-verdadero*⟩ `if` ⟨*expr-lógica*⟩ `else` ⟨*alternativa*⟩

Cuando el parámetro `p` es instanciado con algún valor de verdad, ya no es mas un predicado, sino que es una proposición. La expresión condicional `if` determina su valor en dependencia del valor actual de `p`.

En la expresión `False if p else True` , si `p`↤`True`, el resultado es el valor `False`; cuando no es el caso que `p`↤`True`, la evaluación es el valor `True`.

Ejemplo 1.15

Probaremos el nuevo concepto `neg` con los dos valores booleanos, luego definiremos una proposición con significado particular para utilizar el operador `neg` con esa nueva proposición:

```
>>> neg(True)
False
>>> neg(False)
True
>>> enInviernoHaceFrio=True
>>> neg(enInviernoHaceFrio)
False
>>>
```

En el lenguaje cotidiano es posible encontrar la negación en diferentes formas. En las siguientes expresiones se ha utilizado la negación:

- «No es cierto que se ha actualizado el sistema», «el sistema no ha sido actualizado».
- «No es cierto que el sistema es operante», «el sistema no es operante».

- «No hay elementos en el conjunto», «el conjunto no tiene elementos».
- «El vaso no está vacío».

En el lenguaje natural, en ocasiones es posible traducir una expresión que utiliza la negación en una forma equivalente que no la utilice:

- «No es cierto que el sistema es operante» se puede decir como «el sistema es inoperante».
- «No hay personas dispuestas», «hay personas indispuestas».

La negación tiene la propiedad de que, si el argumento de la función es, de hecho, la negación de una proposición, el valor de verdad resultante es el valor de la proposición original, como si hubieran sido canceladas dos operaciones de negación:

$$\neg\neg p \mapsto p$$

- «No es cierto que no entré a clase»↦«Entré a clase».

Ejemplo 1.16

La negación de una negación. Antes de hacer este ejemplo, asegúrate de haber realizado el ejemplo 1.15 en la misma sesión de interacciones.

```
>>> neg(neg(enInviernoHaceFrio))
True
>>>
```

1.6 Operadores lógicos binarios

Los predicados **binarios** sirven para expresar funciones con exactamente dos argumentos. Tienen la forma $\langle opr\rangle\,(\langle opnd1\rangle, \langle opnd2\rangle)$ donde $\langle opr\rangle$ es el operador lógico, que puede ser el nombre de un procedimiento, o bien una λ-expresión; y los $\langle opnd1\rangle$ y $\langle opnd2\rangle$ son los operandos, que son expresiones simbólicas.

Los primeros cuatro predicados que estudiaremos, que son el predicado Verdad, Falsedad, primera y segunda componente, se llaman **predicados binarios impropios** porque a pesar de que por su definición requieren dos parámetros, para la implementación utilizan uno o ninguno de ellos.

Por otro lado, los **predicados binarios propios** requieren dos parámetros y utilizan ambos para calcular el valor de salida.

1.6.1 Los predicados Verdad y Falsedad

Definición 1.6.1 -- Verdad. El predicado Verdad requiere dos valores booleanos, y sin importar el valor de sus argumentos, siempre devuelve el valor booleano `True`.

Una **tabla de verdad** [Cun16, SH92] expone en forma tabular los posibles valores de los parámetros proposicionales p y q del lado izquierdo; y del lado derecho el valor del predicado que resulta de la evaluación. Se suele construir una tabla de verdad, creando nuevos predicados hacia la derecha. El orden de los renglones puede cambiar.

Llamaremos T al predicado verdad. Si tanto p como q son expresiones booleanas, $T(p,q)$ es el predicado verdad evaluado con p y q, en ese orden. La siguiente tabla de verdad muestra las posibles situaciones:

p	q	$T(p,q)$
True	True	True
True	False	True
False	True	True
False	False	True

Código 1.3: *El predicado verdad*

```
def T(p:bool,q:bool)->bool:
    """ Predicado Verdad
    Sin importar los parámetros, devuelve True.
    """
    return True
```

La función tiene dos argumentos formales `p` y `q`, ambos booleanos, pero en la implementación no se utiliza ninguno de ellos, esto no es una buena práctica de programación, pero ilustra el concepto de predicado impropio.

La función `bool()` nos permite conocer el valor booleano de cualquier expresión, por lo que obtendremos un `True` o `False` en respuesta a la invocación de `bool`, en particular con valores numéricos, por ejemplo `bool(1)`$\mapsto$ `True` y `bool(0)`$\mapsto$ `False`. En general cualquier valor numérico es `True`, excepto el 0; cualquier cadena de texto es `True` excepto la cadena vacía y cualquier tupla, conjunto o diccionario es `True`, excepto las que sean vacías.

```
>>> bool(0)
False
>>> bool("")
False
>>> bool([])
False
>>> bool({})
False
>>> bool("hola")
True
>>> bool(1)
True
>>> bool([1,2])
True
>>> bool({1,2})
True
>>>
```

Ejemplo 1.17

Se prueba el código con los parámetros adecuados. Observa que siempre devuelve `True`.

```
>>> T(True, True)
True
>>> T(True, False)
True
>>> T(False, True)
True
>>> T(False, False)
True
>>>
```

Definición 1.6.2 -- Falsedad. El predicado Falsedad de dos proposiciones lógicas p y q siempre es `False`, sin importar el valor de verdad de p o de q.

Con la siguiente tabla de verdad:

p	q	$F(p,q)$
True	True	False
True	False	False
False	True	False
False	False	False

Código 1.4: *El predicado falsedad*

```
def F(p:bool,q:bool)-> bool:
    """ Predicado Falsedad
    Sin importar los parámetros, devuelve False.
    """
    return False
```

Ejemplo 1.18

Se prueba `F(p,q)` con todas las posibles entradas.

```
>>> F(True, True)
False
>>> F(True, False)
False
>>> F(False, True)
False
>>> F(False, False)
False
>>>
```

1.6.2 Los predicados primera y segunda componente

La primera/segunda componente de un predicado binario, se refiere al lugar que ocupan los parámetros; la primera componente se refiere al primer parámetro [de izquierda a derecha], mientras que la segunda componente es el segundo parámetro.

Definición 1.6.3 -- Primera componente. Si p y q son expresiones booleanas, llamaremos Primera componente al predicado $P(p,q)$ que produce `True` cuando $p \leftarrow\!\shortmid$ `True`; cuando $p \leftarrow\!\shortmid$ `False`, $P(p,q)$ produce `False`. Así el valor devuelto es el valor actual de p y se obtiene sin considerar el valor de q.

La definición formal en una λ-expresión es $P \leftarrow\!\shortmid \lambda p,q \bullet p$.

Definición 1.6.4 -- Segunda componente. El predicado Segunda componente $Q(p,q) \mapsto$ `True` cuando $q \leftarrow\!\shortmid$ `True` y $Q(p,q) \mapsto$ `False` cuando $q \leftarrow\!\shortmid$ `False`; sin importar el valor actual de p.

La definición formal en una λ-expresión es $Q \leftarrow\!\shortmid \lambda p,q \bullet q$.

p	q	$P(p,q)$	$Q(p,q)$
True	True	True	True
True	False	True	False
False	True	False	True
False	False	False	False

Para la implementación basta utilizar p si es el predicado primer componente; o q cuando se trata del predicado segundo componente.

Código 1.5: *El predicado primera componente*

```
def P(p:bool, q:bool)->bool:
    """ Primera componente
    Devuelve el valor del primer parámetro p.
    """
    return p
```

Código 1.6: *El predicado segunda componente*

```
def Q(p:bool, q:bool)->bool:
    """ Segunda componente
    Devuelve el valor del segundo parámetro q.
    """
    return q
```

Ejemplo 1.19

El comportamiento de los predicados primera y segunda componente difiere de acuerdo al valor de cada uno de sus componentes:

```
>>> P(True, False)
True
>>> Q(True, False)
False
>>>
```

1.6.3 Conjunción y disyunción

Definición 1.6.5 -- Conjunción. Si p y q son expresiones lógicas, la **conjunción** de p con q se escribe $p \wedge q$ cuyo valor de verdad es `True` si tanto p como q son `True`, en otro caso será `False`.

En el lenguaje cotidiano se pueden encontrar expresiones lógicas compuestas que incluyen la conjunción, como en los siguientes casos:

- «El cliente puede seleccionar opciones y puede comprar artículos».
- «El día es lluvioso y el día es soleado».
- «La dama es inteligente y el caballero es amable».

También podemos tener ejemplos de conjunciones cuando se construyen proposiciones hechas con operaciones matemáticas como en $(80 > 20) \wedge (51^2 + 3 < 100)$.

La definición formal en una λ-expresión es $\wedge \leftarrow\!\shortmid \lambda p, q \bullet q$ Si p , `False`.

Para la conjunción de dos valores booleanos, tenemos los cuatro casos mostrados en la siguiente tabla de verdad:

	p	q	$p \wedge q$
caso 1:	`True`	`True`	`True`
caso 2:	`True`	`False`	`False`
caso 3:	`False`	`True`	`False`
caso 4:	`False`	`False`	`False`

Solo en el caso 1, la S-expresión $p \wedge q$ es cierta, porque tanto p como q son ciertas, en los demás casos una o ambas expresiones son falsas, por lo que el valor final deberá ser falso.

Para construir una adecuada definición efectiva en `Python`, se debe observar el comportamiento en cada uno de los posibles casos mostrados en la tabla de verdad, para lo cual hagamos `p` $\leftarrow\!\shortmid p$ y `q`$\leftarrow\!\shortmid q$.

Sobre la expresión booleana: Se observa que tanto `p` como `q` son proposiciones, por lo que no es necesario hacer alguna evaluación para obtener su valor de verdad así, si sabemos que `p` es una proposición lógica:

```
if p == True:
    <caso-verdadero>
else:
    <alternativa>
```

es equivalente a:

```
if p:
    <caso-verdadero>
else:
    <alternativa>
```

casos 1 y 2: Cuando `p`↤ `True` , el valor devuelto coincide con el valor de `q`. Como el valor de `q` ya ha sido ligado, el costo computacional es constante. Observa los casos 1 y 2 de la tabla de verdad.

caso2 3 y 4: Cuando `p`↤ `False`, el valor devuelto es `False`, sin importar el valor de `q`. Este comentario corresponde a los casos 3 y 4 de la tabla de verdad.

Así ahora podemos enunciar una nueva definición en base al comportamiento esperado del procedimiento y sus efectos:

> «La conjunción `y` de dos proposiciones `p` y `q`, depende del valor de `p`. Si `p` es `True`, entonces coincide con el valor de `q`; de otro modo será `False` ».

La traducción ahora al lenguaje `Python` es mucho más sencilla y directa:

Código 1.7: *La conjunción*

```
1 def y(p:bool, q:bool)->bool:
2     """ Conjunción
3     Devuelve True cuando ambas proposiciones son True,
4     devuelve False en cualquier otro caso.
5     """
6     if p:
7         return q
8     else:
9         return False
```

Al considerar las constantes `True` y `False`, debemos observar que sin importar los valores de p y q:

1. $p \wedge$ `True` $\mapsto p$.
2. $p \wedge$ `False` $\mapsto$ `False`.
3. $p \wedge q$ es equivalente a $q \wedge p$.

Ejemplo 1.20

Para probar el comportamiento del programa de la conjunción `y(p,q)` utilizaremos la parte de interacciones del IDE de `Python`.

```
>>> y(True, True)
True
>>> y(True, False)
False
>>> y(False, True)
False
>>> y(False, False)
False
>>>
```

Definición 1.6.6 -- Disyunción. La disyunción de dos expresiones booleanas p y q se escribe $p \vee q$ y tiene valor de verdad `True`, si ocurre que p o q o incluso ambas son `True`; pero si ambas proposiciones son `False`, entonces la disyunción $p \vee q$ es `False`.

El comportamiento antes descrito se observa en la tabla de verdad de la disyunción.

p	q	$p \vee q$
`True`	`True`	`True`
`True`	`False`	`True`
`False`	`True`	`True`
`False`	`False`	`False`

La definición formal en una λ-expresión es $\vee \leftarrow \lambda p, q \bullet$ `True` si p, q.

Ejemplo 1.21

Los siguientes ejemplos ilustran el uso de la disyunción en el lenguaje natural:

- «La clave es correcta o es superusuario».
- «La comida tiene sal o el agua es de piña».

Los siguientes ejemplos utilizan la disyunción en matemáticas:

- $(15 < 2^2) \vee (4^3 > 500^2)$.
- $(34 + 56 \leq 56^2) \vee (\frac{34}{2} > 28)$

Al considerar las constantes `True` y `False`, debemos observar que sin importar los valores de p y q:

1. $p \vee$ `True` $\mapsto$ `True`.
2. $p \vee$ `False` $\mapsto p$.
3. $p \vee q$ es equivalente a $q \vee p$.

Para programar la la disyunción, observa que cuando la proposición `p` es `True`, siempre el valor calculado es `True`; pero cuando `p` es `False`, el valor calculado de la disyunción es exactamente el mismo que el valor de la proposición `q`.

Código 1.8: *La disyunción*

```
def o(p:bool, q:bool)->bool:
    """ Disyunción de dos proposiciones:
    Devuelve True cuando al menos una de las proposiciones es True,
    si ambas proposiciones son False, entonces devuelve False.
    """
    if p:
        return True
    else:
        return q
```

Ejemplo 1.22

Probar el comportamiento de la disyunción `o`:

```
>>> o(True, True)
True
>>> o(True, False)
True
>>> o(False, True)
True
>>> o(False, False)
False
>>>
```

Definición 1.6.7 -- Disyunción exclusiva. La disyunción exclusiva de las proposiciones p y q se escribe $p \oplus q$ y es `True` cuando exactamente una de las dos proposiciones ya sea p o q pero no ambas es `True`; y es `False` en cualquier otro caso.

El comportamiento de la disyunción exclusiva puede observar en la siguiente tabla de verdad. La disyunción exclusiva es `False` si ambas proposiciones tienen el mismo valor de verdad. Cuando las proposiciones p y q tengan diferente valor de verdad, entonces el la disyunción exclusiva será `True`.

p	q	$p \oplus q$
True	True	False
True	False	True
False	True	True
False	False	False

Para calcular el valor del predicado $p \oplus q$, es bueno observar la tabla de verdad, donde puedes descubrir que si $p \mapsto$ `True`, $p \oplus q$ produce un valor equivalente a $\neg q$. Por otro lado, cuando $p \mapsto$ `False`, entonces $p \oplus q$ tendrá el mismo valor que q.

p	q	$\neg q$	$p \oplus q$
True	True	False	False
True	False	True	True
False	True	$\longrightarrow$	True
False	False	$\longrightarrow$	False

La definición formal en una λ-expresión es:

$$\oplus \mapsto \lambda p, q \bullet \neg q \text{ si } p,\ q.$$

Observa esto en la descripción de la definición en el código siguiente:

Código 1.9: *Disyunción exclusiva*

```
def ox(p:bool, q:bool)->bool:
    """ Disyunción exclusiva:
    Devuelve True cuando ambas proposiciones tienen valor diferente.
    Cuando las proposiciones coinciden en su valor, devuelve False.
    """
    if p:
        return neg(q)
    else:
        return q
```

Ejemplo 1.23

La disyunción exclusiva ox.

```
>>> ox(True, True)
False
>>> ox(True, False)
True
>>> ox(False, True)
True
>>> ox(False, False)
False
>>>
```

En el lenguaje cotidiano puede ser un poco difícil distinguir si una proposición se trata de una disyunción o de una disyunción exclusiva. Considera las siguientes expresiones:

«La ensalada sabe bien con nueces o con almendras». En este ejemplo si la ensalada tiene tanto nueces como almendras, entonces la ensalada sigue sabiendo bien, y no hay problema en que lleve solamente uno de esos dos ingredientes. Claramente se trata de una disyunción.

«Una ración diaria de granos se cumple con frijoles o con arroz». En este otro ejemplo, aunque es posible hacer un platillo que tenga tanto frijoles como arroz, lo adecuado es que solamente lleve uno de esos dos ingredientes, y no los dos, por lo que es mejor considerar esta frase como una disyunción exclusiva, pero tampoco pasa algo si se combinan esos ingredientes. Una posible solución es expresarlo de manera diferente como «Una ración diaria de granos se cumple *bien* con frijoles *o bien* con arroz».

«Una taza de café se disfruta con azúcar o sin azúcar». Claramente no es posible agregar azúcar y no agregar azúcar al mismo tiempo a una taza de café, por lo que se trata de una disyunción exclusiva.

1.6.4 Implicación y doble implicación

Definición 1.6.8 -- Implicación. La **implicación** de q a causa de p se escribe $p \to q$ siempre es `True` a menor que p sea `True` y q sea `False`.

Se puede interpretar de diferentes maneras, como «p implica q»; «si p, entonces q»; «se requiere p para q»; entre otras.

Debido a que la implicación muestra una relación causal, cuando se escribe $p \to q$, la proposición a la izquierda del símbolo $\to$ recibe el nombre de **antecedente** o **causa**, mientras que la proposición a la derecha del símbolo se llama **consecuente** o **efecto**.

La tabla de verdad muestra que el único valor `False` se obtiene cuando el antecedente es `True` y el consecuente es `False`. Esto indica que la proposición q no es una consecuencia lógica de p.

p	q	$p \to q$
`True`	`True`	`True`
`True`	`False`	`False`
`False`	`True`	`True`
`False`	`False`	`True`

La definición formal en una λ-expresión es:

$$\to\!\leftarrow \lambda p, q \bullet q \text{ si } p, \texttt{True}.$$

Ejemplo 1.24

La frase «si Alondra cumple sus promesas, entonces es de fiar» constituye una expresión condicional que involucra dos proposiciones: «Alondra cumple sus promesas» que es el antecedente de la implicación, por otro lado la proposición «[Alondra] es de fiar» es el consecuente.

La frase «si Arturo no duerme, entonces está estresado», observa aquí que si el consecuente «[Arturo] está estresado» es `True`, puede ser por otras causas que no sean precisamente el antecedente propuesto, por lo que la implicación es `True`.

Los valores de la tabla de verdad de la condicional, establecen que p es una condición necesaria para que ocurra q, de manera resaltable significa que de no ocurrir p, y sin importar el valor de la proposición q, se tiene que $p \to q \mapsto$ `True`. Al analizar una condicional el antecedente ocurre primero porque es la causa y el consecuente después porque es el efecto, eso le da un sentido temporal.

En una expresión condicional, hay 3 actores:

1. El antecedente o causa, es la sentencia a la izquierda del símbolo condicional $\to$.
2. El consecuente o efecto, es la sentencia a la derecha del símbolo condicional.
3. La **relación condicional** es un pensamiento que asocia el antecedente con el consecuente en la forma **si** ⟨*antecedente*⟩, **entonces** ⟨*consecuente*⟩. Cuando se ha valorado la expresión condicional, se crea una nueva una sentencia completa, indivisible con algún valor de verdad. En las condicionales, se parte del antecedente y consecuente para crear la relación condicional *si-entonces*, en ocasiones es posible iniciar desde una relación condicional, para luego determinar su antecedente y consecuente para su análisis por separado; sin embargo esto no es posible en todos los casos, en ocasiones no es posible verificar el valor de verdad del antecedente o del consecuente, por ejemplo «Si tu padre viviera, estaría orgulloso de ti», claramente no es posible revivir al padre, por lo que tampoco podremos saber si está o no orgulloso.

La programación de la condicional se llamará `impl` que tiene la función de la implicación. Se requieren dos proposiciones que llamaremos `p` y `q`. Usaremos la expresión condicional `if` sobre la proposición `p`, para separar las dos posibles valoraciones de la nueva implicación, observando el valor de `q`.

Código 1.10: *Implicación*

```
def impl(p:bool, q:bool)-> bool:
    """ La implicación
    Devuelve True ya sea que el antecedente es False, o bien cuando
    el antecedente es True y el consecuente es True.
    Devuelve False cuando el antecedente es False y el consecuente es False.
    """
    if p:
        return q
    else:
        return True
```

Se comprueba los cuatro posibles casos y se comprueba con la tabla de verdad.

```
>>> impl(True, True)
True
>>> impl(True, False)
False
>>> impl(False, True)
True
>>> impl(False, False)
True
>>>
```

Definición 1.6.9 -- Doble implicación. La doble implicación o **bicondicional** de p y q se escribe $p \leftrightarrow q$ y tiene valor `True` si los valores de verdad de p y q coinciden, de otro modo será `False`.

La tabla de verdad de la bicondicional es:

p	q	$p \leftrightarrow q$
True	True	True
True	False	False
False	True	False
False	False	True

Al observar la tabla de verdad, caemos en cuenta de que cuando p ocurre [primeros dos renglones] el valor de $p \leftrightarrow q$ coincide con q, y cuando p no ocurre [últimos dos renglones] el valor de $p \leftrightarrow q$ es precisamente $\neg q$, lo que podemos aprovechar para formalizar la definición

$$\leftrightarrow\!\!\leftarrow \lambda p, q \bullet q \text{ Si } p \text{ , } \neg q.$$

Código 1.11: *Doble implicación*

```
def ssi(p:bool, q:bool)-> bool:
    """ Doble condicional
    Devuelve True cuando ambas proposiciones tienen el mismo valor
    y devuelve False si las proposiciones son de diferente valor.
    """
    if p:
        return q
    else:
        return neg(q)
```

El bicondicional es un caso más estricto que el condicional, ya que ahora se considera `False` el caso cuando el antecedente es `False` y el consecuente es `True`:

```
>>> ssi(True, True)
True
>>> ssi(True, False)
False
>>> ssi(False, True)
False
>>> ssi(False, False)
True
>>>
```

1.6.5 Precedencia de los operadores lógicos

Frecuentemente se tienen expresiones como $p \wedge q \rightarrow r$, que se puede entender como $p \wedge (q \rightarrow r)$ o bien $(p \wedge q) \rightarrow r$, lo que podría ocasionar errores en los cálculos. Por esta razón se ha establecido el siguiente criterio para el orden de precedencia:

Precedencia	Símbolo matemático	Identificador `Python`
1	$\neg$	`neg`
2	$\wedge$	`y`
3	$\vee, \oplus$	`o`, `ox`
5	$\rightarrow$	`impl`
6	$\leftrightarrow$	`ssi`

Así la expresión $p \wedge q \rightarrow r$ se debe entender $(p \wedge q) \rightarrow r$ ya que la conjunción se hace primero que la implicación. En expresiones donde hay operadores de la misma precedencia, estos se resuelven de izquierda a derecha.

Normalmente se utilizan paréntesis para indicar qué operación se debe hacer primero. Cuando se utilizan paréntesis, las operaciones se resuelven desde el más anidado al menos anidado, y de izquierda a derecha.

Ejemplo 1.25

La expresión $p \leftrightarrow q \wedge r \rightarrow \neg r \vee \neg p$ se debe entender como:

1. Las negaciones: $p \leftrightarrow r \wedge q \rightarrow (\neg r) \vee (\neg p)$
2. La conjunción: $p \leftrightarrow (q \wedge r) \rightarrow (\neg r) \vee (\neg p)$
3. La disyunción: $p \leftrightarrow (q \wedge r) \rightarrow ((\neg r) \vee (\neg p))$
4. La implicación: $p \leftrightarrow ((q \wedge r) \rightarrow ((\neg r) \vee (\neg p)))$
5. La bicondicional: $(p \leftrightarrow ((q \wedge r) \rightarrow ((\neg r) \vee (\neg p))))$

1.7 Equivalencia lógica

Decimos que dos predicados `A` y `B` son **lógicamente equivalentes**, si tienen el mismo valor ante los mismos valores de sus variables.

Ejemplo 1.26

Digamos que $A \leftarrow\!\!\rightarrow p \wedge q$ y $B \leftarrow\!\!\rightarrow q \wedge p$. Calculemos ahora la tabla de verdad de cada predicado.

		A ↓	B ↓
p	q	$p \wedge q$	$q \wedge p$
True	True	True	True
False	True	False	False
True	False	False	False
False	False	False	False

Notamos que las columnas marcadas con flechas son iguales, haciendo énfasis en que se debe preservar el orden de los renglones; como son iguales podemos decir que $p \wedge q$ y $q \wedge p$ son lógicamente equivalentes. Esto hace que la operación $\wedge$ sea conmutativa.

Verificar la equivalencia lógica de dos predicados como los mostrados en el ejemplo 1.26, es un procedimiento que se realiza en 3 pasos:

1. Generar los casos de prueba
2. Para cada uno de los predicados, generar el vector de resultados.
3. Determinar la equivalencia. Se deben comparar ambos vectores de resultados, únicamente cuando son iguales, los predicados son lógicamente equivalentes.

Casos de prueba

Los casos de prueba dependen de la aridad del predicado. En predicados de aridad 1, solamente se requieren dos casos de prueba, cuando la variable es `True` y cuando es `False`. Así la lista de casos de prueba es:

```
[[True], [False]]
```

Un predicado unario debe ser evaluado con los valores de verdad en cada instancia de prueba.

Los predicados de aridad 2 requieren 4 casos de prueba, esto porque cada una de las 2 variables puede tener 2 valores, esto hace que el total de casos de prueba sea $2 * 2 = 4$, estos casos son:

```
[[True, True], [True, False], [False, True], [False, False]]
```

Al evaluar un predicado de aridad 2, es necesario que cada caso de prueba provea los valores que serán asignados a cada una de las variables. Por ejemplo al evaluar el predicado $p \wedge q$ con cada una de las diferentes formas de entrada de argumentos:

	p	q		$p \wedge q$
Caso de prueba: `[True, True]`	`True`	`True`	$\mapsto$	`True`
Caso de prueba: `[True, False]`	`True`	`False`	$\mapsto$	`False`
Caso de prueba: `[False, True]`	`False`	`True`	$\mapsto$	`False`
Caso de prueba: `[False, False]`	`False`	`False`	$\mapsto$	`False`

Vector de resultados

Al considerar de arriba hacia abajo solo los resultados se obtiene una lista de valores booleanos de longitud 2^a, donde a es la aridad del predicado. En el caso de un predicado de aridad 3, se obtendrá un vector de longitud $2^3 = 8$ valores booleanos. En el caso de las evaluaciones de $p \wedge q$, donde $\wedge$ es de aridad 2, se obtiene un vector de longitud 2^2 de los valores booleanos:

```
[True, False, False, False];
```

luego es necesario obtener el vector de resultados del segundo predicado con las mismas entradas y en el mismo orden:

q	$\wedge$	p	
`True`	$\wedge$	`True`	$\mapsto$ `True`
`False`	$\wedge$	`True`	$\mapsto$ `False`
`True`	$\wedge$	`False`	$\mapsto$ `False`
`False`	$\wedge$	`False`	$\mapsto$ `False`

para obtener su vector de resultados:

```
[True, False, False, False].
```

Determinar la equivalencia

Ambos vectores de resultados deben ser comparados tomando en cuenta las entradas a pares, con el fin de calcular la igualdad; al hacer esto la equivalencia lógica quedará verificada.

En el ejemplo se deben comparar:

```
p ∧ q:  [True,  False,  False,  False ]
          ↕       ↕       ↕       ↕
q ∧ p:  [True,  False,  False,  False ]
```

Como ambos vectores de resultados son iguales, se determina que los predicados son lógicamente equivalentes.

Procedimiento para verificar la equivalencia lógica

Un procedimiento automático que evalúa la equivalencia lógica de dos predicados de la misma aridad se puede construir en dos etapas:

1. El generador de vector de resultados. Este procedimiento recibe como entrada un predicado y devuelve un vector de resultados. Para esto se deben realizar tres pasos:
   a) Obtener la aridad del predicado.

b) Generar los casos de prueba. Los casos de prueba consisten en una lista que contiene cero o más listas, cada una de ellas contiene una combinación diferente de valores de verdad, que deben ser asociados a cada uno de los parámetros del predicado a ser evaluado. Por ejemplo, si al evaluar un predicado $\Psi(p,q,r)$ que es de aridad 3 y se tiene el caso de prueba `[True, False, True ]`, en esa prueba $p \longleftarrow$ `True`; $q \longleftarrow$ `False` y $r \longleftarrow$ `True`.

c) Construir el vector de resultados.

2. El verificador de equivalencia lógica para dos predicados. Requiere dos predicados. Devuelve `True` si se determina la igualdad de los vectores de resultados y devuelve `False` si los vectores de resultados son diferentes.

Código 1.12: *Generador de argumentos lógicos*

```
from inspect import signature

def aridad(fun):
    """
    Obtiene la aridad de una función.
    """
    return len(signature(fun).parameters)


def casosPrueba(n, res=None):
    """
    Genera una lista de casos de prueba para predicados
    de aridad n.
    """
    if res == None: res=[[]]
    if n==0:
        return res
    else:
        res1 = list(map(lambda t:[True]+t, res))
        res2 = list(map(lambda t:[False]+t, res))
        res = res1 + res2
        return casosPrueba(n-1, res)

def vectorResultados(Pred):
    """
    Genera un vector de resultados para la evaluación
    del predicado Pred.
    """
    ar = aridad(Pred)
    Lvals = casosPrueba(ar)
    V = list(map(lambda t:Pred(*t), Lvals))
    return V

def logEqv(Pred1, Pred2):
    """
    Verifica la equivalencia lógica de los predicados
    Pred1 y Pred2.
    """
    v1 = vectorResultados(Pred1)
    v2 = vectorResultados(Pred2)
    return v1 == v2
```

Ejemplo 1.27

Para generar los casos de prueba para predicados de aridad 2 y luego de aridad 3:

```
>>> casosPrueba(2)
[[True, True], [True, False], [False, True], [False, False]]
>>> casosPrueba(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]]
>>>
```

El procedimiento primitivo `map`, permite aplicar un mismo procedimiento a una lista de argumentos.

`map(`⟨*callable*⟩`,` ⟨*iterable*⟩ `...)` ↦iterable

`funcion : callable` *es un objeto que se puede invocar*`.iterable : list | tuple | range.`

Aplica la ⟨*funcion*⟩ usando los elementos de las listas para dar valores a los parámetros de la función. Debe haber tantos iterables como aridad de la función y todos los iterables deben ser de la misma longitud. El resultado es una lista encapsulada que contiene cada resultado de la evaluación de ⟨*funcion*⟩ en el orden en que se aplicaron.

Ejemplo 1.28

Si utilizamos p para representar la proposición «Arturo come carne»; usamos q para representar «Arturo come verduras» y r representa «Arturo toma agua», las siguientes expresiones son lógicamente equivalentes:

- «Arturo come carne o Arturo come verduras y toma agua», que es representada por la expresión simbólica $p \vee (q \wedge r)$ ya que la conjunción tiene mayor precedencia que la disyunción.
- «Arturo come carne o verduras y Arturo come carne o toma agua». Que es simbólicamente representada por $(p \vee q) \wedge (p \vee r)$.

La demostración de esta equivalencia queda clara con la tabla de verdad en el siguiente ejemplo.

Ejemplo 1.29

Se verifica la equivalencia lógica de dos predicados de aridad 3. En términos convencionales, uno de los predicados corresponde a $p \vee (q \wedge r)$, que se identificará con `P1` y el segundo corresponde a $(p \vee q) \wedge (p \vee r)$, identificado con `P2` en las interacciones. Esta equivalencia es una de las leyes distributivas en el álgebra de proposiciones. Por cuestión de espacio, en la tabla de verdad se utiliza el símbolo `T` para `True` y `F` para `False`.

				↓			↓
p	q	r	$q \wedge r$	$p \vee (q \wedge r)$	$p \vee q$	$p \vee r$	$(p \vee q) \wedge (p \vee r)$
T	T	T	T	T	T	T	T
T	T	F	F	T	T	T	T
T	F	T	F	T	T	T	T
T	F	F	F	T	T	T	T
F	T	T	T	T	T	T	T
F	T	F	F	F	T	F	F
F	F	T	F	F	F	T	F
F	F	F	F	F	F	F	F

```
>>> P1 = lambda p,q,r: o(p, y(q, r))
>>> P2 = lambda p,q,r: y(o(p, q), o(p, r))
>>> logEqv(P1, P2)
True
>>>
```

Ejercicios

1.- Decide si las siguientes proposiciones son simples o compuestas:
   a) *Ejemplo:* «El sol es una estrella» es una proposición simple.
   b) «Es posible hacer negocios si la tienda está abierta».
   c) «Si el río suena, entonces el río lleva agua».
   d) «Hay argumentos insuficientes».
   e) «El usuario no pagó la mensualidad».
   f) «El usuario no tiene acceso si no paga la mensualidad».

2.- En las siguientes interacciones con `Python` describe cuál ha sido el error y menciona cómo se puede corregir.

```
>>> 34 = False
      34 = False
      ^
SyntaxError: cannot assign to literal
>>> nunca es tarde = False
      nunca es tarde = False
                ^
SyntaxError: invalid syntax
>>>
```

3.- Los siguientes predicados han sido escritos utilizando la notación convencional de matemáticas. Tradúcelos en lenguaje de programación `Python`, no es necesario evaluar la expresión:
   a) *Ejemplo:* $10x^2 = 2zy$ se traduce como `10*x**2 == 2*z*y`.
   b) $4x + 2y > 30$
   c) $3x^2 + 2x = 0$
   d) $21x^2 - 13y + 2z < y^3$
   e) $(p \wedge q) \leftrightarrow (p \vee (\neg q \oplus p))$

4.- Los siguientes predicados usan la notación de `Python`. Tradúcelos al lenguaje convencional de matemáticas, no es necesario evaluar la expresión:
   a) *Ejemplo:* `lambda x:x**2` se traduce como $\lambda x \cdot x^2$.
   b) `lambda a: a*3 == a+6`
   c) `lambda x,y: x**2 >= x*y`
   d) `lambda b,c: b+2*c == b**2 - c`
   e) `lambda x,y: (x*y)**2 + x**2 + 5 == 0`

5.- Digamos que `p` se refiere a la expresión «Come frutas», y `q` se refiere a la expresión «Come carne». Expresa en `Python` de manera simbólica, las siguientes expresiones. Utiliza las funciones creadas en el capítulo:
   a) *Ejemplo:* «Si come frutas entonces come carne» se escribe `impl(p,q)` en `Python` (página 39).
   b) «Come frutas o come carne».
   c) «Come carne y no come frutas».
   d) «Ni come carne, ni come verduras».

6.- El predicado $p \uparrow q$ se conoce como el operador Sheffer [Rau10] y sigue el comportamiento:

$$p \uparrow q \mapsto \begin{cases} \neg q & \text{si } p. \\ \texttt{True} & \textit{eoc.} \end{cases}$$

Escribe la tabla de verdad del operador Sheffer y escribe el procedimiento en `Python` para `sheffer`. En el siguiente código sustituye la instrucción `pass` por la instrucción adecuada para modelar el operador Sheffer.

```
def sheffer(p:bool, q:bool)-> bool:
    """
    sheffer(p,q)-> bool
    p: bool
    q: bool
    """
    pass  # Escribe aquí tu código
```

Comprueba tu programa haciendo las siguientes interacciones

```
>>> sheffer(True, True)
False
>>> sheffer(True, False)
True
>>> sheffer(False, True)
True
>>> sheffer(False, False)
True
>>>
```

7.- El predicado $p \downarrow q$ [Rau10] se conoce como operador Peirce y sigue el comportamiento:

$$p \downarrow q \mapsto \begin{cases} \texttt{True} & \text{si } p. \\ q & eoc. \end{cases}$$

Escribe la tabla de verdad del operador Peirce y escribe el procedimiento en `Python` para `peirce`. En el siguiente código sustituye la instrucción `pass` por la instrucción adecuada para modelar el operador Peirce.

```
def peirce(p:bool, q:bool)-> bool:
    """
    peirce(p,q)-> bool
    p: bool
    q: bool
    """
    pass
```

Comprueba tu programa haciendo las siguientes interacciones

```
>>> peirce(True, True)
True
>>> peirce(True, False)
True
>>> peirce(False, True)
True
>>> peirce(False, False)
False
>>>
```

8.- Verifica las siguientes equivalencias lógicas mediante una tabla de verdad [el símbolo $\equiv$ significa 'equivale a'] y luego traduce la expresión al lenguaje `Python` y verifica la equivalencia lógica utilizando la función `logEqv` (página 43):

a) *Ejemplo:* Leyes de idempotencia:

1) $p \vee p \equiv p$
2) $p \wedge p \equiv p$

$\downarrow$	$\downarrow$	$\downarrow$
p	$p \vee p$	$p \wedge p$
`True`	`True`	`True`
`False`	`False`	`False`

Para verificarlo, creamos una función anónima para cada predicado. Observa que estos predicados son de aridad 1.

```
>>> logEqv(lambda p:p, lambda p: o(p,p))
True
>>> logEqv(lambda p:p, lambda p: y(p,p))
True
>>>
```

b) Leyes asociativas:
1) $(p \vee q) \vee r \equiv p \vee (q \vee r)$
2) $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$

c) Leyes conmutativas:
1) $p \vee q \equiv q \vee p$
2) $p \wedge q \equiv q \wedge p$

d) Leyes distributivas:
1) $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
2) $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$

e) Leyes de identidad:
1) $p \vee \texttt{False} \equiv p$
2) $p \vee \texttt{True} \equiv \texttt{True}$
3) $p \wedge \texttt{False} \equiv \texttt{False}$
4) $p \wedge \texttt{True} \equiv p$

f) Leyes sobre complementos:
1) $p \vee \neg p \equiv \texttt{True}$
2) $p \wedge \neg p \equiv \texttt{False}$
3) $\neg\texttt{False} \equiv \texttt{True}$
4) $\neg\texttt{True} \equiv \texttt{False}$

g) Leyes de DeMorgan:
1) $\neg(p \vee q) \equiv \neg p \wedge \neg q$
2) $\neg(p \wedge q) \equiv \neg p \vee \neg q$

9.- Escribe una función en `Python` llamada `verifVR`, que verifica una función lógica ⟨*Pred*⟩. La verificación se hace comparando la evaluación de `Pred` (página 43), con un vector de resultados ⟨*vr*⟩ pasado como segundo argumento.
La función devuelve `True` si el vector de resultados es igual a `vr`. Cualquier diferencia ocasiona una respuesta `False`. En el siguiente código sustituye la instrucción `pass` por el código de la función.

```
1 def verifVR(Pred, vr:list)-> bool:
2     """
3     Verifica un vector de resultados
4     Pred: Función booleana
5     vr: list
6     """
7     pass # <- escribe aquí tu código
```

Por ejemplo, considera el predicado $\lambda p, q \bullet p \oplus q$, y queremos compararlo con el vector [`False`, `True`, `True`, `False`] que es el vector de resultados de la disyunción exclusiva (ver página 37).

```
>>> verifVR(lambda p,q:ox(p,q), [False, True, True, False])
True
>>>
```

10.- La siguiente tabla de verdad es acerca de un procedimiento Φ de aridad tres. Debes construir Φ mediante la combinación de operadores lógicos, de tal modo que tenga el comportamiento de la tabla de verdad que se muestra enseguida.

p	q	r	$\Phi(p,q,r)$
True	True	True	False
True	True	False	True
True	False	True	True
True	False	False	False
False	True	True	True
False	True	False	True
False	False	True	True
False	False	False	False

Una vez que hayas definido Φ de acuerdo a la tabla de verdad, debes traducirlo al lenguaje `Python`, y luego comprobarlo al verificar la equivalencia lógica con $\Phi(p,q,r)$:

```
1 def Phi(p:bool, q:bool, r:bool)-> bool:
2     """
3     Debe tener el mismo comportamiento que en la tabla de verdad.
4     """
5     pass # <- escribe aquí el código
```

Para comprobar tu definición puedes utilizar el programa propuesto en el ejemplo anterior y guiarte de la siguiente interacción:

```
>>> verifVR(Phi, [False, True, True, False, True, True, True, False])
True
>>>
```

Cuantificadores 2

En ocasiones es necesario hacer operaciones lógicas con una colección de proposiciones, por ejemplo se desea determinar el valor de verdad para la conjunción de las proposiciones siguientes:

1. $p_1 \leftarrowtail 10 > 9$
2. $p_2 \leftarrowtail 20 > 9$
3. $p_3 \leftarrowtail 30 > 9$
4. $p_4 \leftarrowtail 40 > 9$
5. $p_5 \leftarrowtail 50 > 9$

Al utilizar la conjunción tal como ha sido definida en la sección 1.6.5 en la página 34, surge el problema de que la definición utiliza solamente dos operandos y en el ejemplo hay más de dos.

Claro que se puede resolver el problema haciendo:

$$((((p_1 \wedge p_2) \wedge p_3) \wedge p_4) \wedge p_5$$

Esta situación ejemplifica la necesidad de tener expresiones lógicas que permitan extender el uso de la conjunción y de la disyunción para utilizar más de dos proposiciones lógicas como operandos.

2.1 Extensión de la conjunción y disyunción

Es posible extender el concepto de conjunción [y disyunción], al aplicar el mismo predicado, ya sea de conjunción o de disyunción, a una secuencia de longitud indeterminada de expresiones lógicas, la cantidad de expresiones lógicas puede ser incluso cero.

En el caso de la **conjunción extendida** se trata de crear una definición efectiva que actúe con cero o más expresiones lógicas y que a la salida ofrezca un valor de verdad `True`, si todas las expresiones lógicas han sido evaluadas como `True`; y devuelve `False`, si al menos una de ellas se ha evaluado como `False`.

La conjunción extendida de k proposiciones se escribe:

$$\bigwedge_{i=1}^{k} p_i = p_1 \wedge p_2 \wedge \cdots \wedge p_k,$$

Los únicos casos en que la conjunción extendida es `True`, ocurren cuando no hay proposiciones que evaluar, o bien cuando todas las expresiones lógicas son `True`.

Para obtener la evaluación general, utilizaremos un procedimiento llamado `Y`, que tomará una lista de proposiciones [de longitud indeterminada] que llamaremos `Lprop`, y que considerará tres posibles situaciones:

1. Si no hay proposiciones que analizar en `Lprop`, entonces se devuelve `True`.
2. Si la primera proposición en `Lprop` es `True`, entonces se recursa el procedimiento `Y`, con el resto de las proposiciones como nueva lista de argumentos.
3. En cualquier otro caso, entonces el valor final es `False`.

Antes de escribir el código fuente de la conjunción extendida, escribiremos dos funciones que serán muy útiles en adelante, se trata de `car` que devuelve el primer elemento de una lista no vacía y `cdr` que devuelve una lista con el resto de los elementos de la lista no vacía excepto el primero[1].

Es importante asegurarse de que la lista no esté vacía, porque de otro modo será causa de un error que detiene la ejecución del programa, ya que una lista vacía no tiene primer elemento, y por su puesto tampoco tiene un resto de elementos. En una notación textual, podemos hacer referencia a una lista L no vacía como

$$\langle \alpha_0 | L' \rangle$$

donde α_0 es el primer elemento y L' es la lista que contiene al resto de los elementos de L excepto el primero.

Ejemplo 2.1

Supongamos que $L \leftarrow \langle 2, 6, 4, 9, 3, 7 \rangle$. Al representar L como una lista no vacía $\langle \alpha_0 | L' \rangle$, $\alpha_0 \mapsto 2$ y $L' \mapsto \langle 6, 4, 9, 3, 7 \rangle$.

Código 2.1: *car y cdr*

```
1  def car(L:list):
2      """ car Content of Address part of Register [1950s]
3      Devuelve el primer elemento de una lista no vacía.
4      """
5      return L[0]
6
7  def cdr(L:list)-> list:
8      """ cdr Content of Decrement part of Register [1950s]
9      Devuelve una lista con todos excepto el primer
10     elemento de una lista no vacía.
11     """
12     return L[1:]
```

El código para la disyunción de una secuencia no determinada de proposiciones.

[1] *The origin of CAR and CDR in LISP* en `https://www.iwriteiam.nl/HaCAR_CDR.html`

Código 2.2: *La conjunción extendida*

```
1  def Y(*Lprop:list)->bool:
2      """
3      Disyunción extendida
4      Recibe una lista no determinada de proposiciones.
5      """
6      if Lprop == ():
7          return True
8      elif car(Lprop):
9          return Y(*cdr(Lprop))
10     else:
11         return False
```

Siempre es mejor probar las funciones hechas. Aquí se prueba el predicado de aridad múltiple con cero o más parámetros.

```
>>> Y()
True
>>> Y(True)
True
>>> Y(True, True, True, False, True)
False
>>> Y(True, True, True, True, True)
True
>>>
```

Una lista no determinada de parámetros se logra anteponiendo $\star$ al nombre del parámetro. En el siguiente ejemplo la función `items` recibe una lista indeterminada, que es un objeto de la clase `tuple`. Nota tipográfica: La posición vertical del símbolo $\star$ no es relevante, lo importante es que se coloque antes del identificador.

```
def items(*L):
    print(L)
    print(type(L))

>>> items('a','b',True,6)
('a', 'b', True, 6)
<class 'tuple'>
>>>
```

Hay que notar que en la línea 8 del código 2.2 de la conjunción extendida, dentro de la expresión condicional, la cláusula `elif car(Lprop):` permite continuar la evaluación de la lista de proposiciones `Lprop`, cuando la primera proposición sea `True`, es decir el `car(Lprop)`. `Lprop` debe estar conformado de proposiciones, por lo que `car(Lprop)` se refiere a la primera de ellas y su valor booleano determina si se continúa la evaluación aplicando el predicado `Y` al resto de proposiciones, o bien se continúa con la clausula `else`.

Ejemplo 2.2

La conjunción extendida puede recibir cero o más proposiciones:

```
>>> temp10 = True
>>> esDiciembre = True
>>> tarjetaPagada = False
>>> Y(temp10, esDiciembre, tarjetaPagada)
False
>>> Y(temp10, esDiciembre, neg(tarjetaPagada))
True
>>>
```

La **disyunción extendida** se define de manera similar que la conjunción extendida. La idea central es determinar `True` cuando al menos una expresión lógica se evalúa como `True`, sin importar cuál de ellas, o cuántas de ellas han sido `True`.

En todos aquellos casos en los que al menos una expresión lógica es `True`, la disyunción extendida es `True`; mientras que el único caso en que la disyunción extendida es `False`, es cuando todas las expresiones lógicas han sido evaluadas como `False`.

En la notación convencional de matemáticas, la disyunción extendida de una secuencia de proposiciones se puede escribir como:

$$\bigvee_{i=1}^{k} p_i = p_1 \vee p_2 \vee \cdots \vee p_k,$$

donde $p_1, p_2, \ldots, p_k$ son proposiciones.

Código 2.3: *La disyunción extendida*

```
def O(*Lprop:list)->bool:
    """
    Conjunción extendida
    Recibe una lista no determinada de proposiciones.
    """
    if Lprop == ():
        return False
    elif car(Lprop):
        return True
    else:
        return O(*cdr(Lprop))
```

Ejemplo 2.3

Interacciones para observar la disyunción extendida, considerando las variables definidas en el ejemplo 2.2.

```
>>> O(temp10, esDiciembre, tarjetaPagada)
True
>>>
```

2.2 Panorama general de los cuantificadores

Hasta ahora hemos trabajado con proposiciones y algunos operadores lógicos unarios y binarios, incluso hemos extendido la idea de la conjunción y disyunción para ser utilizados con un número no determinado de proposiciones. Sin embargo el poder expresivo de estas construcciones se ve limitado al no tener manera de generalizar el tipo de razonamiento [Fer09, p. 32] [Sai00, p. 157] [Gar01, p. 159], para extenderlo por ejemplo en expresiones como «nunca he logrado dormir 8 horas completas» o «en todas las familias, siempre hay un rebelde».

Las palabras «nunca», «siempre», «algunos», «al menos uno» y otras parecidas, requieren **cuantificadores**, que es una manera de establecer cuántos elementos de un dominio cumplen cierto predicado. Las expresiones como «nunca» o «siempre» son expresiones generales p universales; mientras que las expresiones como «hay uno» o «algunos» son expresiones particulares o existenciales.

En esta sección estudiaremos dos cuantificadores. El cuantificador universal que tiene símbolo $\forall$ y el cuantificador existencial que se escribe con el símbolo $\exists$. Haremos énfasis en un caso especial del cuantificador existencial y es la unicidad en la existencia, lo que se escribe con el símbolo $\exists!$

2.2.1 El predicado del cuantificador

El predicado de un cuantificador es una expresión booleana que contiene proposiciones, variables booleanas y operadores lógicos. Las proposiciones pueden incluir subexpresiones booleanas o no booleanas, unidas con operadores, de tal forma que al evaluar la expresión, siempre resulta un valor de verdad.

Un predicado puede expresarse en notación lambda, que se escribe con el formato habitual expresiones lambda [página 27], pero el resultado siempre debe ser un valor booleano. Este tipo de expresiones son una manera formal de expresar un predicado, porque hacen explícito tanto el nombre de las variables que son definidas dentro del predicado como las operaciones que se realizan

Ejemplo 2.4

El predicado $\lambda x \cdot 3x > 15$ es un predicado de aridad 1. La variable formal es x. El valor de x debe ser asociado con un valor numérico antes de la evaluación. El resultado de la operación es `False` o `True`. Así como ejemplos de evaluación $(\lambda x \cdot 3x > 15)(6) \mapsto$ `True` y $(\lambda x \cdot 3x > 15)(5) \mapsto$ `False`.

La expresión $\lambda x, y \cdot 4x = 2y$ es un predicado de aridad 2. Las variables formales son x y y. Se deben proporcionar dos valores para vincular las formales, el primero de ellos se asocia con x y el segundo valor debe asociarse a y, ambos valores deben ser numéricos. El resultado nuevamente es un valor booleano. También podemos ejemplificar la evaluación de este predicado como $(\lambda x, y \cdot 4x = 2y)(5, 8) \mapsto$ `False` y $(\lambda x, y \cdot 4x = 2y)(2, 4) \mapsto$ `True`.

Cuando se escriben expresiones con cuantificadores, los predicados usados se deben enunciar de manera singular y se prefiere escribirlos en forma positiva.

Ejemplo 2.5

Los siguientes ejemplos ilustran diferentes maneras de escribir nombres de predicados, unas son mejores que otras.

Es mejor	No es mejor	Razón
`comeCarne`(x)	`comenCarne`(x)	Se escribió en plural.
$\neg$`comeCarne`(x)	`noComeCarne`(x)	Se escribió en forma negativa.

Los cuantificadores se reconocen por un símbolo particular, ya sea $\exists$ o bien $\forall$. El predicado dentro del cuantificador utiliza el símbolo del cuantificador en lugar del símbolo λ.

Así, cuando se escribe un cuantificador universal, se escribe

$$\forall \langle formal \rangle \in \langle dominio \rangle, \cdots : \langle predicado \rangle, \ldots,$$

y cuando se escribe un cuantificador existencial, se escribe

$$\exists \langle formal \rangle \in \langle dominio \rangle, \cdots : \langle predicado \rangle, \ldots,$$

donde ,… significa que es permitido repetir el formato las veces que sea necesario.

Un predicado expresado en su notación lambda, no tiene valor de verdad, pues no se conocen los valores de las variables formales. Para determinar el valor de verdad del predicado se deben instanciar sus formales y los valores que son permitidos para los formales son proporcionados por el dominio del cuantificador.

2.2.2 El dominio de aplicación

Una de las partes más importantes de un cuantificador es el **dominio**, que en el contexto de cuantificadores es la lista de valores que una variable deberá tener.

En general, concepto de «dominio» es asociado con el concepto de «dominio del discurso» o «conjunto de valores admisibles» [CG15, p. 4], esto es porque comúnmente los elementos del dominio son diferentes y porque el procedimiento asociado solamente admite los elementos de este conjunto.

Cuando el dominio se usa en cuantificadores, los posibles valores que deben ser asociados a una variable se aplican en el orden en que fue definido el dominio.

Los valores del dominio se encierran entre llaves $\{\dots\}$, como en $\{3, 5, 2, 1\}$.

Los dominios ofrecen los posibles valores que debe tomar una variable definida en el predicado. El predicado asigna uno de los valores del dominio a su variable, se evalúa el predicado y se procede de acuerdo al valor de verdad obtenido.

Ejemplo 2.6

Digamos que x es una variable en el dominio $\{2, 4, 6, 8\}$. Esto significa que:

1. $x \hookleftarrow 2$, luego
2. $x \hookleftarrow 4$, luego
3. $x \hookleftarrow 6$, luego
4. $x \hookleftarrow 8$ y termina.

Se utiliza el orden en que fue definido el dominio. El resultado final no cambia si se utiliza un orden diferente.

2.3 Cuantificador universal

2.3.1 Definición del cuantificador universal

El cuantificador universal es un predicado general, esto es que todos los elementos de un dominio deben de cumplir el predicado, por eso se llama «para todo».

Definición 2.3.1 -- Cuantificador Universal. El cuantificador universal es un predicado que se denota

$$\forall x \in D : P(x),$$

donde $\forall$ es el cuantificador universal; D es el *dominio*; P es un predicado y x es parámetro formal que representa a cada uno de los elementos del dominio. El valor de verdad del cuantificador es `False` si $P(x) \mapsto$ `False` en alguno de los elementos de D, de otro modo será `True`.

Es bueno recordar que el predicado que se utilizará en un cuantificador universal, este debe describirse en singular y no en plural, esto es porque el predicado debe ser aplicado en cada elemento del dominio y no se aplica a todos de una vez.

Ejemplo 2.7

Las siguientes expresión pueden ser modeladas con cuantificadores universales:

1. «Todos los perros del grupo avanzado son obedientes».
   a) El dominio puede ser definido `grupoAvanzado` que presumiblemente tendrá algunos perros.
   b) El predicado puede ser $\lambda p \in$ `grupoAvanzado`$\bullet$`esObediente`(p), claro que hará falta definir cómo se determina si un perro p es obediente o no.

Así el predicado universal «Todos los perros de grupo avanzado son obedientes» se escribe:

$$\forall p \in \texttt{grupoAvanzado} : \texttt{esObediente}(p)$$

2. «Todos los días de la semana hay descuentos».
   a) El dominio puede ser $\texttt{diasSemana} \leftarrowtail \{dom, lun, mar, mie, jue, vie, sab\}$, haciendo referencia a cada uno de los días de la semana.
   b) El predicado lo podemos definir $\lambda d \in \texttt{diasSemana} \bullet \texttt{hayDescuento}(d)$, que debe ser una función booleana que devuelva `True` si en el día s hay descuentos y `False` si no lo hay.

   Así el predicado universal «Todos los días de la semana hay descuentos». se escribe:

$$\forall d \in \texttt{diasSemana} : \texttt{hayDescuento}(d)$$

Observa que el predicado se debe enunciar de manera singular, en los ejemplos anteriores se dice `esObediente` y no se dice `sonObedientes`; se dice `hayDescuento`, es equivocado decir `hayDescuentos`.

2.3.2 Transcripción de una expresión universal

Hay varias formas en las que se puede presentar un cuantificador universal. Es necesario leer con cuidado para determinar si la expresión se puede modelar mediante un cuantificador universal.

Palabras clave que denotan universalidad:

1. «Todos», por ejemplo en «Todos los juegos los ha ganado Enrique». Aquí el dominio pudiera ser $G \leftarrowtail$«los juegos» y el predicado $\lambda g \bullet \texttt{enriqueGana}(g)$, donde $\texttt{enriqueGana}(g) \mapsto \texttt{True}$ cuando Enrique gana el juego g y `False` en otro caso.

$$\forall g \in G : \texttt{enriqueGana}(g)$$

2. «Siempre», por ejemplo en «Siempre que llueve hace calor». Aquí el dominio puede ser $A \leftarrowtail$«Los días del año»; y el predicado es la implicación $\texttt{llueve}(d) \rightarrow \texttt{haceCalor}(d)$, donde d es un día del año.

$$\forall d \in A : \texttt{llueve}(d) \rightarrow \texttt{haceCalor}(d)$$

Entonces, para transcribir formalmente una expresión coloquial que hace referencia a un cuantificador universal, se deben distinguir y anotar los tres elementos del cuantificador:

1. El dominio de aplicación.
2. El predicado enunciado en forma singular.
3. El identificador que hace referencia a cada elemento del dominio.

2.3.3 Definición recursiva del cuantificador universal

El cuantificador universal involucra un único predicado P que debe ser evaluado en cada uno de los elementos de un dominio D. Suponiendo que $D \leftarrowtail \varnothing$ o bien $D \leftarrowtail \{x_0 | D'\}$, P debe ser evaluado con cada uno de los elementos en D. Un procedimiento efectivo debe proponer una estrategia recursiva para determinar el valor de verdad de una expresión con cuantificador. Observa la siguiente definición recursiva.

$$\forall x \in D : P(x) \mapsto \begin{cases} \texttt{True} & \text{si } D = \varnothing. \\ \forall x \in D' : P(x) & \text{si } P(x_0). \\ \texttt{True} & eoc. \end{cases}$$

El valor de verdad de $\forall x \in D : P(x)$ se obtiene de los siguientes casos:

1. Si no hay elementos en el dominio, el valor es `True`. Por su propia definición, ya que no se ha encontrado algún elemento `False`. Este es el caso base.
2. Si el dominio no es vacío, el dominio se puede representar como $\{x_0|D'\}$, y si $P(x_0) \mapsto$ `True`, se debe calcular el valor de verdad de una expresión universal similar, pero con dominio D'.
3. *eoc* («en otro caso»), es decir si no ocurre alguno de los dos casos anteriores, se determina `False`; este es otro caso base.

En `Python` podemos traducir este algoritmo recursivo como:

Código 2.4: *Cuantificador universal recursivo*

```
def paraTodo(P, D:list)-> bool:
    """ Cuantificador universal
    Versión recursiva
    Se verifica el predicado P en todos los elementos del dominio D.
    """
    if D == []:
        return True
    elif P(car(D)):
        return paraTodo(P, cdr(D))
    else:
        return False
```

Ejemplo 2.8

Demuestra en forma recursiva $\forall x \in \{22, 32, 42, 52\} : 2x > 40 \mapsto$ `False`. Enunciamos formalmente el predicado P y el dominio D con la sintaxis del cuantificador: $P \leftarrow \lambda x \cdot 2x > 40$ y $D \leftarrow \{22, 32, 42, 52\}$.

Ahora probamos el predicado en cada uno de los elementos del dominio:

$$\begin{array}{lll|l|l} \texttt{paraTodo}(P, D) & = \texttt{paraTodo}(\lambda x \cdot 2x > 40, \{22, 32, 42, 52\}) & x \leftarrow 22 & P(22) \mapsto \texttt{True} \\ & = \texttt{paraTodo}(\lambda x \cdot 2x > 40, \{32, 42, 52\}) & x \leftarrow 32 & P(32) \mapsto \texttt{True} \\ & = \texttt{paraTodo}(\lambda x \cdot 2x > 40, \{42, 52\}) & x \leftarrow 42 & P(42) \mapsto \texttt{True} \\ & = \texttt{paraTodo}(\lambda x \cdot 2x > 40, \{52\}) & x \leftarrow 52 & P(52) \mapsto \texttt{True} \\ & = \texttt{paraTodo}(\lambda x \cdot 2x > 40, \{\}) & D = \{\} & \\ & = \texttt{True}. & & \end{array}$$

Claro que al utilizar el programa, es necesario transcribir las expresiones respetando la sintaxis del lenguaje:

```
>>> paraTodo(lambda x:2*x>40, [22,32,42,52])
True
>>>
```

2.4 Cuantificador existencial

2.4.1 Definición del cuantificador existencial

El cuantificador existencial es un predicado que requiere el uso de un predicado y se requiere que al menos un elemento de cierto dominio cumpla el predicado, por eso se conoce como «existe un».

Definición 2.4.1 -- Cuantificador Existencial. Se denota

$$\exists x \in D : P(x),$$

donde $\exists$ es el cuantificador existencial; D, P y x son como antes. $\exists x \in D : P(x) \mapsto \texttt{True}$ si $P(x) \mapsto \texttt{True}$ en alguno los elementos de D; de otro modo será `False`.

Ejemplo 2.9

Las siguientes expresiones se modelan con cuantificadores existenciales:

1. «Hay un perro callejero que es amistoso».
   a) El dominio puede suponerse que es el conjunto de perros que viven en la calle, por lo que pensaremos llamarlo `perrosCallejeros`.
   b) El predicado puede ser $\lambda p \bullet \texttt{esAmistoso}(p)$, aunque hace falta definir por completo este predicado, que es `True` si un perro callejero p es amistoso y `False` en otro caso.

   Así el predicado existencial «Hay un perro callejero que es amistoso» se escribe:

   $$\exists p \in \texttt{perrosCallejeros} : \texttt{esAmistoso}(p)$$

2. «Al menos una persona del grupo saluda al llegar».
   a) El dominio puede suponerse que es el grupo de personas, por lo que se puede llamar simplemente `grupo`.
   b) El predicado puede ser $\lambda p \bullet \texttt{saluda}(p)$, que es `True` si la persona p saluda al llegar y `False` en otro caso.

   Así el predicado existencial «Al menos una persona del grupo saluda al llegar» se escribe:

   $$\exists p \in \texttt{grupo} : \texttt{saluda}(p)$$

Al igual que en el caso del cuantificador universal, el predicado que define el cuantificador existencial debe probarse en forma singular con cada elemento del dominio.

2.4.2 Transcripción de una expresión existencial

Hay varias formas en las que se puede presentar un cuantificador existencial dentro de una expresión en lenguaje natural, por lo que hay tener cuidado para determinar los elementos que constituyen el cuantificador existencial.

Palabras clave que denotan existencialidad, por ejemplo:

1. «Alguno», como en «Alguno de los presentes ha enfermado». Aquí el dominio pudiera ser «las personas presentes» y el predicado $\lambda p \bullet \texttt{haEnfermado}(p)$, donde $\texttt{haEnfermado}(p) \mapsto \texttt{True}$ cuando la persona p ha estado enferma y `False` en otro caso.

   $$\exists p \in P : \texttt{haEnfermado}(p)$$

2. «A veces», «en ocasiones», por ejemplo en «Cuando llueve, a veces sale el sol». Aquí el dominio puede ser los días, por lo que pudiéramos identificar el dominio como D; el predicado pudiera ser $\lambda d \bullet \texttt{llueve}(d) \wedge \texttt{saleSol}(d)$, donde d es un día.

   $$\exists d \in D : \texttt{llueve}(d) \wedge \texttt{saleSol}(d)$$

Otras palabras que denotan existencialidad pueden ser «hay un», «existe un», «al menos un», entre otras.

Entonces, para transcribir formalmente una expresión coloquial que hace referencia a un cuantificador existencial, se deben distinguir y anotar los tres elementos del cuantificador:

1. El dominio de aplicación.
2. El predicado enunciado en forma singular.
3. El identificador que hace referencia a cada elemento del dominio.

2.4.3 Definición recursiva del cuantificador existencial

En la definición 2.4.1 se involucra un predicado P que debe ser evaluado en cada uno de los elementos de un dominio D que pudiera ser $\varnothing$ o un conjunto no vacío $\{x_0|D'\}$; el procedimiento continúa hasta encontrar alguno que satisfaga el predicado. Podemos proceder recursivamente, tomando un elemento del dominio y probando el predicado, de lo que resultan tres casos:

$$\exists x \in D : P(x) \mapsto \begin{cases} \texttt{False} & \text{si } D = \varnothing. \\ \texttt{True} & \text{si } P(x_0). \\ \exists x \in D' : P(x) & eoc. \end{cases}$$

Para calcular el valor de verdad de un cuantificador existencial con dominio D:

1. Si $D = \varnothing$, claramente no hay elemento x en D que haga $P(x) \mapsto \texttt{True}$, por lo que $\exists x \in D : P(x) \mapsto \texttt{False}$.
2. Si no ocurre lo anterior significa que $D = \{x_0|D'\}$, de modo que P se evalúa con x_0. Si $P(x_0) \mapsto \texttt{True}$, el proceso termina porque hay al menos un elemento que cumple el predicado y $\exists x \in D : P(x) \mapsto \texttt{True}$.
3. En otro caso, se revisa recursivamente el resto de D (esto es en D').

Código 2.5: *Cuantificador existencial recursivo*

```
def existeUn(P, D:list)-> bool:
    """
    Versión recursiva del cuantificador existencial.
    Verifica que al menos un elemento del dominio D cumpla el predicado P.
    """
    if D == []:
        return False
    elif P(car(D)):
        return True
    else:
        return existeUn(P, cdr(D))
```

Ejemplo 2.10

Demuestra en forma recursiva $\exists x \in \{48,72,22,32,12,52\} : 2x < 40 \mapsto \texttt{True}$.

Identificamos el predicado P y el dominio D:

$P \leftarrow \lambda x \bullet 2x < 40$ y $D \leftarrow \{48,72,22,32,12,52\}$

$$\begin{aligned}
\texttt{existeUn}(P,D) &= \texttt{existeUn}(\lambda x \bullet 2x < 40, \{48,72,22,32,12,52\}) && ; P(22) \mapsto \texttt{False} \\
\texttt{existeUn}(P,D) &= \texttt{existeUn}(\lambda x \bullet 2x < 40, \{72,22,32,12,52\}) && ; P(48) \mapsto \texttt{False} \\
\texttt{existeUn}(P,D) &= \texttt{existeUn}(\lambda x \bullet 2x < 40, \{22,32,12,52\}) && ; P(72) \mapsto \texttt{False} \\
&= \texttt{existeUn}(\lambda x \bullet 2x < 40, \{32,12,52\}) && ; P(32) \mapsto \texttt{False} \\
&= \texttt{existeUn}(\lambda x \bullet 2x < 40, \{12,52\}) && ; P(12) \mapsto \texttt{True} \\
&= \texttt{True}.
\end{aligned}$$

El procedimiento encontró que 12 hace que P sea `True`, por lo que no es necesario continuar evaluando el resto de los elementos del dominio.

```
>>> existeUn(lambda x:2*x<40, [22,32,12,52])
True
>>>
```

2.4.4 Unicidad en la existencia

Definición 2.4.2 -- Unicidad en la existencia. Si P es un predicado con dominio en D, el predicado $\exists! x \in D : P(x)$ se utiliza para denotar la existencia de un único elemento en D que cumple P. Así $\exists! x \in D : P(x) \mapsto \texttt{True}$ si existe exactamente un elemento $x \in D$ que hace $P(x) \mapsto \texttt{True}$ y $\exists! x \in D : P(x) \mapsto \texttt{False}$ en cualquier otro caso.

Ejemplo 2.11

Sea $D \leftarrow \{1, 2, 3, 5, 7, 9, 13\}$ y el predicado $\lambda x \cdot \texttt{esPar}(x)$ que devuelve `True` si x es par y devuelve `False` si x no es par, entonces el cuantificador $\exists! x \in D : P(x) \mapsto \texttt{True}$ ya que en D solamente hay un único elemento que hace `True` el predicado, es el 2.

Esta variante del cuantificador existencial es importante para hacer estructuras algebraicas, que son esencialmente conjuntos acompañados de operaciones con los elementos del conjunto, de tal forma que al usar esas operaciones se pueden observar propiedades importantes.

Observa que $\exists! x \in D : P(x) \mapsto F$ en los siguientes casos:

1. $D = \varnothing$
2. No existe elemento alguno en el dominio que verifique el predicado. Por ejemplo $\exists! x \in \{1, 2, 3, 4\} : x > 5 \mapsto \texttt{False}$ porque no hay elemento alguno que cumpla el predicado.
3. Existe más de un elemento en el dominio que verifique el resultado. Por ejemplo $\exists! x \in \{1, 2, 3, 4, 5, 6, 7\} : x > 5 \mapsto \texttt{False}$ porque hay más de un elemento que cumple el predicado.

2.5 Negación de los cuantificadores

Como los cuantificadores son expresiones booleanas, pueden ser utilizadas como argumentos en conectores lógicos, en particular la negación. Esto permite expresar formalmente otras variantes de cuantificadores por ejemplo «ningún», «no todos», «nunca», entre otros.

2.5.1 Negación universal

La negación del cuantificador universal se denota $\neg \forall x \in D : P(x)$, para algún dominio D y un predicado P. Hay diferentes interpretaciones para esta expresión:

1. «No todos», como en «no a todas las personas les gusta la comida italiana». Aquí el dominio podemos definirlo como $G \leftarrow$«las personas»; el predicado debe ser algo como $\lambda g \cdot \texttt{gustaDeComidaItaliana}(g)$, que es `True` si a la persona g le gusta la comida italiana y será `False` en otro caso.

   $$\neg \forall g \in G : \texttt{gustaDeComidaItaliana}(g)$$

   Observa en este ejemplo que el predicado se expresa en singular porque se aplica a una sola persona a la vez.
2. «No siempre», por ejemplo en «Ángela no siempre viste adecuadamente», aquí el dominio se refiere a las ocasiones, por lo que podemos definir el dominio como

$O \leftarrow$«Las ocasiones» y el predicado $\lambda o \bullet \texttt{angelaVisteAdecuadamente}(o)$, que es `True` cuando Ángela viste adecuadamente en la ocasión o y `False` en otro caso.

$$\neg \forall o \in O : \texttt{angelaVisteAdecuadamente}(o)$$

Ejemplo 2.12

La expresión «En el conjunto $\{1, 2, 3, 5, 7, 9, 15, 17\}$ no todos son primos» se puede modelar formalmente como

$$\neg \forall x \in \{1, 2, 3, 5, 7, 9, 15, 17\} : \texttt{esPrimo}(x)$$

considerando un predicado $\lambda x \bullet \texttt{esPrimo}(x)$ que es `True` si x es un número primo y es `False` en otro caso.

2.5.2 Negación existencial

La negación existencial es un predicado que se denota $\neg \exists x \in D : P(x)$, para algún dominio D y predicado P. De manera similar que en la negación universal, hay expresiones en lenguaje natural que son modeladas con la negación del cuantificador existencial.

1. «Ningún». En el sentido de «Ni uno». Por ejemplo «oh Señor, ninguno hay como tú entre los dioses»[2]. De esta expresión podemos descubrir que se habla del dominio de los dioses, por lo que podemos decir que D es el conjunto de dioses. El predicado hace referencia a la comparación de cada dios con el Señor, por lo que podemos crear un predicado $\lambda d \bullet \texttt{semejanteAlSr}(d)$ y escribir la expresión existencial

$$\neg \exists d \in D : \texttt{semejanteAlSr}(d)$$

Ejemplo 2.13

La expresión «en los números enteros positivos no hay ninguno cuyo producto por -1 sea mayor que cero» se puede modelar formalmente como:

$$\neg \exists n \in \mathbb{Z}^+ : -1 * n > 0$$

El dominio son los números enteros positivos, denotado por $\mathbb{Z}^+$ y el predicado es $\lambda n \bullet -1 * n > 0$.

2.5.3 Eliminación de las negaciones en cuantificadores

Como regla general es más conveniente trabajar con predicados enunciados en términos positivos, es decir, sin negaciones. Por ejemplo, en lugar de decir «no todos», se prefiere decir «al menos uno», esta traducción conlleva un cambio en la notación [O'L16, p. 86]:

- Para quitar la negación en un cuantificador universal:

$$\neg \forall x \in D : P(x) \quad \text{se traduce como} \quad \exists x \in D : \neg P(x)$$

- Para quitar la negación en un cuantificador existencial:

$$\neg \exists x \in D : P(x) \quad \text{se traduce como} \quad \forall x \in D : \neg P(x)$$

Observa que en ambos casos se hacen tres cosas:

1. Se quita el símbolo $\neg$ del inicio de la expresión.
2. Se reemplaza el símbolo $\forall$ por $\exists$ o viceversa.

[2] Tomado de La Biblia en Salmos 86:8

3. Se coloca la negación en el predicado.

La interpretación de la traducción puede dar lugar a nuevos predicados que se enuncian diferente pero conservan el mismo sentido.

- «No a todas las personas les gusta la comida italiana». Se puede modelar formalmente como $\neg\forall g \in G : \texttt{gustaDeComidaitaliana}(g)$ [apartado 2.5.1, página 59], pero al quitar la negación al inicio se reescribe

$$\exists g \in G : \neg\texttt{gustaDeComidaitaliana}(g),$$

  lo que se puede interpretar como «hay alguien a quien no le gusta la comida italiana». La negación del predicado puede dar lugar a un nuevo predicado descrito en forma positiva (sin negación) como $\lambda g \bullet \texttt{disgustaDeComidaItaliana}(g)$, para finalmente escribir el predicado

$$\exists g \in G : \texttt{disgustaDeComidaitaliana}(g),$$

  y la interpretación es que «hay alguien en la gente a quien le disgusta la comida italiana».
- La expresión bíblica «oh Señor, ninguno hay como tú entre los dioses» en la sección 2.5.2 [60] fue modelada como $\neg\exists d \in D : \texttt{semejanteAlSr}(d)$ y al quitar la negación al inicio, se puede reescribir como

$$\forall d \in D : \neg\texttt{semejanteAlSr}(d),$$

  lo que se puede interpretar como «todos los dioses no son semejantes al Señor». En este caso también podemos crear un nuevo predicado enunciado sin negaciones, pero con el mismo sentido.

$$\forall d \in D : \texttt{diferenteAlSr}(d),$$

  lo que finalmente se interpreta como «todos los dioses son diferentes al Señor».

Ejemplo 2.14

Al remover el operador de negación de la expresión $\neg\forall x \in \{1,2,3,4,5\} : x < 5$ se obtiene

$$\exists x \in \{1,2,3,4,5\} : \neg(x < 5),$$

pero esta misma expresión es equivalente a

$$\exists x \in \{1,2,3,4,5\} : x \geq 5,$$

2.6 Cuantificadores con aridad múltiple

Muy frecuentemente se desea modelar expresiones que requieren predicados que involucran dos o más variables de diferentes dominios, para estas situaciones las reglas de construcción se centran más en los dominios que en los predicados.

Veamos algunos predicados de aridad múltiple:

1. $\lambda p, q \bullet (p \to q) \vee (q \to p)$, que es de aridad 2.
2. $\lambda x, y, z \bullet x^z \leq y^z$, que es de aridad 3.
3. $\lambda a, b \bullet \texttt{esHermano}(a, b)$, es de aridad 2.

Para construir cuantificadores con predicados como los enlistados, se debe especificar un dominio para cada variable declarada, los valores en cada dominio se utilizarán para instanciar cada variable y la combinación de valores instanciados debe ser única para no repetir cálculos.

Consideremos el predicado $\lambda p, q \bullet (p \to q) \vee (q \to p)$ que es de aridad 2 y los siguientes dominios $D_p \leftarrow \{\texttt{True}, \texttt{True}, \texttt{False}, \texttt{False}\}$ y $D_q \leftarrow \{\texttt{True}, \texttt{False}, \texttt{True}, \texttt{False}\}$. El dominio D_p contiene los valores que serán aplicados a la variable p, mientras que D_q contiene los valores de q. Un cuantificador universal con este predicado es:

$$\forall p \in D_p, q \in D_q : (p \to q) \vee (q \to p).$$

Para calcular el valor de verdad, cada variable se instancía con un valor del dominio a la vez, por lo que es necesario que todos los dominios tengan la misma cantidad de valores. La siguiente tabla muestra los dominios y las aplicaciones, por espacio se han utilizado las letras `T` para `True` y `F` para `False`.

D_p (Valores para p)	`True`	`True`	`False`	`False`
D_q (Valores para q)	`True`	`False`	`True`	`False`

$(p \to q) \vee (q \to p)$	$(\texttt{T} \to \texttt{T}) \vee (\texttt{T} \to \texttt{T})$
	`True` $\vee$ `True`
	`True`
		$(\texttt{T} \to \texttt{F}) \vee (\texttt{F} \to \texttt{T})$
		`False` $\vee$ `True`
		`True`
			$(\texttt{F} \to \texttt{T}) \vee (\texttt{T} \to \texttt{F})$
			`True` $\vee$ `False`
			`True`
				$(\texttt{F} \to \texttt{F}) \vee (\texttt{F} \to \texttt{F})$
				`True` $\vee$ `True`
				`True`

Así $\forall p \in D_p, q \in D_q : (p \to q) \vee (q \to p) \mapsto$ `True`

Para escribir cuantificadores multivariable hay que observar dos reglas sobre los dominios:

1. El número de dominios debe coincidir con la aridad del predicado.
2. El número de elementos en cada dominio debe ser igual.

Ejemplo 2.15

El valor de verdad de $\exists x \in \{1,2,3\}, y \in \{5,6,7\}, z \in \{2,2,2\} : x^z \leq y^z$ se calcula instanciando cada variable con uno de los valores de su propio dominio.

$x^z \leq y^z$	Evaluación
$1^2 \leq 5^2$	`True`
$2^2 \leq 6^2$	`True`
$3^2 \leq 7^2$	`True`

Por lo que $\exists x \in \{1,2,3\}, y \in \{5,6,7\}, z \in \{2,2,2\} : x^z \leq y^z \mapsto$ `True`.

`all(`⟨*iterable*⟩`)`. Devuelve `True` si el ⟨*iterable*⟩ está vacío o todos sus elementos son verdaderos.
`any(`⟨*iterable*⟩`)`. Devuelve `False` si el ⟨*iterable*⟩ está vacío o alguno de sus elementos es verdadero.

Las siguientes definiciones permiten el uso de aridad múltiple. Estas definiciones sustituyen a las definiciones del cuantificador universal y existencial en el código 2.4, página 56 y código 2.5, página 58 respectivamente.

Código 2.6: *Cuantificador universal*

```
1 def paraTodo(P, *D)-> bool:
2     """ Cuantificador universal
3     Devuelve True cuando todas las proposiciones en D
4     son True, de otro modo devuelve False.
5     """
6     return all(map(P, *D))
```

Código 2.7: *Cuantificador existencial*

```
def existeUn(P, *D:list)-> bool:
    """ Cuantificador existencial
    Devuelve True cuando al menos una de las proposiciones
    son True, y devuelve False cuando todas son False.
    """
    return any(map(P, *D))
```

En estas definiciones de código fuente, primero se evalúa con `map` cada elemento del dominio, generando un iterable con valores `True` o `False`. Luego se utiliza `all` en el caso del cuantificador universal para determinar si todos ellos son `True` o no, o bien `any` en el caso del cuantificador existencial para determinar su valor de verdad.

2.7 Cuantificadores anidados

Ya que los cuantificadores son expresiones booleanas, éstas pueden formar parte de otras expresiones booleanas, formando así nuevas expresiones más complejas que describen situaciones que involucran predicados en los predicados.

Ejemplo 2.16

Consideremos la tabla de operaciones $\odot$ siguiente, que tiene dominio $D \leftarrow \{a, b, c\}$:

$\odot$	a	b	c
a	a	b	c
b	b	a	c
c	c	c	a

Podemos decir que «hay un único elemento en el dominio que es el resultado de operar los elementos del dominio consigo mismos», lo que formalmente podemos escribir como:

$$\exists! x \in D : \forall y \in D : \odot(y, y) \mapsto x.$$

2.7.1 Ámbito de un cuantificador

El ámbito de un cuantificador es el segmento sintáctico en donde tiene validez la variable definida en el cuantificador. Recordemos que un cuantificador tiene la forma $\boxed{\Psi x \in D : P(x)}$, donde Ψ es o bien $\forall$ o $\exists$ o incluso puede ser $\exists!$ El marco indica el ámbito del cuantificador, donde queda bien determinada toda la expresión.

En el ejemplo 2.16 se tienen cuantificadores anidados, que tienen ámbitos anidados:

$$\boxed{\exists x \in D : \boxed{\forall y \in D : \odot(y, y) \mapsto x}}$$

Siguiendo con el ejemplo, el ámbito de la variable x abarca incluso el cuantificador universal, por su parte, el ámbito del cuantificador universal es más restringido pues solo abarca la operación $\odot$.

Visto de otro modo, si no se tuviera la expresión existencial, habría un identificador cuyo valor es desconocido, por lo que no se podría hacer la operación:

$$\boxed{\forall y \in D : \odot(y, y) \mapsto x}$$

Ejemplo 2.17

Considera la expresión «Todos los vecinos no han pagado al menos la cuota de un mes, aunque todos ellos son buenas personas».

Si hacemos:

$\texttt{V} \leftarrow\!\!\shortmid \{\text{Vecinos}\}$

$\texttt{M} \leftarrow\!\!\shortmid \{\texttt{ene}, \texttt{feb}, \texttt{mar}, \texttt{abr}, \texttt{may}, \texttt{jun}, \texttt{jul}, \texttt{ago}, \texttt{sep}, \texttt{oct}, \texttt{nov}, \texttt{dic}\}$

$\lambda v, m \bullet \texttt{haPagado}(v, m)$ es un predicado que es `True` cuando un vecino v ha pagado el mes m y el `False` en caso contrario.

$\lambda v \bullet \texttt{esBuenaPersona}(v)$ es un predicado que es `True` si el vecino v es buena persona y es `False` en caso contrario.

Podemos modelar el texto como:

$$\forall v \in \texttt{V} : [\exists m \in M : \neg \texttt{haPagado}(v, m)] \wedge \texttt{esBuenaPersona}(v),$$

donde la variable m es uno de los meses y v representa a uno de los vecinos. Los ámbitos de los cuantificadores son:

$$\forall v \in \texttt{V} : \boxed{\exists m \in M : \neg \texttt{haPagado}(v, m)} \wedge \texttt{esBuenaPersona}(v)$$

Ejemplo 2.18

Calculemos el valor de verdad de la expresión $\forall x \in \{1, 2, 3\} : \forall y \in \{4, 5, 6\} : x < y$.

Aquí el predicado escrito en notación lambda es $\lambda x \bullet (\lambda y \bullet x < y)$. En primer lugar, se asigna un primer valor a x, luego se asignan los valores a la variable y que es la más interna, esto genera un ciclo en donde la variable más externa no cambia, pero la interna toma cada valor en su dominio.

Luego se cambia el valor de x y se repiten todos los valores de y, así hasta terminar.

Esto produce 9 expresiones que deben ser evaluadas, cuando $x \leftarrow\!\!\shortmid 1$, y toma los valores 4, 5 y 6. Luego cambiar $x \leftarrow\!\!\shortmid 2$ y repetir las asignaciones para y; finalmente $x \leftarrow\!\!\shortmid 3$ y repetir las asignaciones para y.

En la siguiente tabla se muestras las asignaciones, la lectura debe ser por por renglones de arriba hacia abajo, en cada renglón por columnas de izquierda a derecha.

$x < y$	$y \leftarrow\!\!\shortmid 4,$	$y \leftarrow\!\!\shortmid 5,$	$y \leftarrow\!\!\shortmid 6,$	$\forall y \in \{4, 5, 6\}$
$x \leftarrow\!\!\shortmid 1 :$	$1 < 4,$	$1 < 5,$	$1 < 6,$	`True`
$x \leftarrow\!\!\shortmid 2 :$	$2 < 4,$	$2 < 5,$	$2 < 6,$	`True`
$x \leftarrow\!\!\shortmid 3 :$	$3 < 4,$	$3 < 5,$	$3 < 6,$	`True`
			$\forall x \in \{1, 2, 3\}$	`True`

Cada asignación de valor para x, produce un vector de resultados creado por las evaluaciones del predicado con el valor actual de x y cada valor de y, esto da valor de verdad al cuantificador interno.

Al terminar las evaluaciones, se evalúa el vector de resultados para determinar el valor de verdad del cuantificador externo.

Ejercicios

1. Escribe una demostración recursiva como la que se hizo en el ejemplo 2.8 de la página 56, para los siguientes casos donde $A \leftarrow \{2,9,4,1,0,7,11,29,41\}$:
   a) *Ejemplo:* $\forall x \in A : x \bmod 2 = 0 \mapsto \texttt{False}$
      $\texttt{paraTodo}(\lambda x \bullet x \bmod 2 = 0, \{2,9,4,1,0,7,11,29,41\}) \mid x \leftarrow 2 \mid 2 \bmod 2 = 0 \mapsto \texttt{True}$
      $\texttt{paraTodo}(\lambda x \bullet x \bmod 2 = 0, \{9,4,1,0,7,11,29,41\}) \mid x \leftarrow 2 \mid 9 \bmod 2 = 0 \mapsto \texttt{False}$
   b) $\exists x \in A : \forall y \in A : x = y \rightarrow x - y > 0 \mapsto \texttt{False}$
   c) $\exists x \in A : \texttt{esPrimo}(x)$
   d) $\forall x \in A : x \neq 0 \rightarrow \frac{3}{x} > \frac{x^2}{3}$
2. Transcribe al lenguaje simbólico de lógica las siguientes declaraciones que se encuentran en lenguaje natural. Declara cuál es el dominio, el predicado y la variable dummy.
   a) *Ejemplo:* «A veces no duermo bien». El dominio pueden ser las noches, por lo que hacemos el dominio `Noches`, el predicado es $\lambda n \bullet \neg \texttt{duermobien}(n)$, así la expresión es $\exists n \in \texttt{Noches} : \neg \texttt{duermoBien}(n)$
   b) «Todos los platillos del menú son caros y no son sabrosos».
   c) «Siempre que voy al cine, encuentro a alguien conocido».
   d) «Hay juguetes que no son aptos para niños o son caros».
3. Transcribe a `Python` las siguientes declaraciones. Utiliza las definiciones hechas en el capítulo para los cuantificadores:
   a) *Ejemplo:* $\forall x \in \{0,\ldots,10\} : x^2 < 100$ se escribe
      ```
      paraTodo(lambda x:x**2 <10, [0,1,2,3,4,5,6,7,8,9,10])
      ```
   b) $\exists x \in \{-10,\ldots,10\} : \dfrac{(x+3)^2 - 5}{5} < 0$
   c) $\forall x \in \{3,6,9\}, y \in \{2,2,2\} : x^y < 100$
4. Calcula el valor de verdad de las siguientes declaraciones:
   a) *Ejemplo:* $\forall x \in \{0,\ldots,10\} : x^2 < 100 \mapsto \texttt{False}$
   b) $\forall x \in \{n \in \mathbb{Z} | 0 \leq n < 10\} : x^2 < 50 \leftrightarrow x \leq 7$
   c) $\exists x \in \{n \in \mathbb{Z} | -10 \leq n \leq 10\} : \dfrac{(x+3)^2 - 5}{5} < 0$
   d) $\exists x \in \{n \in \mathbb{Z} | -10 \leq n \leq 10\} : \forall y \in \{n \in \mathbb{Z} | -10 \leq n \leq 10\} : x < -2y \vee x < 2y$
5. Escribe una función en `Python` llamada `existeUnUnico` que reciba como entrada un predicado P y un dominio D. Se devuelve `True` si en D existe exactamente un elemento x tal que $P(x) \mapsto \texttt{True}$ y devuelve `False` en cualquier otro caso.

```python
def existUnUnico(P, D:list)-> bool:
    """ Unicidad en la existencia
    Devuelve True cuando existe un único elemento e
    en el dominio D, que cumple el predicado P.
    """
    pass # <- Escribe aquí tu función
```

Utiliza los siguientes ejemplos para comprobar tus resultados.

```
>>> existeUnUnico(lambda x:x > 5, [1,2,6,3,4,5,6])
False
>>> existeUnUnico(lambda x:x > 5, [1,2,6,3,4,5])
True
>>> existeUnUnico(lambda nomb:nomb=='Ana', ['Art', 'Mary', 'Lua', 'Ana', 'Jo'])
True
>>> existeUnUnico(lambda nomb:nomb=='Ana', ['Ana', 'Mary', 'Lua', 'Ana', 'Jo'])
False
>>>
```

6. Las siguientes expresiones involucran cuantificadores negados, escribe una traducción en lenguaje natural adecuada.
   a) *Ejemplo:* $\neg\forall d \in \texttt{Dias} : \texttt{haceFrio}(d)$, puede interpretarse como «no todos los días hace frío».
   b) $\neg\exists p \in \texttt{Perros} : \texttt{esBravo}(p)$
   c) $\neg\forall a \in \texttt{Archivos} : \neg\exists r \in \texttt{Registros} : \texttt{falta}(a, r)$
7. Escribe la negación de las siguientes proposiciones, y encuentra una expresión equivalente que no tenga negación al inicio:
   a) *Ejemplo:* $\exists x \in \{1,2,3,4\} : 2x > 5$.
   La negación de $\exists x \in \{1,2,3,4\} : 2x > 5$ es $\neg\exists x \in \{1,2,3,4\} : 2x > 5$, pero esta expresión es equivalente a $\forall x \in \{1,2,3,4\} : \neg(2x > 5)$, que es precisamente $\forall x \in \{1,2,3,4\} : 2x \leq 5$.
   b) $\forall x \in \mathbb{Z} : x^2 \geq 2x$
   c) $\exists x \in \{1,2,3,4\} : \neg\exists y \in \{2,4,6,8\} : xy \neq -xy \rightarrow (x > 0 \wedge y > 0)$.
   d) «No todas las comidas son saludables».
   e) «Todas las comidas tienen al menos un ingrediente saludable».
8. Considera la siguiente interacción en `Python` y determina el valor de verdad de la proposición:

```
>>> D = [20, 16, 18, 14, 6, 10, 4, 12, 8]
>>> P = lambda x:neg(impl(x*3 < 3*x**2, x > 5))
>>> existeUnUnico(P, D)
_____________
>>>
```

9. Analiza la siguiente interacción en `Python` y escribe lo que se imprime al terminar la instrucción condicional.

```
>>> D = [20, 16, 18, 14, 6, 10, 4, 12, 8]
>>> if paraTodo(lambda x:impl(2*x>4,x%6==0),D):
        print("A")
    else:
        print("B")

_____________
>>>
```

10. Descubre el error en las siguientes instrucciones de `Python`.
    a) `paraTodo(lambda x: x+1, [1,2,3])`
    b) `existeUn(lambda y: x>y, [10,20,30], [1,2,3])`

II

Teoría de Conjuntos

Conceptos fundamentales 3

3.1 Generalidades

En teoría de conjuntos hay tres conceptos que se han definido de manera intuitiva. Aquí supondré que este par de conceptos son tan básicos, que ya son bien entendidos y no requieren explicación:

> Elemento : *Miembro de un conjunto.*
> Conjunto : *Agrupación de elementos.*
> Pertenencia : *Circunstancia acerca de un elemento que forma parte de un conjunto.*

Estos conceptos constituyen una definición *circular*, que es cuando un concepto se define en términos de otro, y ese otro se define en términos del primero. Este tipo de definiciones no son útiles en términos computacionales; por eso se apela a la intuición y al sentido común para poder entenderlos. Así, de manera informal podemos decir que un conjunto es «cualquier colección bien definida de objetos» [Dev03, p. 57]. Una vez de acuerdo en lo que se comprende como conjunto, podemos definir conjuntos en particular.

Hay dos maneras de definir un conjunto en particular, se basan en la manera de decidir los elementos que pertenecen conjunto:

Explícita: Es cuando se enlistan [O'L16, p. 118] uno a uno los elementos que deben pertenecer al conjunto. A esta manera también se le llama **extensional**.

Implícita: Es cuando se enuncia un predicado que se debe cumplir para determinar la pertenencia, de tal forma que si la evaluación del predicado es `True`, entonces el objeto pertenece al conjunto; por otro lado, si la evaluación es `False`, entonces el objeto no pertenece. A esta manera también se le llama: una forma **intencional** de definir el conjunto.

3.1.1 Notación para conjuntos

En general, comprendemos que se está haciendo referencia a un conjunto porque el texto dice algo como «el conjunto de ...», pero en matemáticas se estila delimitar el conjunto mediante los símbolos de llaves: { para iniciar el conjunto y } para terminar. Así todo lo que se encuentre entre llaves es parte del mismo conjunto.

Si el conjunto está descrito en forma explícita, se escriben sus elementos en forma de lista, sin repetir elementos, por ejemplo:

1. $\{a, e, i, o, u\}$.
2. $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$
3. $\{2, 4, 6, 8, \ldots\}$, aunque el conjunto es infinito, dando los primeros elementos podemos determinar que se trata de los números enteros positivos pares.

Como es importante no repetir elementos, crearemos una función auxiliar en `Python` que elimina los elementos repetidos de una lista.

Código 3.1: *Quita elementos duplicados de una lista*

```
1  def quitaDup(L:list, R:list = [])-> list:
2      """
3      Quita los elementos duplicados en una lista.
4      """
5      if esVacio(L):
6          return R
7      elif en(car(L), cdr(L)):
8          return quitaDup(cdr(L), R)
9      else:
10         return quitaDup(cdr(L), R+[car(L)])
```

Si el conjunto está descrito en forma implícita, se debe escribir el predicado con el que se determina si un elemento debe pertenecer o no al conjunto, por ejemplo:

1. {las vocales en español} El predicado está descrito en lenguaje natural, no cualquier letra debe pertenecer al conjunto, solo las vocales del alfabeto en español.
2. $\{\sigma \in \Sigma_{\text{español}} | \sigma \text{ es vocal}\}$. Aquí ya intervienen dos símbolos importantes, $\in$ que se lee «que pertenece a» y el símbolo | que se lee «tal que» [en la sección 3.3 se estudia con más detalle]. Así la lectura de este conjunto es «el conjunto de las letras σ del alfabeto en español, tales que cumplen que σ es una vocal».
3. $\{\sigma \in \Sigma_{\text{español}} | \texttt{esVocal}(\sigma)\}$. Es el mismo conjunto anterior [se lee de igual manera], pero ahora se ha definido el predicado en la forma de una función booleana.
4. $\{\sigma_0 | \Sigma'\}$ es una notación que se refiere a un conjunto que tiene al menos un elemento, que es representado por σ_0 y Σ' es el conjunto que contiene al resto de los elementos quitando a σ_0 [esta notación se estudia más profundamente en la sección 3.3.3].

Como regla general, las primeras letras mayúsculas se usan para representar conjuntos, como A, B, C,D, E, A_1, A_n. Hay letras griegas que se utilizan frecuentemente para representar conjuntos como Σ. También se pueden usar diferentes tipografías para representar diferentes conjuntos, por ejemplo un conjunto R [posiblemente de ratones], puede ser diferente al conjunto $\mathbb{R}$, que es utilizado generalmente para los números reales. En este libro se utilizarán letras mayúsculas o palabras que inicien con mayúscula para denotar los conjuntos, como `A`, `B`, `S`, `Alfabeto`.

Los elementos de los conjuntos se representan con las primeras letras minúsculas del alfabeto en español o griego, como a, b, c, `a`, `b`, `e` o `x`; las letras griegas minúsculas se refieren a variables que hacen referencia a elementos de conjuntos, como α, β.

3.2 Creación de conjuntos

En ciencias computacionales, al igual que en matemáticas, no se puede utilizar un conjunto [y en general ningún otro objeto] sin haberlo creado anteriormente. Los conjuntos pueden ser creados de manera explícita o implícita.

Para crear un conjunto asociado a un identificador escribimos $A \leftarrow \{\ldots\}$ lo que significa que en adelante, y hasta que se diga otra cosa, A «es» un conjunto que contiene lo que esté escrito en lugar de los tres puntos. En código fuente la notación cambia un poco por la sintaxis del lenguaje y del acuerdo siguiente.

En el código fuente denotaremos los conjuntos definidos por extensión encerrando entre corchetes los elementos del conjunto, sin repetir elementos.

En `Python` se ofrece el tipo de dato `set` para denotar conjuntos, como en `set(1,2,3)`, que produce un objeto de tipo `set`, sin embargo, en este libro modelaremos conjuntos mediante listas, esto debido a una limitante importante de `Python`, que es la imposibilidad de denotar conjuntos de conjuntos, que es un tema importante en matemáticas discretas.

```
>>> set(set(1,2,3), set(3,4))
TypeError: set expected at most 1 argument, got 3
>>>
```

Cuando conocemos cada uno de los elementos, podemos crear un conjunto con ellos, haciéndolo de manera **explícita**. Si conocemos la propiedad que los elementos deben cumplir, el conjunto será definido de manera **implícita**.

Ejemplo 3.1

Se escribe $C_1 \leftarrow \{0, 2, 4, 6, 8\}$ para hacer saber que C_1 será el conjunto que contiene los números 4, 2, 0, 6 y 8. El conjunto $C_2 \leftarrow \{x | 0 \leq x < 10 \wedge x \bmod 2 = 0\}$, permite saber que C_2 es el conjunto que reúne a aquellos objetos x que hacen verdadero el predicado $\lambda x \cdot 0 \leq x < 10 \wedge x \bmod 2 = 0$.

En `Python` los mismos ejemplos se pueden lograr haciendo:

1. `C1 = conj(0,2,4,6,8)`, que describe un conjunto explícito con los números 0, 2, 4, 6 y 8. El resultado es una lista, que en este contexto debe será interpretada como un conjunto.
2. Para el caso de C_2, el conjunto es definido por el predicado del conjunto. Así podemos hacer `C2 = conj(lambda x:0<=x<10 and x%2 == 0)`.

Enseguida se detalla la función `conj` que sirve para crear un conjunto. Los valores asociados a los parámetros formales determinan cómo debe ser considerado el conjunto, ya sea como una lista o como un predicado.

Código 3.2: *Definición de conjuntos*

```
1  def conj(*items)-> list:
2      """ Crea un conjunto de manera implícita o explícita.
3      Si no hay argumentos, crea un conjunto vacío
4      Si items es una lista no determinada, el conjunto es la lista
5      con los elementos dados.
6      Si items es un predicado, el conjunto es definido por ese predicado.
7      """
8      params = list(items)
9      if items==():
10         return []
11     elif callable(params[0]):
12         return params[0]
13     else:
14         return quitaDup(params)
```

Ejemplo 3.2

Sea $A_1 = \{a, b, c, d\}$ un conjunto. En `Python` usaremos la función para crear un conjunto.

```
>>> A1 = conj('a', 'b', 'c', 'd')
>>>
```

El resultado es una variable llamada `A1` con valor `['a', 'b', 'c', 'd']`, que interpretaremos como el conjunto A_1.

```
>>> A1
['a', 'b', 'c', 'd']
>>>
```

$A_2 = \{x | 0 \leq x < 10 \wedge x \bmod 2 = 0\}$ se puede modelar como:

```
>>> A2 = conj(lambda x: 0 <= x < 10 and x % 2 == 0)
>>>
```

3.3 La pertenencia

El concepto de pertenencia es intuitivo en la teoría de conjuntos, lo que significa que no hay una definición formal para el concepto, en términos coloquiales la pertenencia es una circunstancia relativa a un elemento que forma parte de un conjunto.

Es posible determinar si un elemento pertenece a un conjunto si se cumple una de las siguientes condiciones, que se derivan de la manera en que el conjunto ha sido definido.:

1. Cuando el conjunto es explícitamente definido, el elemento pertenece al conjunto si este se encuentra explícitamente entre los elementos enlistados.
2. Cuando el conjunto es implícitamente definido, el elemento pertenece al conjunto en caso de que se verifique el predicado que define al conjunto.

Si A es un conjunto, a es un elemento y si alguna de las condiciones anteriores se cumple, lo escribimos $a \in A$, de otro modo lo escribimos $a \notin A$.

Observa que $a \in A$ es una proposición lógica, cuyo valor de verdad es `True` si se cumple alguna de las condiciones enlistadas anteriormente; será `False` en otro caso. Por su parte, $a \notin A$ es una proposición lógicamente equivalente a $\neg a \in A$, por lo que cuando $a \in A \mapsto$ `True`, $a \notin A \mapsto$ `False` y cuando $a \in A \mapsto$ `False`, $a \notin A \mapsto$ `True`.

Así, una definición efectiva para determinar si un elemento pertenece o no a un conjunto debe considerar ambos casos.

Definición 3.3.1 -- Pertenencia. Si A es un conjunto y e un elemento, e pertenece al conjunto A es una proposición que escribimos $e \in A$ y tiene los siguientes valores:

$$a \in A \mapsto \begin{cases} \texttt{True} & \text{si } (\exists x \in A : x = a) \oplus P(x). \\ \texttt{False} & \textit{eoc.} \end{cases}$$

La expresión $\exists x \in A : x = a$ se utiliza cuando el conjunto A es definido explícitamente y $P(x)$ se utiliza cuando el conjunto fue definido implícitamente con predicado P.

Observa en la definición que el símbolo $\in$ aparece en dos momentos: $a \in A$ y $x \in A$. El primero es una proposición que determinará su valor de verdad en dependencia de la proposición a la derecha del símbolo $\mapsto$ [que se lee «produce»]. La segunda no es una proposición ya que es parte del cuantificador existencial. Establece que la variable x tomará valor con cada elemento de A se evaluará el predicado $x = a$ hasta encontrar alguno con el que resulte `True`.

Calcular el valor de verdad de la disyunción exclusiva será posible solamente para alguna de las definiciones de conjunto.

Código 3.3: *Pertenencia*

```
def en(a, C=None)-> bool:
    """ Pertenencia.
    Determina la pertenencia de un elemento a un conjunto
    sin importar si se trata de un conjunto extensional o intencional
    Se devuelve un valor booleano.
    """
    if isinstance(C, list):
        return existeUn(lambda x: x == a, C)
    elif callable(C):
        try:
            res = C(a)
        except TypeError:
            return False
        else:
            return res
    else:
        return False
```

Ejemplo 3.3

Considera los conjuntos $A_1 \leftarrow \{9,6,2,1,4\}$ y $A_2 \leftarrow \{9,6,1,4\}$. Utiliza el programa `en` para determinar el valor de $2 \in A_1$, $2 \in A_2$. Luego considera el conjunto $B = \{x | 0 \leq x^2 \leq 100\}$ y determina $4 \in B$ y $16 \in B$

```
>>> A1 = [9,6,2,1,4]
>>> A2 = [9,6,1,4]
>>> en(2,A1)
True
>>> en(2,A2)
False
>>> B = conj(lambda x: 0 <= x**2 <= 100)
>>> en(4,B)
True
>>> en(16,B)
False
>>>
```

Podemos ir un poco mas lejos, probando si un objeto que no es un número pertenece al conjunto.

```
>>> en('a',A1)
False
>>> en('a',A2)
False
>>> en('a',B)
False
>>>
```

`Python` incluye el comando `try`, que es la manera de gestionar errores que pudieran ocasionar comportamientos indeseados en la ejecución del programa. La sintaxis es como sigue:

```
try:
    ⟨expresión-try⟩
except ⟨Tipo-Error⟩:
    ⟨expresión-except⟩
else:
    ⟨expresión-else⟩
```

Cuando se ejecuta el comando `try`, primero trata de realizar lo que esté en la ⟨*expresión-try*⟩, esperado que no ocurra algún error. En caso de que no ocurra error alguno; en caso de que se detecte algún error, se ejecuta la ⟨*expresión-except*⟩ que corresponda al tipo de error que se

detectó, de otro modo se ejecuta la ⟨*expresión-else*⟩. Una descripción más detallada y completa se puede encontrar en `https://docs.python.org/es/3/tutorial/errors.html` [versión en español].

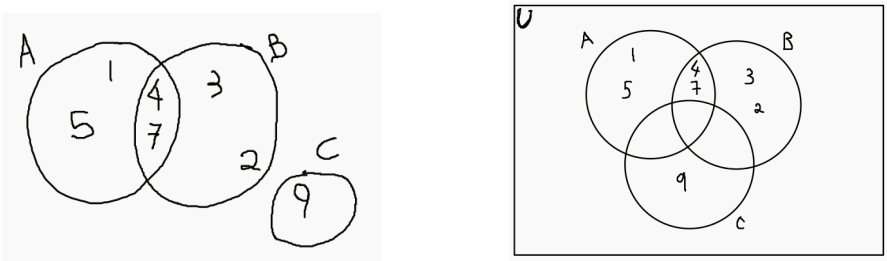
`callable`(⟨*object*⟩): Devuelve `True` si el argumento se puede invocar como una función, y `False` de otro modo.

`isinstance`(⟨*object*⟩, ⟨*classinfo*⟩): Devuelve `True` si ⟨*object*⟩ es una instancia de la clase ⟨*classinfo*⟩, y `False` si no lo es.

3.3.1 Diagramas de Euler

Los conjuntos, especialmente cuando son explícitamente definidos y tienen pocos elementos, pueden ser representados gráficamente por diagramas de Euler [Eul12, Lya20].

Los diagramas de Euler se construyen encerrando en una curva cerrada a los elementos que contiene el conjunto, cada conjunto se dibuja en una curva diferente, si hay elementos compartidos entre los conjuntos, las curvas pueden superponerse, como se muestra en la figura 3.1a.



(a) *Diagrama de Euler* ***(b)*** *Diagrama de Venn*

Figura 3.1: *Diagramas para los conjuntos* $A \leftarrow\!\shortmid \{1, 4, 5, 7\}$, $B \leftarrow\!\shortmid \{4, 3, 7, 2\}$ *y* $C \leftarrow\!\shortmid \{9\}$.

Hay una técnica similar para mostrar los conjuntos de manera gráfica, son los diagramas de Venn. A diferencia de los diagramas de Euler, los diagramas de Venn deben mostrar todas las relaciones entre los conjuntos dados, particularmente el conjunto universo, denotado por U, que generalmente se dibuja como un rectángulo que encierra todos los demás conjuntos, como se muestra en el ejemplo de la figura 3.1b.

En la práctica cotidiana, es más fácil mostrar los conjuntos mediante diagramas de Euler, pero se puden utilizar ambos de manera indistinta.

3.3.2 El conjunto vacío

Definición 3.3.2 Un conjunto vacío es un conjunto que no tiene elementos.

Si uno se pregunta ¿qué utilidad puede tener un conjunto vacío? la respuesta es equivalente a saber la utilidad del 0 en los números. El conjunto vacío sirve como punto de inicio, como referencia base, particularmente en las definiciones recursivas que frecuentemente haremos en este libro.

El conjunto vacío se denota $\varnothing$. Como $\varnothing$ es un conjunto, también se puede escribir con las llaves {}, así sin encerrar elemento alguno, pues por su definición no tiene elementos.

El conjunto vacío puede ser un conjunto de cualquier cosa, así un conjunto vacío de caballos, es lo mismo que un conjunto vacío de computadoras o de personas. Se denota del mismo modo y tiene las mismas propiedades.

En `Python`, crearemos el conjunto vacío como un un conjunto sin elementos:

Código 3.4: *El conjunto vacío*

```
# Se crea un conjunto sin elementos
vacio = conj()
```

Si C es un conjunto, la proposición $C = \varnothing$ tiene valor `True` cuando C no tiene elementos y será `False` si C tiene al menos un elemento. $C = \{\}$ es una manera diferente de escribir la misma proposición.

La notación $C \leftarrow\!\shortmid \varnothing$ es una expresión que indica que se ha creado una nueva variable de nombre C, a la cual se le ha asignado el conjunto vacío como su valor.

Cada vez que se crea un nuevo concepto en programación, es recomendable crear un predicado que permita determinar si un objeto es o no el concepto creado, por lo que vamos a crear un predicado que reciba cualquier tipo de dato y devuelva `True` si es el conjunto vacío o `False` si no lo es.

Código 3.5: *Predicado para verificar el conjunto vacío*

```
def esVacio(C)-> bool:
    """
    Determina si C es el conjunto vacío.
    """
    return C == vacio or C == ()
```

Ejemplo 3.4

Verifica utilizando la definición en el código 3.5, que el conjunto {} es vacío y que el conjunto $\{2, i, w, 8\}$ no lo es.

```
>>> esVacio(conj())
True
>>> esVacio(vacio)
True
>>> esVacio([])
True
>>> esVacio(conj(2,'i','w',8))
False
>>> esVacio([2,'i','w',8])
False
>>>
```

Observa que el conjunto vacío se ha escrito de tres formas diferentes: invocando la función `conj()`; como una lista vacía `[]`; y utilizando el concepto `vacio`.

3.3.3 Conjuntos no vacíos

Un conjunto que tiene al menos un elemento no es vacío. Como se estableció en la sección 3.1.1 [página 70], hay dos maneras de establecer que un conjunto explícito no es vacío, en los siguientes ejemplos se asocia el identificador A con un conjunto no vacío:

1. $A \leftarrow\!\shortmid \{a_1, \ldots, a_n\}$
2. $A \leftarrow\!\shortmid \{a_0 | A'\}$

En la primera forma, se establece que el conjunto A tiene n elementos, con $n \geq 1$. El subíndice en la notación, significa que hay una relación de cada elemento del conjunto con un elemento del conjunto de los números naturales, pero de ningún modo establece un orden específico en los elementos del conjunto.

En la segunda forma, el conjunto A tiene al menos un elemento, que es a_0, que puede ser cualquier elemento del conjunto, el término A' representa el conjunto que tiene a todos los demás elementos, excepto a_0. La elección de la literal a para representar uno de los elementos de A es arbitraria.

La notación $\{a_0|A'\}$ recuerda la notación para listas no vacías descrita en la sección 2.1 de la página 50. Como en los conjuntos el orden en que los elementos ocurran no es importante, el primer elemento puede en principio, ser cualquiera. Por la misma razón y por un acuerdo no escrito, tomaremos como primer elemento aquel que ocurra más a la izquierda en la definición extensional del conjunto.

Para acceder al primer elemento del conjunto utilizamos la función `car`, mientras que el resto de los elementos son obtenidos mediante el `cdr` del conjunto [código 2.1, página 50].

Ejemplo 3.5

Sean los conjuntos $A \leftarrow \{3,4,5\}$ y $B \leftarrow \{3,a,\texttt{True}\}$. En `Python` podemos crear los conjuntos A y B utilizando el código 3.2 como:

```
>>> A = conj(3,4,5)
>>> A
[3, 4, 5]
>>> B = conj(3,'a',True)
>>> B
[3, 'a', True]
>>>
```

Utilizando las funciones `car` y `cdr` podemos obtener el primer elemento de los conjuntos A y B, así como el conjunto que tiene al resto de sus elementos:

```
>>> car(A)
3
>>> car(B)
3
>>> cdr(A)
[4, 5]
>>> cdr(B)
['a', True]
>>>
```

Si $A = \varnothing$, entonces la notación $\{a_0|A'\}$ no tiene sentido, pues el conjunto vacío no tiene ni primer elemento ni conjunto que tenga al resto de los elementos.

La expresión $A \neq \varnothing$ es una proposición con valor `True` si A contiene al menos un elemento; será `False` si $A = \varnothing$.

3.4 La cardinalidad

El concepto de «tamaño» es algo intuitivo de conocer, se refiere a la cantidad de elementos que contiene. En la teoría de conjuntos se ha creado un término que resultará mas apropiado.

Definición 3.4.1 -- Cardinalidad. Si A es un conjunto, $|A|$ denota la cardinalidad de A y es la cantidad de elementos que contiene.

Hay otras notaciones matemáticas para la cardinalidad, como $\#A$ y $n(A)$. En este libro utilizaremos $|A|$ en la notación matemática y `card`(A) en la notación de programación.

Observamos que $|\varnothing| \mapsto 0$ [se lee: «al calcular la cardinalidad de $\varnothing$ *se obtiene* 0»], porque no tiene elementos. Un conjunto $\{a_1,\ldots,a_n\}$, tiene n elementos, donde $n \geq 1$, y

un conjunto no vacío $\{a_0|A'\}$ tiene al menos un elemento, que es precisamente a_0 ya que A' podría ser $\varnothing$.

Definición 3.4.2 -- Equinumerosidad. Dos conjuntos que tienen la misma cardinalidad son equinumerosos, un sinónimo es «equipotencia» [MQ02, p. 103].

Se piensa que en tiempos remotos, para saber cuántas posesiones tenía una persona, se colocaba una marca por cada una de sus posesiones, así se podía determinar si alguien tenía más que otro o si al final del día se tenían las mismas posesiones que al iniciar la jornada. El procedimiento es hacer corresponder cada una de las posesiones con cada una de las marcas, este ejercicio es la base para determinar si dos conjuntos son equinumerosos.

Cuando los conjuntos son pequeños y sus elementos son claramente distintos unos de otros, la correspondencia entre ambos conjuntos es fácil de hacer. El problema empieza cuando alguno de los conjuntos tiene tantos elementos que se vuelve impráctica la tarea de hacer la correspondencia, por ejemplo cuando uno de los conjuntos es un saco de granos de arroz.

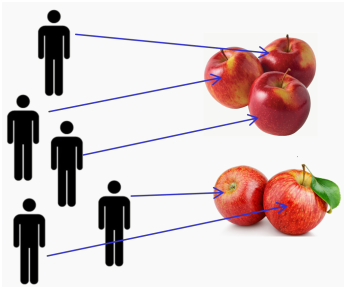


Figura 3.2: *Conjuntos equinumerosos*

Ejemplo 3.6

El conjunto de los números enteros $\mathbb{Z}$ es equinumeroso al conjunto de los números naturales que incluyen al 0 $\mathbb{N}^*$. Si hacemos corresponder el 0 de los enteros con el 0 de los naturales, y luego el -1 de los enteros con el 1 de los naturales, el 1 de los enteros con el 2 de los naturales, y seguimos así en adelante. Podremos darnos cuenta de que es posible hacer corresponder uno a uno los elementos de ambos conjuntos.

Algunos conjuntos que son frecuentemente utilizados en matemáticas discretas son:

$\mathbb{B}$: el conjunto de los valores booleanos, es decir {`True`,`False`}.
$\mathbb{Z}$: el conjunto de los números enteros.
$\mathbb{Z}^+$: el conjunto de los números enteros positivos.
$\mathbb{Z}^*$: el conjunto de los números enteros no negativos [incluyen al 0].
$\mathbb{N}$: el conjunto de los números naturales, es decir $\{1,2,3,\ldots\}$.
$\mathbb{N}^*$: el conjunto de los números naturales incluyendo al 0, es decir $\{0,1,2,3,\ldots\}$.
$\mathbb{N}_n$: el conjunto de los números naturales desde el 1 hasta n, es decir, $\{1,2,\ldots,n\}$.
$\mathbb{N}^*_n$: el conjunto de los números naturales desde el 0 hasta $n-1$, es decir, $\{0,1,\ldots,n-1\}$.
$\mathbb{R}$: el conjunto de los números reales.

3.4.1 Conjuntos finitos e infinitos

Casi al final de la sección anterior enunciamos un ejemplo de cuando ya no es práctico hacer corresponder los elementos de un conjunto [el saco de arroz], por lo que introducimos la siguiente definición.

Si un conjunto A es equinumeroso al conjunto $\mathbb{N}_n$, entonces es un conjunto **finito**, de otro modo, A es un conjunto **infinito**

El símbolo ∞ sirve para representar el infinito. Así, si A es finito, lo podemos escribir como $|A| < \infty$, de lo contrario podemos escribir $|A| = \infty$ para decir que A tiene una infinidad de elementos. En ocasiones la cardinalidad de un conjunto finito es relativamente pequeña, en esos casos podemos escribir $|A| \ll \infty$.

Hay conjuntos que, aunque son técnicamente finitos, la cantidad de elementos que contiene es tan grande que para fines prácticos pueden ser considerados como infinitos.

Ejemplo 3.7

Los siguientes son ejemplos de conjuntos infinitos.

- El conjunto de los números enteros pares.
- El conjunto de los números reales dentro del intervalo semiabierto $[0.0, 1.0)$.
- El conjunto de granos de arena en el desierto es un conjunto finito, pero tan grande que puede decirse que es prácticamente infinito.

Aunque la cardinalidad de un conjunto es un número, en la expresión $|A| = \infty$, ∞ no es un número. Se debe entender que A tiene una cantidad de elementos tan grande, que no alcanzan los números para hacer corresponder la cardinalidad con alguno.

En programación, el infinito tiene una representación que depende del lenguaje que se utilice, a veces el lenguaje no ofrece un símbolo primitivo que esté asociado con las propiedades de ∞. El caso de `Python` es particular, porque aún cuando no hay una constante que tenga las propiedades de ∞, si es posible trabajar con infinito.

```
infPos = float('inf')
```

En la expresión anterior, se ha creado una variable llamada `infPos` que está ligada a lo que en `Python` se conoce como infinito, que es una instancia de la clase `float`, pero no es un número con el que se puedan hacer operaciones aritméticas, veamos qué sucede:

```
>>> infPos = float('inf')
>>> infPos
inf
>>> type(infPos)
<class 'float'>
>>> isinstance(infPos, float)
True
>>> infPos * 10.0
inf
>>> infPos / infPos
nan
>>> infPos + 100.0
inf
>>> infPos - 100.0
inf
>>> infPos - infPos
nan
>>> infPos > infPos
False
>>> infPos > 1000000000
True
>>>
```

Como observamos en las interacciones, `infPos` adquiere el valor del infinito positivo, que es una instancia de la clase `float`, pero a diferencia de los números de punto flotante, el producto de `infPos` por cualquier otro número es `inf` [infinito], dividir `infPos` entre sí mismo, no es la unidad, sino `nan` que se refiere a un objeto que no es un número [del inglés *Not a number*], eso sí, puede ser comparado en magnitud con los números, pero `infPos` será mayor que cualquier constante numérica, pero no es mayor que sí misma.

3.4.2 Conjuntos numerables e innumerables

Cuando los elementos de un conjunto tienen una correspondencia de uno con los elementos de $\mathbb{N}$, podemos decir que el conjunto es **numerable**, también le podemos decir «contable»; si el conjunto no es numerable entonces es **innumerable**.

Ejemplo 3.8

El conjunto de personas vivas en el mundo es un conjunto numerable, porque es técnicamente posible asignar a cada persona un número entero desde 1 en adelante.

El conjunto de gotas de agua en el mar es un conjunto innumerable, porque no se puede asociar un número natural a cada gota. Suponiendo que se toma una pipeta y se toma una gota y se le asigna un número natural, esa gota puede ser dividida en dos gotas más pequeñas, y estas en otras gotas más pequeñas, haciendo que la asignación fracase.

En la práctica conjuntos como los granos de sal en un saco y el polvo de azúcar son innumerables.

Para ser un poco más precisos, se ha hecho una subclasificación. Si el conjunto es finito y numerable, podemos decir que el conjunto es **cuando mucho numerable**; si el conjunto es infinito y numerable decimos que es numerable.

Ejemplo 3.9

- El conjunto $\{x|5 \leq x \leq 10\}$, donde x es un número entero, es un conjunto cuando mucho numerable.
- El conjunto $\{x|5 \leq x\}$, donde x es un número entero, es un conjunto numerable.
- El conjunto $\{x|5 \leq x\}$, donde x es un número real, es un conjunto infinito e innumerable.

3.4.3 Cardinalidad en conjuntos explícitos e implícitos

La manera de determinar la cardinalidad de un conjunto es asociar cada elemento del conjunto con un elemento de los números naturales.

A		$\mathbb{N}$
a_1	$\mapsto$	1
a_2	$\mapsto$	2
$\vdots$		$\vdots$
a_n	$\mapsto$	n
		$\vdots$

En los conjuntos explícitos donde sus elementos son nombrados uno a uno, el procedimiento para calcular su cardinalidad consiste asociar un conjunto con un número

natural, quitar un elemento del conjunto y al conjunto que resta, asociarlo con el número siguiente y así recursivamente hasta terminar con el conjunto vacío.

Para contar los elementos de un conjunto finito se requiere:

1. Un conjunto A definido de manera explícita, que puede ser $\varnothing$ o no.
2. Una variable n que es un número natural cuyo valor inicial debe ser 0. Tener parámetros con la finalidad de construir progresivamente la respuesta, es un modo de crear procesos iterativos dentro de funciones recursivas [ASS96, p. 31].

El procedimiento asocia a cada conjunto evaluado, un valor del conjunto de los naturales. El valor asociado al conjunto vacío debe ser cero. El procedimiento es recursivo y es como sigue:

Primero. En caso de que $A = \varnothing$, la cardinalidad está almacenada en n, cuyo valor inicial es 0.

Segundo. En otro caso [*eoc*], esto es cuando $A \neq \varnothing$, el conjunto se puede representar como $\{a_0|A'\}$ es necesario recursar el algoritmo calculando ahora la cardinalidad de A', pero empezando la cuenta en $n+1$, que representa una nueva asociación de un nuevo elemento del conjunto con un nuevo número entero.

$$|A,[n=0]| \mapsto \begin{cases} n & \text{si } A = \varnothing. \\ \\ |A', n+1| & eoc. \end{cases}$$

Código 3.6: *La cardinalidad de un conjunto*

```
1  def card(A:list, n:int=0)-> int:
2      """
3      Calcula la cardinalidad de un conjunto dado por extensión.
4      """
5      if esVacio(A):
6          return n
7      else:
8          return card(cdr(A), n+1)
```

Las funciones en `Python` pueden ser definidas usando argumentos posicionales y argumentos opcionales con palabras clave.

Los parámetros con palabra clave se colocan después de los parámetros posicionales. Observa los siguientes ejemplos:

```
def foo(a:int, b:int=2, c:int=3)-> int:
    print(f"a={a}; b={b}; c={c}")
    return b * (a - c)
```

En este ejemplo, los parámetros formales son `a`, que es un argumento posicional entero; `b` que es un argumento tipo *palabra clave* que es un entero; y `c` también es un argumento tipo *palabra clave* entero. Los **argumentos actuales** aparecen en la invocación de la función. Siempre deben aparecer en primer lugar los valores para los parámetros posicionales y luego los valores para los parámetros de tipo palabra clave, que pueden aparecer en cualquier orden, siempre y cuando se escriba qué parámetro debe recibir el valor.

```
>>> foo(4)
a=4; b=2; c=3
2
>>> foo(4,3,5)
a=4; b=3; c=5
-3
>>> foo(4,5,3)
a=4; b=5; c=3
5
>>> foo(4,c=5,b=3)
a=4; b=3; c=5
-3
>>>
```

Ejemplo 3.10
Utiliza la función `card` para calcular la cardinalidad del conjunto $A \leftarrow \{1, 2, 3, x, y, z\}$.

```
>>> A = [1, 2, 3, 'x', 'y', 'z']
>>> card(A)
6
>>>
```

Así hemos calculado $|A| \mapsto 6$.

En el caso de los conjuntos que son implícitamente definidos, el procedimiento tiene el mismo principio, asociar cada elemento del conjunto con un elemento del conjunto de los números naturales.

Si la asociación de elementos–números termina en el número n, entonces la cardinalidad es finita y es precisamente ese número n. Si la asociación no termina, entonces el conjunto es infinito, pero numerable. El símbolo generalmente empleado para denotar el infinito es ∞.

Hay conjuntos cuyos elementos no pueden ser relacionados uno a uno con los números naturales, por ejemplo los números reales.

Es usual escribir $n < \infty$ para denotar que literalmente, el número n es menor que infinito, pero lo que realmente se quiere decir es que n es un número finito.

Si $|A| \mapsto n$, decimos que A es de **orden** n; así $\varnothing$ es de orden 0, $\{a\}$ es de orden 1, $\{1, 2, \ldots\}$ es de orden ∞ [MH81, p. 8].

3.4.4 Conjunto unitario

Definición 3.4.3 -- Conjunto unitario. Si A es un conjunto de tal manera que $|A| \mapsto 1$, decimos que A es unitario.

Cualquier conjunto unitario es de orden 1.

Es bueno contar con un predicado que determine si un conjunto es unitario o no lo es. La siguiente función recibe como argumento un conjunto [dado en forma de lista] y devuelve un valor booleano, que será `True` si el conjunto es unitario y `False` si no lo es.

Código 3.7: *Determinar si un conjunto es unitario*

```
1 def esUnit(A:list)-> bool:
2     """ Determina si un conjunto es unitario.
3     Un conjunto es unitario si tiene exactamente
4     un elemento.
5     """
6     return card(A) == 1
```

Ejemplo 3.11
Utiliza la función `esUnit` para verificar que $\{\langle 0 \rangle\}$, $\{s\}$ y $\{\varnothing\}$ sean conjuntos unitarios.

```
>>> esUnit(conj([0]))
True
>>> esUnit(conj('s'))
True
>>> esUnit(conj(vacio))
True
>>>
```

3.5 Subconjuntos

Definición 3.5.1 -- Subconjunto. Si A y B son conjuntos, decimos que el conjunto A es subconjunto del conjunto B y se denota como $A \subseteq B$, si se cumple

$$\forall a \in A : a \in B$$

Observa que la definición de subconjunto se ha escrito en términos de un cuantificador universal, lo que permite saber que el resultado es un valor booleano, por lo que $A \subseteq B$ es una proposición una vez determinados los valores para A y B.

Esta definición puede ser implementada en Python considerando la pertenencia de un elemento a un conjunto [sección 3.3, página 72], y un cuantificador universal [definición 2.3.1, página 54], ya que se debe verificar la pertenencia *para todo* elemento.

Código 3.8: *Subconjunto de un conjunto*

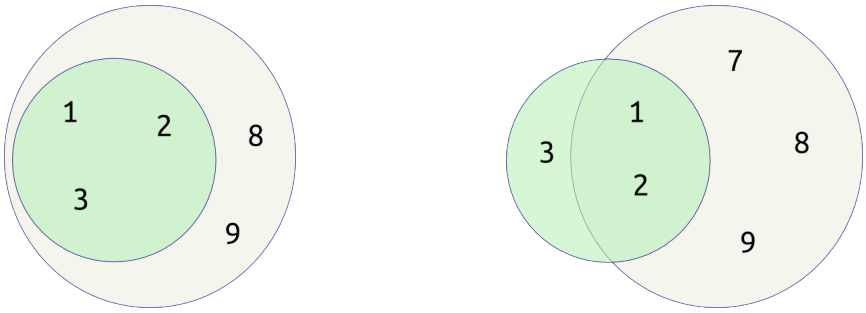
```
1 def esSubc(A:list, B:list)-> bool:
2     """ A es subconjunto de B.
3     Determina si A es subconjunto de B
4     si todos los elementos de A
5     pertenecen también al conjunto B.
6     """
7     return paraTodo(lambda a: en(a, B), A)
```

Ejemplo 3.12

Si $A \leftarrow \{1,2,3\}$ y $B \leftarrow \{3,8,2,9,1\}$ son conjuntos, A es subconjunto de B porque todos los elementos de A pertenecen también a B.

Si ahora $A \leftarrow \{1,2,3\}$ y $B \leftarrow \{1,2,7,8,9\}$, A no es subconjunto de B, porque no todos los elementos de A pertenecen también a B.

```
>>> A = [1,2,3]
>>> B = [3,8,2,9,1]
>>> esSubc(A,B)
True
>>> B = [1,2,7,8,9]
>>> esSubc(A,B)
False
>>>
```



(a) $A \leftarrow \{1,2,3\}$ *y* $B \leftarrow \{3,8,2,9,1\}$ ***(b)*** $A \leftarrow \{1,2,3\}$ *y* $B \leftarrow \{1,2,7,8,9\}$

Figura 3.3: *Diagramas de Euler para mostrar las condiciones de los conjuntos, en (a)* $A \subseteq B \mapsto$ `True`*; mientras que en (b)* $A \subseteq B \mapsto$ `False`*, esto es porque no todos los elementos de A pertenecen a B.*

Si no es verdad que $A \subseteq B$, entonces se verifica que $\neg(A \subseteq B)$, lo que podemos escribir como $A \not\subseteq B$.

Ejemplo 3.13

En la figura 3.3b, cuando $A \leftarrow\!\shortmid \{1,2,3\}$ y $B \leftarrow\!\shortmid \{1,2,7,8,9\}$, se observa que $A \nsubseteq B$, porque $3 \in A$, pero $3 \notin B$. Al mismo tiempo $B \nsubseteq A$ por razones similares, ya que tiene elementos que no pertenecen al conjunto A.

Otras observaciones importantes que son válidas para los subconjuntos son:

- $A \subseteq B \rightarrow |A| \leq |B|$.
- $|A| > |B| \rightarrow A \nsubseteq B$.
- $\varnothing \subseteq A$, para cualquier conjunto A. La razón es porque la definición exige que si a es un elemento de $\varnothing$, entonces también debe ser un elemento de A. Claramente no hay elementos en $\varnothing$, pero la implicación sigue siendo `True` a pesar de que el antecedente sea `False`.
- $A \subseteq A$, para cualquier conjunto A. Esto es porque todos los elementos de A son elementos de él mismo.

Por otro lado, si $A \subseteq B$, también podemos decir que B es un **superconjunto** de A, y lo podemos denotar $B \supseteq A$. La siguiente proposición es una tautología $A \subseteq B \leftrightarrow B \supseteq A$.

Ejemplo 3.14

La taxonomía de los animales es un claro ejemplo de superconjuntos. Considera la familia *Félidos* (felinos), la subfamilia *Felinos*, como los gatos domésticos, ocelotes, tigrillos, pumas, chitas o cervales; y la subfamilia *Panterinos* como el tigre, el león, el leopardo, la pantera y el jaguar.

Aquí los *Félidos* son un superconjunto de los *Felinos* y de los *Panterinos*.

Observa que la función `esSubc` recibe como primer argumento un conjunto en forma de lista, por lo que puede recibir el resultado de crear un conjunto dando explícitamente los elementos que contiene [ver página 71], o bien dando un subconjunto de elementos tomados de otro conjunto de referencia [ver página 84].

Ejemplo 3.15

Considera los siguientes conjuntos:

$$B \leftarrow\!\shortmid \{n | 1 \leq n \leq 10\}$$
$$C \leftarrow\!\shortmid \{1,2,3,4,5,6,7,8,9,10\}$$
$$A_1 \leftarrow\!\shortmid \{4,5,6,7,8\}$$
$$A_2 \leftarrow\!\shortmid \{x \in C : 4 \leq x \leq 8\}$$

Utiliza la función `esSubc` para valorar las siguientes expresiones: $A_1 \subseteq B$ y $A_2 \subseteq B$. Esto nos servirá para mostrar cómo trabaja la función usando conjuntos definidos de ambas maneras. Luego repite el proceso con las siguientes expresiones $A_1 \subseteq C$ y $A_2 \subseteq C$.

Observa que B está definido de manera intencional, mientras que C está definido de manera extensional.

```
>>> B = conj(lambda n: 1 <= n <= 10)
>>> C = conj(1,2,3,4,5,6,7,8,9,10)
>>> A1 = conj(4,5,6,7,8)
>>> A2 = subc(lambda x:4<= x <= 8, C)
>>> esSubc(A1,B)
True
>>> esSubc(A2,B)
True
>>> esSubc(A1,C)
True
>>> esSubc(A2,C)
True
>>>
```

3.5.1 Construcción de subconjuntos

Una manera implícita de crear subconjuntos es muy similar a la forma de crear conjuntos de manera implícita. La diferencia es que ahora se especifica de qué conjunto son tomados los elementos que formarán el subconjunto.

Ejemplo 3.16

El subconjunto de los números pares tomados del conjunto de números del 0 al 10 inclusive. Este subconjunto contiene los números 2, 4, 6, 8 y 10.

Aquí se toma como referencia el conjunto $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, de los cuales solamente se tomarán los pares.

Para construir el subconjunto se requieren dos elementos:

1. Un conjunto de referencia D.
2. Un predicado P.

El procedimiento es tomar uno a uno los elementos del conjunto D y seleccionar solamente aquellos que hacen que el predicado P sea `True`. Esta acción puede verse como un tamiz o filtro, que solamente deja pasar aquellos elementos que cumplen la condición establecida.

Al crear subconjuntos de esta manera, tenemos que escribir tanto el conjunto de referencia como el predicado o propiedad [Sto63, p. 7], siguiendo el siguiente patrón

$$\{x \in D | P(x)\}.$$

Observa lo similar que es esta notación con la manera de escribir conjuntos de manera implícita $\{x|P(x)\}$ o bien $\{x : P(x)\}$ [Das14, p. 4].

Ejemplo 3.17

La siguiente base de datos, llamada `ASP`, corresponde a las personas que se han inscrito como aspirantes a un puesto importante. Se les ha solicitado su edad y nacionalidad. Sin embargo no todos son elegibles, pues los requisitos son, que la edad sea mayor o igual a 18 y sean de nacionalidad mexicana. Se desea entonces hacer un subconjunto con las personas seleccionables.

Índice	Nombre	Nacionalidad	Edad
1	John Mooney	Chile	17
2	Roth Oneil	Colombia	21
3	Shelly Keith	Chile	23
4	Graham Odom	Costa Rica	17
5	Tobias Benson	México	19
6	Tad Rodgers	Chile	30
7	Isabelle Morgan	México	16
8	Breanna Huffman	Costa Rica	21
9	Halla O'connor	México	21
10	Wylie Simon	México	26
11	Ralph Barrera	México	26
12	Carol Clemons	Chile	23

$$\{p \in \texttt{ASP} : \texttt{nac}(p) =' Mexico' \wedge \texttt{edad}(p) \geq 18\},$$

donde $nac(p)$ es una función que recibe uno de los registros y verifica que la nacionalidad sea igual a $'Mexico'$, y $edad(p)$ es una función que toma un registro p y verifica que la edad sea mayor o igual a 18. El subconjunto contiene los registros 5,9,10 y 11.

Índice	Nombre	Nacionalidad	Edad
5	Tobias Benson	Mexico	19
9	Halla O'connor	Mexico	21
10	Wylie Simon	Mexico	26
11	Ralph Barrera	Mexico	26

Código 3.9: *Construcción de subconjuntos*

```
def subc(P:callable, D:list)-> list:
    """ Crea un subconjunto.
    Construye un subconjunto de D con aquellos
    elementos que verifican el predicado P.
    """
    return list(filter(P, D))
```

filter(⟨*callable*⟩, ⟨*iterable*⟩): Genera un nuevo iterable con aquellos elementos de ⟨*iterable*⟩ que devuelven `True` al ser evaluados con el ⟨*callable*⟩.

Una función o una expresión lambda son ejemplos de objetos que pueden ser invocados (`callable`).

Un ⟨*iterable*⟩ es un objeto que se puede iterar, como las listas, las tuplas y las cadenas de caracteres.

Ejemplo 3.18

Utiliza la función `subc` para crear el subconjunto A de la figura 3.4.

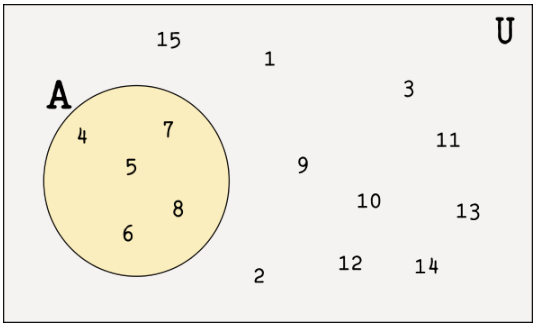


Figura 3.4: $U \leftarrow \{n | 1 \leq n \leq 15\}$

Observamos que U contiene los números desde 1 hasta 15 inclusive, este es el conjunto de referencia. El predicado que debemos utilizar, al ser evaluado con los elementos de U, debe resultar `True` solamente en aquellos elementos de A, por lo que $P \leftarrow \lambda n \bullet 4 \leq n \leq 8$ debe funcionar bien. Así el subconjunto es:

```
A = subc(lambda n: 4 <= n <= 8, U)
```

```
>>> U = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
>>> A = subc(lambda n: 4 <= n <= 8, U)
>>> A
[4, 5, 6, 7, 8]
>>>
```

3.5.2 Subconjuntos propios

Definición 3.5.2 -- Subconjunto propio. Si $A \subseteq B$ y $\exists b \in B : b \notin A$, decimos que A es un subconjunto propio de B, y lo denotamos $A \subset B$.

Esta es justo la situación que se aprecia en la figura 3.3a, notamos que efectivamente $A \subseteq B$, pero los elementos 8 y 9 que son elementos de B, no pertenecen a A.

Cuando no es verdad que $A \subset B$, lo escribimos $A \not\subset B$. Debe ser claro que si $A \subset B$, entonces $|A| < |B|$.

Otras observaciones referentes a los subconjuntos propios son las siguientes:

- $A \neq \varnothing \leftrightarrow \varnothing \subset A$
- $A \not\subset A$, para cualquier conjunto A.

Vamos a escribir un predicado que determine si un conjunto es subconjunto propio de otro conjunto.

Para esto se requiere verificar dos cosas de acuerdo con la definición:

1. Que A sea un subconjunto de B
2. Que exista al menos un elemento de B que no pertenezca a A.

La salida debe ser `True` si A es subconjunto propio de B y `False` en el otro caso.

Código 3.10: *Subconjunto propio de un conjunto*

```
1  def esSubcP(A:list, B:list)-> bool:
2      """ Determina si A es subconjunto propio de B.
3      Un conjunto A es subconjunto propio de B si
4      A es subconjunto de B y existe al menos un elemento de B que
5      no se encuentre en A.
6      """
7      res = y(esSubc(A,B), existeUn(lambda b:neg(en(b,A)),B))
8      return res
```

Ejemplo 3.19

Considera los conjuntos $A_1 \leftarrow \{1,2,3\}$, $A_2 \leftarrow \{3,8,2,9,1\}$ y $B \leftarrow \{1,2,3,9,8\}$. Con la ayuda de la función `esSubcP`, determina el valor de verdad de $A_1 \subset B$ y $A_2 \subset B$.

```
>>> A1 = [1,2,3]
>>> B = [1,2,3,9,8]
>>> esSubcP(A1, B)
True
>>> A2 = [3,8,2,9,1]
>>> B = [1,2,3,9,8]
>>> esSubcP(A2, B)
False
>>>
```

3.6 Igualdad de conjuntos

Dos conjuntos A y B son iguales, denotado $A = B$, si tienen exactamente los mismos elementos.

Para determinar que dos conjuntos son iguales debemos hacer dos cosas [Sto63]:

1. Todos los elementos de A deben pertenecer también a B, esto es $A \subseteq B$ [figura 3.5 izquierda].
2. Asegurar que no existan elementos de B que no pertenezcan al conjunto A [figura 3.5 derecha].

La primera condición se verifica con $A \subseteq B$, ya que por definición, significa verificar que todos los elementos de A pertenecen a B.

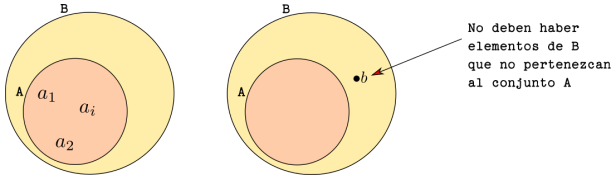


Figura 3.5: *Condiciones para determinar la igualdad entre conjuntos.*

La segunda condición puede establecerse formalmente como:

$$\neg\exists b \in B : b \notin A \tag{3.1}$$

Al trabajar un poco con la expresión 3.1, podemos encontrar una expresión equivalente que no tenga negaciones al inicio:

$$\begin{aligned} \neg\exists b \in B : b \notin A &= \forall b \in B : \neg(b \notin A) \\ &= \forall b \in B : \neg(\neg(b \in A)) \\ &= \forall b \in B : b \in A, \end{aligned}$$

pero $\forall b \in B : b \in A$ significa que todos los elementos de B deben ser también elementos de A, que justo es la definición de $B \subseteq A$. Con las dos condiciones se forma una conjunción lógica que será utilizada para la siguiente definición:

Definición 3.6.1 -- Conjuntos iguales. Dos conjuntos A y B son iguales, denotado $A = B$, si se cumple que

$$A \subseteq B \wedge B \subseteq A$$

Código 3.11: *Conjuntos iguales*

```
def cIguales(A:list, B:list)-> bool:
    """ Igualdad en conjuntos.
    Determina si A es igual a B
    determinando si A es subconjunto de B y
    B es subconjunto de A.
    """
    return y(esSubc(A,B), esSubc(B,A))
```

Ejemplo 3.20

Considera los siguientes conjuntos:

$$\begin{aligned} U &\leftarrow \{1,2,3,4,5,6,7,8,9,10\} \\ A_1 &\leftarrow \{4,5,6,7,8\} \\ A_2 &\leftarrow \{x \in U : 4 \leq x \leq 8\} \end{aligned}$$

Utiliza la función `cIguales` para valorar la expresión: $A_1 = A_2$. Observa que A_2 está definido de manera intencional, mientras que A_1 está definido de manera extensional.

```
>>> U = conj(1,2,3,4,5,6,7,8,9,10)
>>> A1 = conj(4,5,6,7,8)
>>> A2 = subc(lambda x:4<= x <= 8, U)
>>> cIguales(A1,A2)
True
>>>
```

La operación de igualdad en los conjuntos es conmutativa, lo que permite que $A = B$ tenga el mismo valor de verdad que $B = A$. Esto se verifica después de notar que la conjunción lógica también es conmutativa con las proposiciones lógicas.

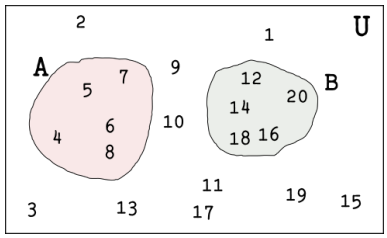
Ejemplo 3.21

Considera los mismos conjuntos del ejemplo anterior.

```
>>> cIguales(A1,A2)
True
>>> cIguales(A2,A1)
True
>>>
```

Ejercicios

1. Considera el universo $U \leftarrow \{n \in \mathbb{Z}^* | 1 \leq n \leq 20\}$. Dibuja diagramas de Euler para los siguientes subconjuntos de U.
   a) *Ejemplo:* $A \leftarrow \{a \in U | 4 \leq a \leq 8\}$, $B \leftarrow \{b \in U | \texttt{esPar}(b) \wedge b \geq 12\}$



   b) $A \leftarrow \{a \in U | 4a < 20\}$; $B \leftarrow \{b \in U | 1 < b < 15\}$
   c) $C \leftarrow \{a \in U | a > 10\}$; $D \leftarrow \{b \in U | 2b < 20\}$
   d) $E \leftarrow \{x \in U | 3x < 10\}$; $F \leftarrow \{y \in U | 3x > 10\}$
2. En las siguientes expresiones en `Python` que definen subconjuntos, identifica el predicado; el conjunto de referencia y calcula la cardinalidad del subconjunto.
   a) *Ejemplo:* `subc(lambda x: 0 < x < 100, [-20, -30, -40, -50])`
   El predicado es $\lambda x \cdot 0 < x < 100$
   y el conjunto de referencia es $\{-20, -30, -40, -50\}$.
   b) `subc(lambda x: -25 < x, [-20, -30, -40, -50])`
   c) `subc(lambda x: (x * x) < 1000, [10, 20, 30, 40, 50, 60])`
   d) `subc(lambda x: 3*x**2+2*x < 500, [2,4,5,6,7,12,31])`
3. Considera $U \leftarrow \{0, 1, 2, \ldots, 50\}$.
   a) Crea una definición en el lenguaje `Python` del conjunto U.
   b) Llena la siguiente tabla, poniendo en una columna la notación de subconjuntos de U y en otra columna la notación correspondiente en `Python`. Observa el ejemplo.

	Notación de conjuntos	Notación de `Python`
El subconjunto de números mayores que 28	$\{n \in U \mid n > 28\}$	`subc(lambda n: n>28, U)`
El subconjunto de U con los números entre el 5 y 48 inclusive.		
El subconjunto de U con los números entre el 3 y 42 que sean múltiplos de 5.		
El subconjunto de U con los números primos.		

4. Dado el conjunto de referencia $U \leftarrow \{2, 4, 7, 34, 12, 8, 16, 23\}$, calcula los siguientes conjuntos. Si hay algún error, identifícalo y escribe cual es.
   a) *Ejemplo:* El conjunto $x|10 \leftarrow x$.
   Esta expresión no sigue la sintaxis de un conjunto, que debe ser de la forma $\{x \in D | P(x)\}$. Quizá se quería escribir $\{x \in U | 10 < x\}$.
   b) El conjunto $\{x \in U | x < 15 \rightarrow \texttt{esImpar}(x)\}$ [ver la página 38.]
   c) El conjunto $\{y \in U | x < 10 \oplus \texttt{esPar}(x)\}$ [ver la página 37.]
   d) El conjunto $\{x \in H | 5x^2 + 4x - 3\}$
5. Considera el conjunto $A \leftarrow \{x | \texttt{esEntero}(x) \wedge 0 < 3x \leq 30\}$. Calcula el valor de verdad de las siguientes proposiciones:

a) *Ejemplo:* $50 \in A \mapsto$ `False`

b) $\{2,4,6\} \subseteq A \mapsto$ ☐

c) $\varnothing \subseteq A \mapsto$ ☐

d) $\{x|0 < 2x < 15\} \subseteq A \mapsto$ ☐

e) $\{19,21,32,14,28,4\} \subseteq A \mapsto$ ☐

6. Los siguientes conjuntos tienen elementos que «esconden» una propiedad. Escribe el conjunto en la forma $\{x|P(x)\}$, escribiendo la propiedad de los elementos del conjunto en forma de un predicado.

Ejemplo: $\{2,4,6,8,10\}$	$\{x\|\texttt{esPar}(x) \wedge 1 < x \leq 10\}$
$\{1,3,5,7,9\}$	
$\{1,4,9,16,25,36,49\}$	
{"11001", "11010", "110", "1100101"}	

Operaciones con conjuntos 4

En este capítulo retomaremos las definiciones de los capítulos anteriores, especialmente aquellas de teoría de conjuntos y operadores lógicos, para crear nuevos operadores que permitan manipular conjuntos que los haga útiles para tareas específicas.

4.1 Agregar un elemento a un conjunto

Empezamos este capítulo con una operación que en la teoría de conjuntos no es muy frecuente o se da por sentada, pero computacionalmente es necesario construir.

Definición 4.1.1 -- Agregar. La operación agregar genera un nuevo conjunto que contiene los elementos del conjunto A junto con el elemento e, siempre que éste no pertenezca a A. Esta operación se puede escribir como $e \triangleright A$.

$$e \triangleright A \mapsto \begin{cases} \text{Si } e \in A & , A \\ \text{Si } e \notin A & , \{e|A\}. \end{cases}$$

Cuando $e \notin A$ y después de hacer $e \triangleright A$, el elemento e ahora debe pertenecer al conjunto A, se puede pensar que e sea el primer elemento, computacionalmente es mas conveniente que sea el último y A el conjunto [posiblemente vacío] que contenga al resto de los elementos del nuevo conjunto. En general, para cualquier conjunto A, $|e \triangleright A| = |A| + 1$.

Ejemplo 4.1

Sea $A \leftarrow \{2, 3, 4, 5\}$ y considera los elementos $e_1 \leftarrow 1$ y $e_2 \leftarrow 2$.

$e_1 \triangleright A \mapsto e_1 \triangleright \{2, 3, 4, 5\} \mapsto 1 \triangleright \{2, 3, 4, 5\} \mapsto \{2, 3, 4, 5, 1\}$.

$e_2 \triangleright A \mapsto e_2 \triangleright \{2, 3, 4, 5\} \mapsto 2 \triangleright \{2, 3, 4, 5\} \mapsto \{2, 3, 4, 5\}$.

En Python se puede modelar la operación agregar ocupando de nueva cuenta el predicado de pertenencia [definición 3.3.1, página 72], que permitirá incluir el nuevo elemento e siempre y cuando no pertenezca al conjunto A.

La nueva función `agrega` requiere dos parámetros formales, un elemento cualquiera `e` y un conjunto `A`. Esta nueva función requiere una expresión condicional `if` y claro, un predicado que sea evaluado con los argumentos para determinar si el elemento `e` pertenece o no al conjunto `A`.

```
def agrega(e, A:list)-> list:
    """
    Agrega un elemento e al conjunto A.
    El elemento es agregado cuando no pertenece al conjunto.
    Si el elemento ya pertenece, el conjunto permanece sin cambio.
    """
    R = R = list(A) if en(e,A) else list(A)+[e]
    return R
```

```
>>> A = []
>>> agrega(1, A)
[1] # este es un nuevo conjunto, diferente al conjunto A
>>> agrega(2, A)  # aquí A no se ha modificado
[2]
>>> A = agrega(1, A) # aquí sí se modifica el conjunto A
>>> A = agrega(2, A) # se modifica nuevamente el conj. A
>>> A
[1, 2]
>>>
```

El operador + tiene muchas aplicaciones, la más común es la de sumar números, pero también se puede utilizar con listas, siguiendo el siguiente patrón:

⟨*lista1*⟩+⟨*lista2*⟩[+...]

Lo que tiene el efecto de concatenar las listas, generando una nueva lista que contiene los elementos de ⟨*lista1*⟩ seguidos por los elementos de ⟨*lista2*⟩ sin alterar el orden de los elementos de las listas.

```
>>> []+[]
[]
>>> []+[1,2]
[1, 2]
>>> [1,2]+[]
[1, 2]
>>> [1,2]+[]+[3,4]
[1, 2, 3, 4]
>>>
```

Ejemplo 4.2

En la Ciudad de México se está levantando un padrón para un programa social. Al inicio el padrón se encuentra vacío, pero pronto se agregan nuevos nombres, sin embargo puede ocurrir que alguien trate de enlistarse más de una vez. La secuencia de personas agregadas fueron Antonio, Gustavo, Andrea, Antonio, Felipe. Esto se puede modelar con un conjunto del siguiente modo:

$$\begin{aligned} padron &\leftarrowtail \varnothing \\ padron &\leftarrowtail Gustavo \triangleright padron \\ padron &\leftarrowtail Andrea \triangleright padron \\ padron &\leftarrowtail Antonio \triangleright padron \end{aligned}$$

Actualmente $padron \mapsto \{Gustavo, Andrea, Antonio\}$, al tratar de agregar nuevamente el nombre de *Antonio*, el padrón permanece sin efecto.

$$\begin{aligned} padron &\leftarrowtail Gustavo \triangleright padron \\ padron &\mapsto \{Gustavo, Andrea, Antonio\} \end{aligned}$$

Finalmente se agrega el nombre de *Felipe* al padrón.

$$padron \leftarrow Felipe \triangleright padron$$
$$padron \mapsto \{Gustavo, Andrea, Antonio, Felipe\}$$

```
>>> padron = conj()
>>> padron = agrega('Gustavo', padron)
>>> padron = agrega('Andrea', padron)
>>> padron = agrega('Antonio', padron)
>>> padron
['Gustavo', 'Andrea', 'Antonio']
>>> padron = agrega('Antonio', padron)
>>> padron
['Gustavo', 'Andrea', 'Antonio']
>>> padron = agrega('Felipe', padron)
>>> padron
['Gustavo', 'Andrea', 'Antonio', 'Felipe']
>>>
```

4.2 Unión

La unión de conjuntos es una operación que tiene mucha importancia computacional, ya que se utiliza para reunir en una sola base de datos, los registros que aparecen en dos o más bases de datos diferentes.

La unión de los conjuntos A y B se escribe $A \cup B$ y es el nuevo conjunto $\{x | x \in A \vee x \in B\}$ [Hal74].

Así, el conjunto resultante tendrá como miembros a aquellos elementos que pertenezcan al menos a uno de los conjuntos participantes en la unión, como se muestra en la tabla de verdad del predicado en la declaración de la unión [página 94].

Ejemplo 4.3

Dos personas buscaron en Internet autos usados a un buen precio. Uno de ellos buscó autos de modelo 2015 en adelante, el otro buscó autos con transmisión estándar.

1. Podemos decir que la primera persona obtuvo un conjunto de páginas web que contenían información sobre autos cuyo modelo es mayor o igual a 2015. Este conjunto puede ser establecido como:

   $$A \leftarrow \{x | \texttt{modelo}(x) \geq 2015\},$$

   asumiendo que $\texttt{modelo}(x) \in \mathbb{N}^*$ para alguna información de auto x.
2. La segunda persona obtuvo información sobre autos con transmisión automática, lo que se puede modelar como:

   $$B \leftarrow \{x | \texttt{esEstandar}(x)\},$$

   asumiendo que $\texttt{esEstandar}(x) \in \mathbb{B}$ para alguna información de auto x.
   Así, $A \cup B$ es el conjunto de páginas con información de autos que podemos modelar como

   $$A \cup B \leftarrow \{x | \texttt{modelo}(x) \geq 2015 \vee \texttt{esEstandar}(x)\}.$$

La tabla de verdad de la disyunción es una herramienta muy útil para determinar que la única manera en que un elemento x no pertenezca a la unión del conjunto A con el conjunto B, es que $x \notin A$ y al mismo tiempo $x \notin B$.

$x \in A$	$x \in B$	$x \in A \vee x \in B$
True	True	True
True	False	True
False	True	True
False	False	False

Utilizando los diagramas de Euler, podemos identificar la unión de dos conjuntos como la zona coloreada que incluye tanto los elementos del conjunto A como aquellos del conjunto B, observa la figura 4.1.

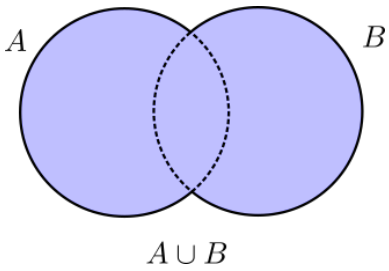


Figura 4.1: *La unión de conjuntos*

$A \cup B$ definido como $\{x | x \in A \vee x \in B\}$ es un ejemplo de una **definición declarativa**, porque establece las condiciones o características que un objeto x debe tener para ser considerado elemento del nuevo conjunto. Este tipo de definiciones establecen «qué es el objeto». En ciencias computacionales se prefieren las **definiciones efectivas**, que a diferencia de la anterior, se establece «cómo se obtiene tal objeto».

En computación estamos más interesados en tener un procedimiento efectivo que proporciona una manera con la que se obtiene la unión de los conjuntos A y B.

Para crear la definición efectiva tomaremos como base el hecho de que todos los elementos de uno de los conjuntos pertenecen a la unión. Seleccionaremos A de manera arbitraria ya que sabemos *a priori*, que todos sus elementos [si los tiene], cumplen con $a \in A$, lo que hace que también pertenezcan a la unión. Lo que resta es saber qué elementos de B se deben agregar, puesto que algunos de ellos quizá ya estén siendo considerados.

Así que tomaremos uno a uno los elementos de B, si un elemento b del conjunto B, cumple $b \in A$, entonces no hay necesidad de agregarlo, pero si $b \notin A$, significa que es un nuevo elemento que debe ser considerado en el resultado.

La buena noticia es que el procedimiento `agrega` [definición 4.1.1, página 91], ya hace la tarea de verificar la pertenencia, de modo que simplemente utilizaremos ese procedimiento para agregar un elemento solamente cuando haga falta. Así tenemos la siguiente definición:

Definición 4.2.1 -- Unión. Sean A y B dos conjuntos. La unión de A con B es un nuevo conjunto que escribimos $A \cup B$ y se construye recursivamente:

$$A \cup B \mapsto \begin{cases} A & \text{si } B = \varnothing. \\ (b_0 \triangleright A) \cup B' & eoc. \end{cases}$$

Veamos:

1. Si $B = \varnothing$, el resultado de $A \cup B$, es el mismo conjunto A, sin importar si $A = \varnothing$. Verificar si un conjunto es vacío, en programación es realizado por el predicado `esVacio` [página 75].
2. Si no ocurre lo anterior, se sabe entonces que $B = \{b_0|B'\}$, es decir que B tiene un primer elemento b_0 y un subconjunto B' que contiene a todos los otros elementos. Ahora, si $b_0 \in A \mapsto$ `True`, no hay necesidad de agregarlo y recursivamente se hace la unión de A con B'. Observa que cada invocación recursiva se realiza con una instancia menor de B', lo que garantiza que eventualmente se realice el caso base.

Código 4.1: *Unión de conjuntos*

```
def union(A: list, B: list)-> list:
    """ Union de dos conjuntos
    Genera un conjunto con la unión de A con B.
    Se devuelve el conjunto A junto con los elementos de B que no
    pertenezcan a A.
    """
    if esVacio(B):
        return A
    else:
        return union(agrega(car(B), A), cdr(B))
```

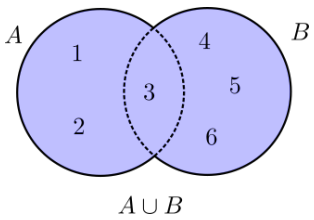


Figura 4.2: $A \leftarrow \{1,2,3\}$, $B \leftarrow \{4,5,3,6\}$. $A \cup B \mapsto \{6,5,4,1,2,3\}$

```
>>> A = [1,2,3]
>>> B = [4,5,3,6]
>>> union(A,[])
[1, 2, 3]
>>>
>>> union(A, B)
[1, 2, 3, 4, 5, 6]
>>>
```

Hay algunas propiedades importantes que tiene el conjunto que resulta de la unión de otros dos conjuntos:

1. $(A \cup \varnothing) = A$ elemento neutro. Es el caso base de la definición.
2. Para cualquier conjunto A, se cumple que $A \cup A \mapsto A$ (idempotencia). De acuerdo con la definición, suponiendo que no es el caso base [propiedad anterior], se agregan aquellos elementos que no pertenezcan a A aunque no se agrega ni uno mas, porque todos ellos pertenecen a A.
3. Siendo A y B dos conjuntos cualesquiera, $A \subseteq A \cup B$ y también $B \subseteq A \cup B$.
4. Si se toma un subconjunto A' de A, entonces $A' \cup A \mapsto A$.
5. $(A \cup B) \cup C = A \cup (B \cup C)$ ley asociativa.
6. $(A \cup B) = (B \cup A)$ ley conmutativa.

4.3 Intersección

La palabra «intersección» tiene sus orígenes en el vocablo latín *intersectio* que significa «punto de encuentro entre dos o más cosas». En la teoría de conjuntos, la intersección es una selección muy particular de elementos, convencionalmente si A y B son dos conjuntos, la intersección de A con B se escribe $A \cap B$ y es el conjunto $A \cap B = \{x | x \in A \wedge x \in B\}$ [Hal74].

Utilizando un diagrama de Euler, podemos ver en la zona sombreada de la figura 4.3, lo que corresponde a la intersección de dos conjuntos.

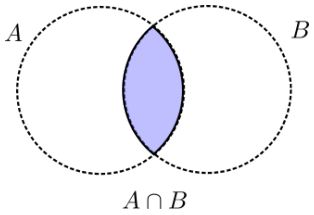


Figura 4.3: *La intersección de conjuntos*

Nuevamente utilizaremos la tabla de verdad con las proposiciones $x \in A$, $x \in B$ y $x \in A \cap B$, que es lógicamente equivalente a la conjunción de las dos primeras proposiciones [ver la equivalencia lógica en la sección 1.7 en la página 41].

$x \in A$	$x \in B$	$x \in A \wedge x \in B$
True	True	True
True	False	False
False	True	False
False	False	False

Observa que la única manera [de las cuatro] en que un elemento x puede pertenecer a la intersección, es que al mismo tiempo pertenezca al conjunto A y al conjunto B.

Ejemplo 4.4

Para una competencia de conocimientos sobre historia se han seleccionado alumnos del nivel licenciatura. Los participantes deben ser aquellos que hayan tomado algún curso de historia de México y que sean mexicanos de nacimiento.

Aquí podemos notar dos conjuntos. Uno, que llamaremos A, es el conjunto de personas que han tomado el curso de historia de México; y otro conjunto, que llamaremos B, que es el conjunto de personas nacidas en México.

Así los participantes en el concurso serán los que pertenezcan al siguiente conjunto:

{Personas con estudios en historia de México} ∩ {Personas nacidas en México}

La definición efectiva que nos conducirá a hacer un programa de computadora, considera algunos elementos importantes. Notemos en primer lugar que los elementos de la intersección, si los hay, pertenecen a ambos conjuntos, tomaremos arbitrariamente uno de ellos como conjunto de referencia. Luego se trata de verificar del otro conjunto, solamente aquellos que también pertenezcan al conjunto de referencia, lo que nos conduce a la siguiente definición:

Definición 4.3.1 -- Intersección. Si A y B son conjuntos, la intersección de A con B se escribe $A \cap B$ y se obtiene seleccionando de A aquellos elementos que pertenezcan a B, esto es:

$$A \cap B \mapsto \{x \in A | x \in B\}$$

```
1 def intersec(A: list, B: list)-> list:
2     """ Intersección de dos conjuntos
3     Genera un conjunto con la intersección de A con B.
4     Devuelve aquellos elementos del conjunto A
5     que pertenecen también al conjunto B.
6     """
7     return subc(lambda b:en(b,A), B)
```

El efecto que produce `subc` es seleccionar del conjunto `B`, solo aquellos elemento que cumplan con el predicado $\lambda b \bullet b \in A$, que es lo que queremos.

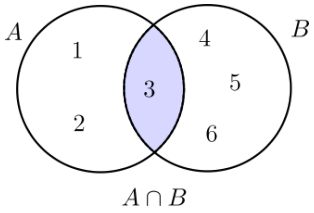


Figura 4.4: *Ejemplo intersección*

Ejemplo 4.5

Considera los conjuntos $A \leftarrow \{1,2,3\}$ y $B \leftarrow \{3,4,5,6\}$. Utilizando el programa `intersec` hallaremos $A \cap B$:

```
>>> intersec([1,2,3], [4,5,6,3])
[3]
>>>
```

Una aplicación de la intersección es para calcular $|A \cup B|$, porque se puede dar la situación de que existan algunos elementos de A que también pertenezcan a B.

La cardinalidad de $A \cup B$ no siempre es la suma de $|A| + |B|$, lo es cuando $A \cap B = \varnothing$, ahí $|A \cup B| = |A| + |B|$; pero no siempre es el caso. Cuando $A \cap B \neq \varnothing$, es decir que tienen al menos un elemento en común, hay que considerar contar los elementos de la intersección una sola vez, por lo que

$$|A \cup B| = |A| + |B| - |A \cap B|$$

Para aclarar un poco más esta igualdad, tenemos que sumar tres números:

1. Los elementos que sólo pertenecen a A
2. Los elementos que sólo pertenecen a B
3. Los elementos que sólo pertenecen a la intersección de A con B.

De modo que:

$$\begin{aligned} |A \cup B| &= (|A| - |A \cap B|) + (|B| - |A \cap B|) + |A \cap B| \\ &= |A| + |B| - |A \cap B|) - |A \cap B| + |A \cap B| \\ &= |A| + |B| - 2|A \cap B|) + |A \cap B| \\ &= |A| + |B| - |A \cap B| \end{aligned}$$

Ejemplo 4.6

En una universidad se hará un concurso de programación de computadoras. Los requisitos exigen que los participantes conozcan al menos un lenguaje de programación y que cursen actualmente al menos el sexto semestre. Se encontró que de los interesados en participar, 75 conocen al menos un lenguaje de programación; 46 cursan actualmente al menos el sexto semestre; y 18 cursan al menos el sexto semestre y conocen al menos un lenguaje de programación. ¿Cuántas personas se interesaron en participar?

Con lo anterior se conoce que:

1. $A \leftarrow \{$Conoce al menos un lenguaje de programación$\}$.
2. $B \leftarrow \{$Cursa al menos el cuarto semestre$\}$.
3. $A \cap B \leftarrow \{$Conoce al menos a un LP y cursa al menos el cuarto semestre$\}$.
4. $|A| \mapsto 75$.
5. $|B| \mapsto 46$.
6. $|A \cap B| \mapsto 18$.

Y se deduce:

$$|A \cup B| = 75 + 46 - 18 = 103$$

Como debe ser interpretado, la intersección resalta los elementos que son compartidos por ambos conjuntos. Una intersección no vacía significa que hay elementos en común. En aplicaciones como la teoría de lenguajes formales y gramáticas, es importante tener conjuntos donde la intersección sea vacía. Es tan importante que se ha creado un término especial para ese caso.

Definición 4.3.2 -- Conjunto disjunto. Si para dos conjuntos A y B se tiene que $A \cap B = \varnothing$, entonces decimos que A es disjunto con B; también se puede decir que A y B son disjuntos o completamente ajenos.

Ejemplo 4.7

Digamos que $N = \{S, A, B, C\}$ es el conjunto de símbolos no terminales de una gramática y $\Sigma = \{0, 1, 2\}$ es el conjunto de símbolos terminales de la misma gramática. Como $N \cap \Sigma \mapsto \varnothing$, entonces N y Σ son disjuntos. Esta característica es importante porque si no fueran disjuntos, habría confusión a la hora de declarar que una palabra pertenece al lenguaje, o debe seguir siendo derivada a causa de un no-terminal que es el mismo que un terminal. Aunque las gramáticas formales no es un tema de este libro, quedémonos con la importancia de saber cuándo dos conjuntos son disjuntos.

4.4 Diferencia de un conjunto respecto de otro

En ocasiones es necesario utilizar conjuntos que no tengan cierta propiedad. Esta característica origina la siguiente operación entre conjuntos que es la diferencia de un conjunto que a veces también se le llama el complemento de un conjunto respecto de otro.

Sean A y B dos conjuntos. La diferencia de A respecto a B se escribe $A \backslash B$ y es el conjunto $\{x | x \in A \wedge x \notin B\}$.

Esta definición establece que los elementos de la diferencia de A respecto a B son aquellos elementos de A que no pertenecen a B. La no pertenencia al conjunto B permite suponer que los elementos de la diferencia o bien no están explícitamente enlistados en B, o no tienen las características de los elementos en B.

Si consideramos que la diferencia del conjunto A respecto de B genera un subconjunto de A, podemos reescribir la definición en términos de un subconjunto de A:

Definición 4.4.1 -- Diferencia de conjuntos. Si A y B son conjuntos, la diferencia de A respecto de B se escribe $A\backslash B$ y se obtiene haciendo:

$$A\backslash B \mapsto \{x \in A | x \notin B\}.$$

Con diagramas de Euler apreciamos la zona sombreada de la figura 4.5, que es la región correspondiente a los elementos de A que no pertenecen a B. En algunos libros la diferencia de A respecto de B se denota $A - B$.

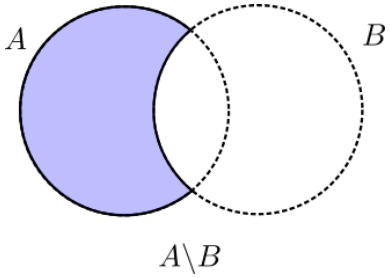


Figura 4.5: *La diferencia de A respecto a B*

Ejemplo 4.8

1. Se debe elegir un nuevo representante sindical. Se requiere a alguien con antigüedad mínima de 5 años como miembro del sindicato y que no haya tenido antes ese puesto.
   Podemos hacer $A \leftarrow$ {miembros con antigüedad mayor o igual a 5}, y $B \leftarrow$ {personas que ya han sido representantes}. Así los candidatos al puesto están en el conjunto $A\backslash B$, es decir, los que tienen al menos 5 años de antigüedad y no han sido representantes sindicales.
2. Un pasante está acordando con su director de tesis los días de la semana en los que tendrá la asesoría. El director le propuso que podía tener las asesorías de la tesis todos los días de la semana que no sean ni sábado, ni domingo ni miércoles.
   Si $A \leftarrow \{dom, lun, mar, mie, jue, vie, sab\}$ son todos los días de la semana; $B \leftarrow \{sab, dom, mie\}$ los días que el director ha solicitado que no son elegibles, entonces los posibles días de dirección de la tesis son:
   $$A\backslash B \mapsto \{lun, mar, jue, vie\},$$
   es decir, el director puede los lunes, martes, jueves y viernes.

La implementación en `Python` de la diferencia de conjuntos la podemos llamar `difc`, y debe recibir dos conjuntos [modelados como listas] y generar un nuevo conjunto [también una lista]. El procedimiento está basado en la generación de un subconjunto de un conjunto de referencia y toma los elementos del otro conjunto.

Código 4.2: *Diferencia de un conjunto respecto de otro*

```
1 def difc(A: list, B: list)-> list:
2     """ Diferencia del conjunto A respecto al conjunto B.
3     Devuelve un conjunto con aquellos elementos
4     del conjunto A que no pertenecen al conjunto B.
5     """
6     return subc(lambda a:neg(en(a,B)), A)
```

Hay situaciones en las que se requieren conjuntos de «todo aquello que no es», es decir, si A es un conjunto, nos referimos aquí al conjunto de elementos que no son

elementos de A. Frecuentemente se utiliza un conjunto de referencia para tener el complemento de un conjunto respecto de tal conjunto de referencia. Este conjunto generalmente es el conjunto universal, denotado por U.

Definición 4.4.2 -- Complemento. Si A es un conjunto, llamamos el complemento de A denotado A^c al conjunto:

$$A^c \mapsto \{x \in U | x \notin A\}$$

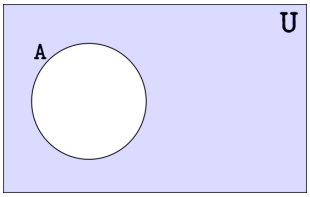
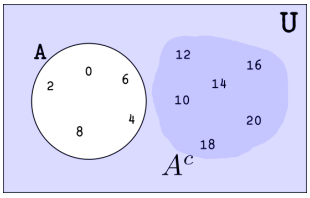


Figura 4.6: *El complemento de A considerando un conjunto universo U.*

Ejemplo 4.9

Si consideramos el universo U de los números enteros no negativos pares menores que 20, y $A \leftarrow\!\shortmid \{x \in U | x < 10\}$, el complemento de A es el conjunto $A^c \mapsto \{x \in U | x \geq 10\}$.

Primero observemos que $U \leftarrow\!\shortmid \{0, 2, 4, 6, 8, 10, 12, 14, 16, 18\}$, por lo que A^c es el conjunto $\{10, 12, 14, 16, 18\}$.



Hay algunas observaciones importantes acerca del complemento de un conjunto:

$A \cup A^c \mapsto U$**:** La unión de un conjunto A con su complemento, es el conjunto universal. Esto es porque en $A \cup A^c$ están los elementos que pertenecen a A o a A^c.

$A \cap A^c \mapsto \varnothing$**:** Es claro porque no hay elementos que pertenezcan a A y que al mismo tiempo no pertenezcan a A. Otro modo de ver esto es que no hay elementos que tengan la propiedad de los elementos de A y que al mismo tiempo no la tengan.

$U^c \mapsto \varnothing$**:** El conjunto universal contiene a todos los elementos que serán tomados en cuenta, por lo que no hay elementos que no sean parte del conjunto universo y sean tomados en cuenta.

$\varnothing^c \mapsto U$**:** Los elementos que no son parte del conjunto vacío son todos, es decir el conjunto universo.

$A^{cc} \mapsto A$**:** Este es el conjunto de elementos que no pertenecen al conjunto de elementos que no son parte de A, es decir el conjunto de elementos que no pertenecen al complemento de A, esto es aquellos elementos que pertenecen al conjunto A.

4.5 Quitar un elemento de un conjunto

Quitar un elemento de un conjunto no es una operación estándar en la teoría de conjuntos, pero computacionalmente es muy útil. Al quitar un elemento de un conjunto, se genera un nuevo conjunto de acuerdo a la siguiente definición.

Definición 4.5.1 -- Quitar un elemento de un conjunto. Sea A un conjunto y e un elemento cualquiera. La operación de quitar el elemento e del conjunto A lo denotamos $e \triangleleft A$ y es el conjunto:

$$e \triangleleft A \mapsto A \backslash \{e\}$$

Ejemplo 4.10

En un curso de fotografía, se inscribieron inicialmente Joselito, Arturo, Marcela, Josefa y Ramiro. Por problemas de salud Marcela se tuvo que dar de baja, de modo que se requiere actualizar la lista de personas que asisten al curso.

Si $F \leftarrow \{Joselito, Arturo, Marcela, Josefa, Ramiro\}$ es el conjunto de personas que inicialmente se inscribieron al curso, al darse de baja Marcela, se debe hacer la siguiente actualización.

$$\begin{aligned} F &\leftarrow Marcela \triangleleft F \\ &\leftarrow \{Joselito, Arturo, Josefa, Ramiro\}. \end{aligned}$$

La implementación en `Python` es directa una vez que se tienen las funciones `conj` para la creación de conjuntos [código 3.2, página 71] y para obtener la diferencia de un conjunto respecto de otro.

Código 4.3: *Quitar un elemento de un conjunto*

```
1 def quita(e, A:list)-> list:
2     """ Quitar un elemento de un conjunto.
3     Devuelve un conjunto con todos los elementos
4     del conjunto A, excepto el elemento e.
5     """
6     return difc(A, conj(e))
```

Ejemplo 4.11

Utilizaremos la función `quita` para reproducir el ejemplo anterior. Primero creamos el conjunto A con los nombres Joselito, Arturo, Marcela, Josefa, Ramiro, luego podemos aplicar la función `quita` para remover el elemento Marcela del conjunto A recién creado.

```
>>> A = ['Joselito', 'Arturo', 'Marcela', 'Josefa', 'Ramiro']
>>> quita('Marcela', A)
['Joselito', 'Arturo', 'Josefa', 'Ramiro']
>>>
```

4.6 La diferencia simétrica de dos conjuntos

Una operación que puede ser de mucha ayuda en ciertas ocasiones es la diferencia simétrica entre conjuntos, de la cual resulta:

Definición 4.6.1 -- Diferencia simétrica. Sean A y B dos conjuntos. La diferencia simétrica de A y B se escribe $A \Delta B$ y es el conjunto

$$A \Delta B \mapsto \{x \in A \cup B | x \in A \oplus x \in B\} \tag{4.1}$$

El diagrama de Euler de la diferencia simétrica muestra en la zona sombreada [figura 4.7]. la notación para esta operación puede ser diferente en otras obras, otra manera de escribirla es $A \oplus B$, aludiendo a la operación lógica utilizada en la definición.

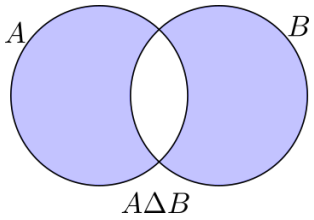


Figura 4.7: *La diferencia simétrica de conjuntos*

En todo caso, en la diferencia simétrica de conjuntos se encuentran aquellos elementos de A o B, exceptuando aquellos que tienen en común:

$$A\Delta B \mapsto \{x \in A \cup B | x \notin A \cap B\} \tag{4.2}$$

Se produce un conjunto con los elementos que pertenecen a la unión de los conjuntos A y B, exceptuando aquellos elementos que pertenecen a la intersección de A con B:

$$A\Delta B \mapsto A \cup B \backslash A \cap B \tag{4.3}$$

Pero se puede llegar al mismo resultado usando otra estrategia, que es unir la diferencia de uno respecto del otro con la diferencia del segundo respecto del primero:

$$(A \backslash B) \cup (B \backslash A). \tag{4.4}$$

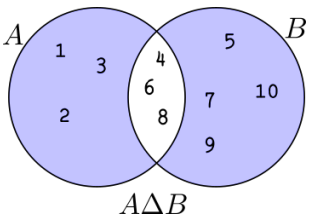
Así que hay maneras diferentes que producen resultados equivalentes para obtener la diferencia simétrica. El código siguiente se basa en la definición 4.6.1.

```
1 def difSim(A:list, B:list)-> list:
2     """ Diferencia simétrica de un conjunto respecto de otro.
3     Devuelve el subconjunto con los elementos de la unión de A con B
4     que pertenecen a la diferencia simétrica de esos conjuntos.
5     """
6     return subc(lambda x:ox(en(x,A), en(x,B)), union(A,B))
```

Ejemplo 4.12

Considera los conjuntos $A \leftarrow \{1,8,2,6,3,4\}$ y $B \leftarrow \{7,8,9,5,4,10,6\}$. Con la ayuda de la función `difSim`, calcula $A\Delta B$.

```
>>> A = [1,8,2,6,3,4]
>>> B = [7,8,9,5,4,10,6]
>>> difSim(A, B)
[10, 5, 9, 7, 1, 2, 3]
>>>
```



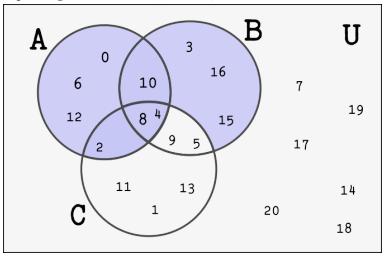
Ejercicios

1. Escribe un razonamiento deductivo para explicar las propiedades 3 y 4 de la unión de conjuntos [página 95]
2. Escribe una situación de la vida cotidiana que se pueda modelar mediante la diferencia simétrica de conjuntos. Define los conjuntos y describe las características del conjunto resultante.

Para los ejercicios 3,4 y 5 considera los conjuntos:

$U \leftarrow \{0,1,2,3,4,\ldots,20\}$
$A \leftarrow \{0,2,4,6,8,10,12\}$
$B \leftarrow \{3,4,5,8,9,10,15,16\}$
$C \leftarrow \{8,4,2,9,13,11,5,1\}$

3. Calcula:
   a) *Ejemplo:* $(A \cup B) \cap C \mapsto \{8,4,2,9,5\}$.
   b) $(C \cap A) \cup (C \cap B)$
   c) $B \cap ((A \cup B) \backslash C)$
   d) $(C \backslash A) \Delta (A \backslash B)$
4. Sean A, B y C los del ejercicio anterior y sea $U \leftarrow \{0,1,2,3,4,\ldots,20\}$. Calcula:
   a) *Ejemplo:* $A^c \mapsto \{1,3,5,7,9,11,13,14,15,1617,18,19,20\}$.
   b) $(A \cup C)^c$
   c) $(B \cap C)^c$
   d) $B^c \Delta (A \cup C)^c$
5. Dibuja diagramas de Venn para los siguientes casos:
   a) *Ejemplo:* $A \cup (B \backslash C) \mapsto \{16,15,3,0,2,4,6,8,10,12\}$



   b) $B^c \cap (A \backslash C)$
   c) $(C \Delta B) \cap A$
   d) $(A \cup B)^c$
   e) $A^c \cap B^c$
6. Calcula el valor final del conjunto G después de las siguientes operaciones.
   a) $G \leftarrow \varnothing$
   b) $G \leftarrow x \triangleright G$
   c) $G \leftarrow f \triangleright G$
   d) $G \leftarrow z \triangleright G$
   e) $G \leftarrow x \triangleleft G$
   f) $G \leftarrow a \triangleright G$
   g) $H \leftarrow \{f,g,u,j\}$
   h) $G \leftarrow H \backslash G$
7. En un taller de mecánico hay tres especialistas que atienden a los clientes que llegan durante el día. En ocasiones estos clientes solicitan servicios muy grandes o complicados, de modo que requieren la atención no solo de un especialista, sino a veces dos o incluso los tres. Al final del mes, ellos quieren saber qué clientes

fueron atendidos por un solo especialista. Debes hacer un programa en `Python` que resuelva este problema.
Tu programa debe recibir tres líneas de texto introducido por el teclado. Cada línea es una lista de números enteros separados por un espacio. Cada número representa el número del cliente que fue atendido. En las tres listas de números pueden haber números repetidos, esto indica que ese cliente solicitó un trabajo que fue resuelto por más de un especialista. El programa debe devolver una lista de números que contiene los números que representan a los clientes que fueron atendidos por un solo especialista.
Considera que los números de entrada son enteros positivos menores que 100. Observa el ejemplo.

Entrada	Salida	Explicación

`1 2 15 5 6 7 3 4`
`7 4 3 8 9 11 10`
`10 11 12 13 14 5 6 7`
	`1 2 8 9 12 13 14 15`	Estos números representan a los clientes que fueron atendidos por un único especialista. `7` no está porque fue atendido por más de un especialista.

Familias de Conjuntos 5

En este capítulo estudiaremos algunos conjuntos especiales que incluyen otros conjuntos. Los conjuntos que estudiaremos aquí tienen utilidad en futuros temas y en otras áreas de estudio, como la Teoría de Lenguajes Formales y Autómatas, Estructuras Algebraicas, Topología; también se han aplicado en Gestión de Proyectos, particularmente en la gestión de equipos de trabajo, entre otras aplicaciones.

5.1 El conjunto potencia

Si A es un conjunto, el conjunto potencia de A se denota $\mathcal{P}(A)$ y es el conjunto de todos los subconjuntos de A. Otras notaciones pueden ser $\mathbb{P}(A)$, $\mathbf{P}(A)$, $\mathscr{P}(A)$ o incluso 2^A.

Ejemplo 5.1

Si $A \leftarrow \{1,2,3\}$, el conjunto potencia $\mathcal{P}(A) \mapsto \{\varnothing, \{1\}, \{2\}, \{2,1\}, \{3\}, \{3,1\}, \{3,2\}, \{3,2,1\}\}$

Además de conocer la definición, es muy práctico saber cómo obtener el conjunto potencia de un conjunto dado. En los siguientes párrafos delinearemos un procedimiento recursivo.

Para construir el procedimiento recursivo-iterativo [función recursiva que genera procesos iterativos] empezaremos considerando el conjunto A en el caso más simple, cuando $A = \varnothing$.

Si $A = \varnothing$, entonces $\mathcal{P}(A)$ tiene solamente un elemento $\mathcal{P}(A) \mapsto \{\varnothing\}$, ya que el conjunto vacío es el único subconjunto de $\varnothing$. El siguiente caso es cuando A tiene un único elemento, el elemento de A puede ser cualquiera, digamos arbitrariamente que $A \leftarrow \{3\}$, entonces $\mathcal{P}(A) \mapsto \{\varnothing, \{3\}\}$, la siguiente tabla muestra los primeros cuatro casos.

A	$\mathcal{P}(A)$	$\lvert A\rvert$	$\lvert\mathcal{P}(A)\rvert$
$\{\}$	$\{\{\}\}$	0	1
$\{1\}$	$\{\{1\},\{\}\}$	1	2
$\{1,2\}$	$\{\{1,2\},\{2\},\{1\},\{\}\}$	2	4
$\{1,2,3\}$	$\{\{1,2,3\},\{2,3\},\{1,3\},\{3\},\{1,2\},\{2\},\{1\},\{\}\}$	3	8
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Observa que:

1. Con respecto a los elementos.
   a) Si $A = \varnothing$, $\mathcal{P}(A) \mapsto \{\varnothing\}$, este es un caso base.
   b) Si $A \neq \varnothing$, A es de la forma $\{a_0|A'\}$. La mitad de los elementos del conjunto potencia tiene incluye a a_0; la mitad que no lo tiene, es justamente $\mathcal{P}(A')$. Observa en la tabla cuando $A \leftarrow\!\shortmid \{1,2,3\}$, aquí $a_0 \leftarrow\!\shortmid 1$ y $A' \leftarrow\!\shortmid \{2,3\}$. La mitad de elementos de $\mathcal{P}(\{1,2,3\})$ contiene a 1, y la otra mitad es $\mathcal{P}(\{2,3\})$.
2. Con respecto a la cardinalidad del conjunto potencia. Observamos que cada vez que la cardinalidad del conjunto base aumenta una unidad, la cardinalidad de su conjunto potencia aumenta una potencia de 2.

5.1.1 Algoritmo para el conjunto potencia

Usaremos esta observación para crear un algoritmo recursivo para generar el conjunto potencia de un conjunto. En el siguiente algoritmo se tiene como segundo parámetro un conjunto R que es opcional, en caso de no escribirlo $R \leftarrow\!\shortmid \{\varnothing\}$.

Definición 5.1.1 -- Conjunto Potencia. El conjunto potencia de un conjunto A, es el conjunto que se obtiene mediante:

$$\mathcal{P}(A, R \leftarrow\!\shortmid \{\varnothing\}) \mapsto \begin{cases} R & \text{si } A = \varnothing. \\ \\ \mathcal{P}(A', \texttt{enCada}(\lambda X \bullet a_0 \triangleright X, R) \cup R) & eoc \end{cases} \tag{5.1}$$

En la definición 5.1.1 se menciona la función `enCada`, la cual podemos definir como $\texttt{enCada}(f, \langle a_1, \ldots, a_n \rangle) \mapsto \langle f(a_1), \ldots, f(a_n) \rangle$. En este caso se aplica la función $\triangleright$, llamada `agrega` [definición 4.1.1, página 91], agregando x_0 en cada conjunto X de R; lo que genera una lista con los resultados obtenidos en el orden en que fue aplicada.

Código 5.1: *Aplica una función en cada elemento de un dominio*

```
1 def enCada(P:callable, D:list)-> list:
2     """
3     Aplica P en cada elemento de D.
4     """
5     return list(map(P, D))
```

Ejemplo 5.2

Aplica la función incremento en uno, en cada elemento del dominio $\{1, 5, 8\}$. La función incremento en uno se puede definir como $\texttt{inc1} \leftarrow\!\shortmid \lambda x \bullet x + 1$, por lo que podemos hacer:

```
>>> inc1 = lambda x:x+1
>>> enCada(inc1, conj(1, 5, 8))
[2, 6, 9]
>>>
```

De nuevo con el algoritmo de $\mathcal{P}(A)$. Cuando $A \neq \varnothing$, se toma R que contiene los conjuntos generados hasta el momento y en cada uno de ellos se le agrega el elemento a_0 que corresponde. Esto produce la mitad de la respuesta; la otra mitad es R.

Ejemplo 5.3
Calcular $\mathcal{P}(\{x,y,z\})$.

$$
\begin{array}{lcll}
\mathcal{P}(\{x,y,z\}) & \mapsto & \mathcal{P}(\{y,z\},\{\{x\}\}\cup\{\}) & \text{Comentario:} \\
 & & = \mathcal{P}(\{y,z\},\{\{x\},\{\}\}) & R \leftarrow \{\{\}\} \\
 & \mapsto & \mathcal{P}(\{z\},\{\{x,y\},\{y\}\}\cup\{\{x\},\{\}\}) & A' \leftarrow \{y,z\}; R \leftarrow \{\{x\},\{\}\} \\
 & & = \mathcal{P}(\{z\},\{\{x,y\},\{y\},\{x\},\{\}\}) & A' \leftarrow \{z\}; R \leftarrow \{\{x,y\},\{y\},\{x\},\{\}\} \\
 & \mapsto & \mathcal{P}(\{\},\{\{x,y,z\},\{y,z\},\{x,z\},\{z\}\}\cup\{\{x,y\},\{y\},\{x\},\{\}\}) & \\
 & & = \mathcal{P}(\{\},\{\{x,y,z\},\{y,z\},\{x,z\},\{z\},\{x,y\},\{y\},\{x\},\{\}\}) & \text{El caso base.} \\
 & \mapsto & \{\{x,y,z\},\{y,z\},\{x,z\},\{z\},\{x,y\},\{y\},\{x\},\{\}\} &
\end{array}
$$

Código 5.2: *El conjunto potencia de un conjunto*

```
1  def cPot(A:list, R:list=conj(vacio))-> list:
2      """ Conjunto potencia de un conjunto
3      Calcula el conjunto potencia de un conjunto
4      A en un conjunto.
5      R guarda el resultado. Es un conjunto de conjuntos.
6      """
7      if esVacio(A):
8          return R
9      else:
10         return cPot(cdr(A), enCada(lambda X:agrega(car(A), X), R) + R)
```

Ejemplo 5.4
Utiliza la función `cPot` para calcular el conjunto potencia de $A \leftarrow \{x,y,z\}$.

```
>>> A = conj('x', 'y', 'z')
>>> cPot(A)
[['x', 'y', 'z'], ['y', 'z'], ['x', 'z'], ['z'], ['x', 'y'], ['y'], ['x'], []]
>>>
```

Ejemplo 5.5
Una empresa que se dedica a la comunicación gráfica, tiene un departamento de edición que consta de 10 diseñadores gráficos, la empresa quiere trabajar en un nuevo proyecto para el que requiere formar un equipo de trabajo. En el momento, la empresa puede disponer de cuatro diseñadores, estos son: Ana, Gustavo, Peter y Giovana. ¿Cuántos equipos de trabajo puede formar la empresa con sus diseñadores disponibles? y más importante, ¿cuáles son esos equipos?

El conjunto de diseñadores puede ser modelado por el conjunto D que debe contener los nombres de los diseñadores, esto es

$$D \leftarrow \{\texttt{ana}, \texttt{gustavo}, \texttt{peter}, \texttt{giovana}\}.$$

Cada equipo de trabajo puede ser visto como un subconjunto no vacío de los elementos en D, por lo que tendrá

$$2^{|D|} - 1 = 2^4 - 1 = 16 - 1 \mapsto 15 \quad \text{Equipos de trabajo con diseñadores.}$$

Como el equipo sin diseñadores no es útil para realizar la tarea, podemos retirarlo de la solución, de modo que todos los equipos que se pueden formar, sin considerar el conjunto vacío,

es el conjunto de subconjuntos del conjunto potencia, excepto el conjunto que tiene al conjunto vacío.

$$\mathcal{P}(D)\backslash\{\varnothing\}$$

Apoyados con la computadora, podemos obtener el listado de todos ellos.

```
>>> difc(cPot(conj('ana', 'gustavo', 'peter', 'giovana')), [[]])
[['ana', 'gustavo', 'peter', 'giovana'],
 ['gustavo', 'peter', 'giovana'],
 ['ana', 'peter', 'giovana'],
 ['peter', 'giovana'],
 ['ana', 'gustavo', 'giovana'],
 ['gustavo', 'giovana'],
 ['ana', 'giovana'], ['giovana'],
 ['ana', 'gustavo', 'peter'],
 ['gustavo', 'peter'],
 ['ana', 'peter'],
 ['peter'],
 ['ana', 'gustavo'],
 ['gustavo'],
 ['ana']]
>>>
```

5.1.2 Cálculo de la cardinalidad del conjunto potencia

La segunda observación del comportamiento en el desarrollo del conjunto potencia [página 105], es la que tiene que ver con la cardinalidad.

Se observa que el caso base es cuando $A \hookleftarrow \varnothing$, lo que genera $\mathcal{P}(A) \mapsto \{\varnothing\}$, que obviamente $|\mathcal{P}(A)| \mapsto 1$. De modo que, cuando $|A| = 0$, se tiene $|\mathcal{P}(A) = 1|$.

En los casos siguientes, el algoritmo toma el conjunto potencia anterior y prácticamente hace una copia, a la cual se le hace cierta modificación. Pero el hecho de hacer una copia, indica que cada conjunto del resultado previo aparece dos veces, sin embargo a una de esas copias se le agrega un nuevo elemento y la otra permanece sin cambio, de esto resulta que en cada paso el número de subconjuntos aumenta el doble, como se muestra en la figura 5.1.

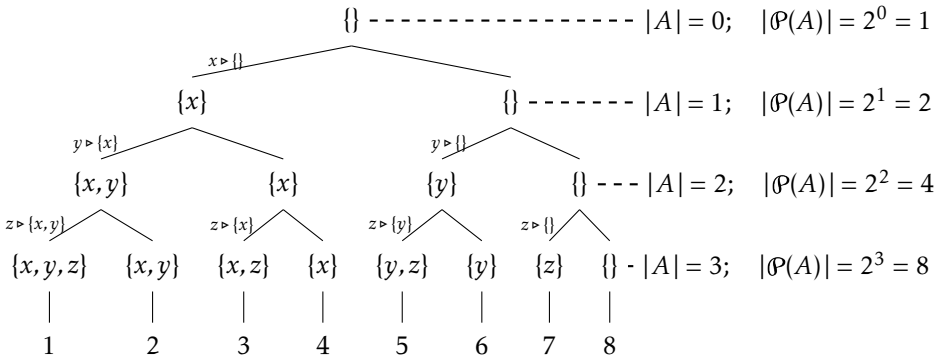


Figura 5.1: *Cardinalidad del conjunto potencia de un conjunto.*

Así la cardinalidad del conjunto potencia de un conjunto de $n \geq 0$ elementos es precisamente 2^n [Eri09, p. 3].

5.2 Familias de conjuntos

Definición 5.2.1 -- Familia de conjuntos. Una familia $\mathscr{F}$ de conjuntos, es un conjunto cuyos elementos son conjuntos.

Siendo A un conjunto, el $\mathcal{P}(A)$ contiene todas las posibles familias de conjuntos que tienen elementos en A, por lo que si $\mathscr{F} \subseteq \mathcal{P}(A)$, los miembros de $\mathscr{F}$ son subconjuntos de A.

Una familia de conjuntos $\mathscr{F}$ se puede denotar siguiendo las mismas reglas que cualquier otro conjunto definido del mismo modo [sección 3.1.1, página 70]. La notación más frecuente es $\mathscr{F} \leftarrow \{S \subseteq D \mid P(S)\}$, es decir, el conjunto de subconjuntos S de un dominio D que cumplen un predicado P.

Al observar la notación para conjuntos $\{x \in D \mid P(x)\}$, las dos diferencias importantes en la notación para familias de conjuntos son:

1. La variable S es un subconjunto de un dominio D. Por lo que los elementos de $\mathscr{F}$ serán conjuntos.
2. $P(S)$ es un predicado que toma como argumento un conjunto.

Esencialmente las familias de conjuntos obedecen todas las reglas de definición de los conjuntos [sección 3.2, página 71]. Así una particular familia de conjuntos puede definirse de dos maneras:

1. Enlistando uno a uno los conjuntos que pertenecen a la familia. Esta manera es la forma **extensional** de definir la familia.
2. Describiendo un predicado que al ser evaluado con los miembros de la familia, el predicado es `True`. Esta manera es la forma **intencional** de definir la familia. En este caso en particular se le conoce como **clase de conjuntos**.

Ejemplo 5.6

Los siguientes son diferentes maneras de definir familias de conjuntos:

1. El conjunto $\mathscr{F} = \{\{1,2,3\},\{a,b,c\},\{f,g\}\}$ es una familia de conjuntos con 3 integrantes, los cuales son explícitamente el conjunto $\{1,2,3\}$, el conjunto $\{a,b,c\}$ y el conjunto $\{f,g\}$.
2. El conjunto $\mathscr{F} = \{X \subseteq \{1,2,3,4\} \mid \{2,4\} \cap X \neq \varnothing\}$ es también una familia de conjuntos. Literalmente se puede expresar como «el conjunto de subconjuntos de $\{1,2,3,4\}$ que al menos tienen el 2 o el 4».

En particular, $\{\varnothing\}$ es la familia que tiene como único miembro al conjunto vacío, y es una familia de conjuntos obtenida de cualquier conjunto.

5.2.1 Unión generalizada

La unión de los miembros de una familia de conjuntos se puede diferenciar de la unión de dos conjuntos porque el concepto se aplica directamente a todos los conjuntos miembros de la familia, mientras que la unión está definida para exactamente dos conjuntos. La unión extendida sobre una familia de conjuntos $\mathscr{F}$ se denota con el símbolo:

$$\bigcup \mathscr{F}$$

La aplicación de la unión generalizada tiene que hacerse son los miembros de la familia. Si $F \leftarrow \{A_1, A_2, A_3, \ldots, A_n\}$ por lo que para denotar la unión de todos los miembros de la familia de conjuntos $\mathscr{F}$ escribimos:

$$\bigcup \{A_1, A_2, A_3, \ldots, A_n\}.$$

En este caso podemos utilizar súper índices y subíndices para denotar una aplicación generalizada de la unión [Hal74, p. 34].

$$\bigcup_{i \leftarrow 1}^{n} A_i.$$

También es posible indicar que la unión se realiza sobre los subconjuntos de una familia, indicando que una variable tomará el valor de cada miembro de la familia para servir como operando. Frecuentemente se utiliza esta notación al trabajar con los elementos de una familia de conjuntos.

$$\bigcup_{A \in \mathscr{F}} A.$$

El procedimiento para calcular la unión generalizada es equivalente a realizar la unión convencional con el primer par de conjuntos, el conjunto resultante será ahora el primer conjunto de la colección, luego se unirá este conjunto con el que ahora es el primero del resto, continuando así de manera recursiva hasta que solamente queden dos conjuntos:

$$\bigcup \{A_1, A_2, A_3, \ldots, A_n\} = (\ldots((A_1 \cup A_2) \cup A_3)\ldots) \cup A_n)$$

Ahora, si $\mathscr{F}$ es un conjunto no vacío, se puede escribir $\{A_0|F'\}$, donde A_0 es un miembro de la familia, cualquier miembro puede servir y $\mathscr{F}'$ es el resto de los subconjuntos de la familia, excepto el que se ha elegido como primero. Con lo que la notación de la unión generalizada se escribe:

$$\bigcup \mathscr{F} = \bigcup \{A_0|F'\}.$$

El procedimiento para llevar al cabo la unión generalizada se puede describir mediante un algoritmo recursivo como sigue:

$$\bigcup \mathscr{F} \mapsto \begin{cases} \varnothing & \text{si } \mathscr{F} = \varnothing. \\ A_0 & \text{si } \mathscr{F} = \{A_0|\mathscr{F}'\} \rightarrow \mathscr{F}' = \varnothing. \\ \bigcup (A_0 \cup B_0) \triangleright \mathscr{B} & \text{si } \mathscr{F}' = \{B_0|\mathscr{B}\}. \end{cases}$$

La idea general es:

1. Si la familia no tiene miembros, entonces se produce el conjunto vacío.
2. Si la familia tiene unicamente un miembro es decir, el resto de los miembros $\mathscr{F}'$ es vacío, entonces simplemente se devuelve ese único miembro de la familia, que es precisamente A_0.
3. Finalmente el otro caso, cuando el resto de los conjuntos $\mathscr{F}'$ no es vacío, lo que aquí se ha denotado como $\{B_0|\mathscr{B}\}$; entonces se hace la unión de los primeros dos conjuntos, luego se agrega el conjunto resultante a la nueva familia, conformada por la unión de los dos primeros, con los demás.

Ejemplo 5.7

Si $X_1 = \{1,2,3,4,5\}$, $X_2 = \{3,5,7,9\}$, $X_3 = \{2,4,6,8,10,12\}$ y $X_4 = \{1,8\}$; donde $\mathscr{F} = \{X_1, X_2, X_3, X_4\}$ es una familia de conjuntos, el cálculo de la unión generalizada sobre los conjuntos de la familia, siguiendo el algoritmo recursivo es como sigue:

		Comentario:
$\bigcup \mathscr{F}$	$= \bigcup \{X_1 \mid \mathscr{F}'\}$	
	$= \bigcup (X_1 \cup X_2) \triangleright \mathscr{B}$	# $B_0 \mapsto X_2; \mathscr{B} \mapsto \{X_3, X_4\}$
	$= \bigcup (\{1,2,3,4,5\} \cup \{3,5,7,9\}) \triangleright \mathscr{B}$	
	$= \bigcup \{1,2,3,4,5,7,9\} \triangleright \mathscr{B}$	# $\mathscr{B} \mapsto \{\{1,2,3,4,5,7,9\}, X_3, X_4\}$
	$= \bigcup (\{1,2,3,4,5,7,9\} \cup X_3) \triangleright \mathscr{B}$	# $B_0 \mapsto X_3; \mathscr{B} \mapsto \{X_4\}$
	$= \bigcup (\{1,2,3,4,5,7,9\} \cup \{2,4,6,8,10,12\}) \triangleright \mathscr{B}$	
	$= \bigcup \{1,2,3,4,5,7,9,6,8,10,12\} \triangleright \mathscr{B}$	# $\mathscr{B} \mapsto \{\{1,2,3,4,5,7,9,6,8,10,12\}, X_4\}$
	$= \bigcup \{1,2,3,4,5,7,9,6,8,10,12\} \cup X_4 \triangleright \mathscr{B}$	# $B_0 \mapsto X_4; \mathscr{B} \mapsto \varnothing$
	$= \bigcup \{1,2,3,4,5,7,9,6,8,10,12\} \cup \{1,8\} \triangleright \mathscr{B}$	
	$= \bigcup \{1,2,3,4,5,7,9,6,8,10,12\} \triangleright \mathscr{B}$	# $\mathscr{F} \mapsto \{\{1,2,3,4,5,7,9,6,8,10,12\} \mid \varnothing\}$
	$= \{1,2,3,4,5,7,9,6,8,10,12\}.$	

El procedimiento recursivo genera un programa como el siguiente:

Código 5.3: *Unión generalizada*

```
def Union(*FamC)-> list:
    """ Unión generalizada.
    Recibe una lista no determinada de conjuntos en forma de listas
    Devuelve una lista que es interpretada como la unión
    de todos los conjuntos dados.
    """
    if esVacio(FamC):
        return vacio
    elif esVacio(cdr(FamC)):
        return car(FamC)
    else:
        nuevoC = union(car(FamC), car(cdr(FamC)))
        nuevaF = agrega(nuevoC, FamC[2:])
        return Union(*nuevaF)
```

Ejemplo 5.8

El mismo ejemplo anterior, pero ahora resuelto con el programa `Union`.

```
>>> X1 = conj(1,2,3,4,5)
>>> X2 = conj(3,5,7,9)
>>> X3 = conj(2,4,6,8,10,12)
>>> X4 = conj(1,8)
>>> Union(X1, X2, X3, X4)
[12, 10, 8, 6, 9, 7, 1, 2, 3, 4, 5]
>>>
```

Los parámetros de tipo `*args` sirven para pasar múltiples argumentos a una función [Hun20, p. 128]. La lista de argumentos son colocados en una tupla, no una lista. Al definir argumentos tipo `*args`, estos deben ser colocados después de todos los argumentos obligatorios, de otro modo se generará un error. Por ejemplo en la función `foo` se define un argumento obligatorio y lo siguiente será considerado como una lista no determinada de argumentos.

```
def foo(a,*b):
    print(a)
    print(b)
```

```
>>> foo(3,5,8,9)
3
(5, 8, 9)
>>>
```

5.2.2 Intersección generalizada

La intersección de los miembros de una familia de conjuntos es una generalización de la intersección de dos conjuntos. La notación y el cálculo, en lo general, se parece mucho a la unión generalizada.

$$\bigcap \mathscr{F}$$

La notación puede cambiar con las mismas variantes que la unión generalizada, por ejemplo si $\mathscr{F} \mapsto \{A_1, \ldots, A_n\}$ se puede escribir

$$\bigcap_{A \in \mathscr{F}} A, \quad \text{o bien} \quad \bigcap_{i \hookleftarrow 1}^{n} A$$

Cuando la familia de conjuntos no es vacía, se puede hacer referencia a un primer miembro de la familia y a un conjunto que contenga al resto de los miembros de la familia, en una notación $\{A_0 | \mathscr{F}'\}$. Esta notación es perfecta para realizar un algoritmo recursivo que calcule la intersección sobre los miembros de la familia $\mathscr{F}$:

$$\bigcap \mathscr{F} \mapsto \begin{cases} \varnothing & \text{si } \mathscr{F} = \varnothing \\ \varnothing & \text{si } \mathscr{F} = \{A_0 | \mathscr{F}'\} \rightarrow A_0 = \varnothing \\ A_0 & \text{si } \mathscr{F} = \{A_0 | \mathscr{F}'\} \rightarrow \mathscr{F}' = \varnothing \\ \bigcap (A_0 \cap B_0) \triangleright \mathscr{B} & \text{si } \mathscr{F}' = \{B_0 | \mathscr{B}\} \end{cases}$$

Este procedimiento recursivo considera tres diferentes casos base:

1. Si no hay miembros en la familia, el resultado es el conjunto vacío.
2. Si el primer conjunto es el conjunto vacío, la intersección generalizada es también el conjunto vacío.
3. Si solamente hay un miembro en la familia, el resultado es ese único subconjunto.

El único caso recursivo se considera cuando hay más de un conjunto en la familia. En este caso el procedimiento exige tomar los primeros dos conjuntos, hacer la intersección con ellos, y el resultado de eso agregarlo al resto de los conjuntos. Una vez agregado, se tendrá una nueva familia de conjuntos pero de cardinalidad inferior, con la que se hará recursivamente la intersección sobre sus elementos.

Ejemplo 5.9
Calcula la intersección de los conjuntos en la familia $\mathscr{F} \leftarrow \{X_1, X_2, X_3, X_4\}$, donde $X_1 = \{1,2,3,4,5\}$, $X_2 = \{1,3,5,7,9\}$, $X_3 = \{1,2,4,6,8,10,12\}$ y $X_4 = \{1,8\}$.

$$
\begin{aligned}
\bigcap \mathscr{F} &= \bigcap \{X_1 | \mathscr{F}'\} && \text{Comentario: } \mathscr{F}' \mapsto \{X_2 | \mathscr{B}\} \\
&= \bigcap (X_1 \cap X_2) \triangleright \mathscr{B} \\
&= \bigcap (\{1,2,3,4,5\} \cap \{1,3,5,7,9\}) \triangleright \mathscr{B} \\
&= \bigcap \{1,3,5\} \triangleright \mathscr{B} && \{\{1,3,5\}, X_3, X_4\} \\
&= \bigcap (\{1,3,5\} \cap X_3) \triangleright \mathscr{B} \\
&= \bigcap (\{1,3,5\} \cap \{1,2,4,6,8,10,12\}) \triangleright \mathscr{B} \\
&= \bigcap \{1\} \triangleright \mathscr{B} && \{\{1\}, X_4\} \\
&= \bigcap \{1\} \cap \{1,8\} \triangleright \mathscr{B} \\
&= \bigcap \{1\} \triangleright \mathscr{B} && \{\{1\}\} \\
&= \{1\}
\end{aligned}
$$

5.3 Cubrimientos

Consideremos ahora un conjunto A y $\mathscr{F} \in \mathcal{P}(U)$ una familia de subconjuntos de U, con la que podemos establecer una situación muy útil que se presenta con las familias de conjuntos.

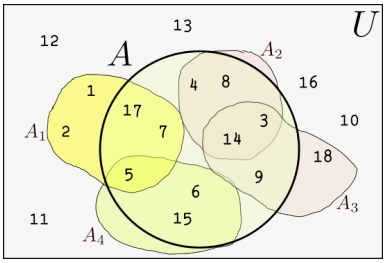
Definición 5.3.1 -- Cubrimiento. Digamos que $\mathscr{F} \in \mathcal{P}(U)$ es una familia no vacía de subconjuntos de un universo U, y A un conjunto. Decimos que $\mathscr{F}$ es un **cubrimiento** de A si se cumple que

$$A \subseteq \bigcup \mathscr{F}$$

La definición 5.3.1 requiere en principio un conjunto A que es un subconjunto de algún universo U, aunque es posible que $A = U$. En este caso se requiere que

$$A = \bigcup \mathscr{F}$$

Ejemplo 5.10
Considera el conjunto universo $U \leftarrow \{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18\}$, un conjunto $A \leftarrow \{17,7,5,6,15,14,3,9,8,4\}$ del cual se tiene una familia $\mathscr{F} \leftarrow \{A_1, A_2, A_3, A_4\}$ donde $A_1 \leftarrow \{1,2,5,7,17\}$, $A_2 \leftarrow \{14,3,8,4\}$, $A_3 \leftarrow \{18,9,3,14\}$ y $A_4 \leftarrow \{15,6,5\}$. La situación se muestra en el siguiente diagrama de Euler.



Observamos que

$$\begin{aligned}\bigcup_{i=1}^{4} A_i &= A_1 \cup A_2 \cup A_3 \cup A_4 \\ &= \{1,2,5,7,17\} \cup \{14,3,8,4\} \cup \{18,9,3,14\} \cup \{15,6,5\} \\ &\mapsto \{1,2,5,7,17,8,4,14,3,18,9,15,6\}\end{aligned}$$

Luego,

$$A \subseteq \bigcup_{i=1}^{4} A_i = \{17,7,5,6,15,14,3,9,8,4\} \subseteq \{1,2,5,7,17,8,4,14,3,18,9,15,6\}$$

$$\mapsto \texttt{True}.$$

Por lo que $\mathscr{F}$ es un cubrimiento de A.

Una función en `Python`, que permite saber si una familia `F` de conjuntos es un cubrimiento de un conjunto `A`, utiliza la función `Union` [código 5.3, página 111] se puede escribir como sigue:

Código 5.4: *Cubrimiento*

```
1  def esCub(F, A):
2      """ Verifica un cubrimiento
3      determina si la familia de conjuntos F
4      es un cubrimiento para el conjunto A
5      devuelve un valor booleano.
6      """
7      return esSubc(A, Union(*F))
```

Ejemplo 5.11

Utiliza la función `esCub` para determinar si la familia de conjuntos $\mathscr{F} \leftarrow \{A_1, A_2, A_3 A_4\}$ es un cubrimiento del conjunto $A \leftarrow \{1,2,3,4,5,6,7,8,9,10\}$. Donde $A_1 \leftarrow \{1,9,2\}$, $A_2 \leftarrow \{2,4,6,8\}$, $A_3 \leftarrow \{1,3,5,7\}$ y $A_4 \leftarrow \{1,4,9,10\}$.

```
>>> A = conj(1,2,3,4,5,6,7,8,10)
>>> A1 = conj(1,9,2)
>>> A2 = conj(2,4,6,8)
>>> A3 = conj(1,3,5,7)
>>> A4 = conj(1,4,9,10)
>>> F = conj(A1, A2, A3, A4)
>>> esCub(F,A)
True
>>>
```

Ejemplo 5.12

Se tiene la siguiente base de datos llamada BD:

Nombre	N^o Id	Puntos Acumulados	Basificado
Jaqueline Alma Benítez	34	93749	False
Santiago Pedro Colunga	58	62465	False
Jesús Aguirre	42	13212	True
Bruno Corona	41	72335	True
Oscar Valdez	68	57368	True
Úrsula Graciela Jiménez	14	81237	False
Sessa Riojas	20	90961	False
Antonia Socorro	45	03083	True
Graciela Vela	76	86548	True
Delia García	36	70076	True
Olivia Bianca Caballero	29	46194	False
Noemí Valentín	77	09027	True

Se pretende hacer una reunión con todas estas personas, pero las deben agrupar bajo los siguientes criterios:

1. Grupo A: Las personas con número de Id menor que 30 o no son basificados.
2. Grupo B: Las personas basificadas con un número de puntos acumulados mayor de 50000.
3. Grupo C: Las personas que si no son basificadas, tienen número de puntos menores a 70000.

Los organizadores se preguntan si todas las personas van a ser convocadas.

Formalmente esta situación se puede resolver al determinar si los grupos A, B y C forman un cubrimiento de la base de datos.

Hagamos entonces:

$A \leftarrow \{p \in \texttt{BD} \mid \texttt{nId}(p) < 50 \vee \neg(\texttt{basif}(p))\}$

$B \leftarrow \{p \in \texttt{BD} \mid \texttt{pAcum}(p) > 50000 \wedge \texttt{basif}(p)\}$

$C \leftarrow \{p \in \texttt{BD} \mid \neg\texttt{basif}(p) \rightarrow \texttt{pAcum}(p) < 70000\}$

Después de hacer los cálculos, se tiene que los grupos están conformados por las siguientes personas (solo se muestra su N^o Id):

$A \leftarrow \{34, 58, 14, 20, 29\}$

$B \leftarrow \{41, 68, 76, 36\}$

$C \leftarrow \{58, 42, 41, 68, 45, 76, 36, 29, 77\}$

Dado que $(A \cup B \cup C) \subseteq \texttt{BD}$, la familia de conjuntos $\{A, B, C\}$ es un cubrimiento de BD, por lo que en efecto, todas las personas serán convocadas.

5.4 Particiones

Ya con la definición de cubrimiento, es posible crear familias de subconjuntos con alguna característica importante, por ejemplo que todos los subconjuntos contengan a un elemento en particular, o cualquier otra condición que sea útil para fines prácticos. En esta sección estudiaremos una condición que caracteriza los cubrimientos como «particiones».

Definición 5.4.1 -- Partición. Sea $\mathscr{F}$ una familia no vacía de subconjuntos de un conjunto A. Decimos que $\mathscr{F}$ es una **partición** de A si se cumple:

1. $\mathscr{F}$ es un cubrimiento de A
2. $\forall X \in \mathscr{F} : X \neq \varnothing$
3. $\forall X \in \mathscr{F} : [\forall Y \in \mathscr{F} : X \neq Y \leftrightarrow X \cap Y = \varnothing]$

Además de que $\mathscr{F}$ debe ser un cubrimiento de A, ahora se exigen un par de nuevas condiciones, que ningún subconjunto en $\mathscr{F}$ sea vacío, ya que de otro modo la siguiente

condición carece de sentido y la otra condición es la clave de la definición, ya que se requiere que ningún par de conjuntos de la familia, si estos son diferentes, compartan elementos.

Ejemplo 5.13

Considera el conjunto $A \leftarrow \{1,2,\ldots,20\}$ y la familia $\mathscr{F} \leftarrow \{A_1, A_2\}$ conformada por los conjuntos $A_1 \leftarrow \{n \in A \mid \texttt{esPar}(n)\}$ y $A_2 \leftarrow \{n \in A \mid \neg\texttt{esPar}(n)\}$.

La familia $\mathscr{F}$ es una partición de A porque $A_1 \cup A_2 = A$ y $A_1 \cap A_2 = \varnothing$.

Si $\mathscr{F} \leftarrow \{A_1,\ldots,A_n\}$ es una partición de un conjunto A, cada subconjunto A_i con $i = 1,\ldots,n$, es un **bloque** de la partición. En una partición de A, cada elemento de A pertenece a un único bloque.

Ejemplo 5.14

Considera nuevamente la base de datos del ejemplo 5.12. Ahora los organizadores están interesados en que cada persona participe en un solo grupo, pero también quieren que todos participen; por lo que proponen los siguientes grupos:

Grupo A: $\{p \in \texttt{BD} \mid \texttt{pAcum}(p) > 50000 \leftrightarrow \texttt{basif}(x)\}$
Grupo B: $\{p \in \texttt{BD} \mid \texttt{pAcum}(p) > 50000 \wedge \neg\texttt{basif}(p)\}$
Grupo C: $\{p \in \texttt{BD} \mid \texttt{pAcum}(p) < 15000 \wedge \texttt{basif}(p)\}$

Código 5.5: *Partición de un conjunto*

```
def esPart(FamC:list, A:list)-> bool:
    """
    Determina si la familia de conjuntos FamC
    es una partición del conjunto A.
    """
    return y(esCub(FamC, A),
             paraTodo(lambda X:
                      paraTodo(lambda Y:
                               ssi(neg(cIguales(X,Y)),
                                   esVacio(inters(X,Y))),
                               FamC),
                          FamC))
```

Ejemplo 5.15

Dado $U \leftarrow \{15,23,78,49,18,33,19,82\}$, utiliza la función `esPart` para determinar cuál familia de conjuntos es una partición de U.

1. $\mathscr{F}_1 \leftarrow \Big\{\{15,49,18,82\},\{23,33,78,49\},\{78,33,82\}\Big\}$
2. $\mathscr{F}_2 \leftarrow \Big\{\{15,18\},\{23,78\},\{19,82\}\Big\}$
3. $\mathscr{F}_3 \leftarrow \Big\{\{15,49,18\},\{23,78\},\{19,33,82\}\Big\}$

```
>>> U = conj(15,23,78,49,18,33,19,82)
>>> F1 = conj(conj(15,49,18,82), conj(23,33,78,49),conj(78,33,82))
>>> F2 = conj(conj(15,18), conj(23,78), conj(19,82))
>>> F3 = conj(conj(15,49,18), conj(23,78), conj(19,33,82))
>>> esPart(F1, U)
False
>>> esPart(F2, U)
False
>>> esPart(F3, U)
True
>>>
```

$\mathscr{F}_1$ no es partición de U, ya que al menos el 33 aparece en dos conjuntos; $\mathscr{F}_2$ tampoco es una partición porque no es un cubrimiento de U; pero $\mathscr{F}_3$ si es una partición de U.

Ejercicios

1. Calcula todos los subconjuntos de $A \leftarrow\!\shortmid \{k, l, m, n\}$.
2. Enlista todos los elementos del siguiente conjunto:

$$\left\{A \in \mathcal{P}(\{1,2,3,4\}) \,\middle|\, 1 \leq |A| \leq 2\right\}$$

3. Sean $A_1 \leftarrow\!\shortmid \{0,2,4,6\}$, $A_2 \leftarrow\!\shortmid \{2,4,6,8,10\}$, $A_3 \leftarrow\!\shortmid \{1,2,3,4,5\}$ y $A_4 \leftarrow\!\shortmid \{1,2,3,7,8,9\}$. Calcula:
   a) $\bigcup_{i=1}^{4} A_i$
   b) $\bigcap_{i=1}^{4} A_i$
   c) $\bigcup \mathcal{P}((A_1 \cap A_2))$

4. Escribe una función recursiva en `Python` para obtener la intersección generalizada. Tu función debe seguir el algoritmo recursivo de la página 112. El nombre de la función debe ser `Inters` y se debe aplicar a una familia no determinada de conjuntos `FamC`. La respuesta debe ser un conjunto que contiene la intersección de todos los conjuntos en la familia `FamC`.

```
def Inters(*FamC)-> list:
    """
    Intersección generalizada
    de una lista no determinada de conjuntos.
    """
    pass  #<-- aquí escribe tu código
```

```
>>> Inters(conj(1,2,3,4,5), conj(1,3,5,7,9), conj(1,2,4,6,8,10,12), conj(1,8])
[1]
>>> Inters(conj(1,2,3,4,5), conj(1,3,5,7,9), conj(), conj(1,8))
[]
>>> Inters()
[]
>>>
```

5. Considera el universo $U \leftarrow\!\shortmid \{3,7,2,9,5,11,23,18,72\}$. Rellena la siguiente tabla con información sobre las siguientes familias de subconjuntos de U:

	¿Es cubrimiento de U?	¿Es partición de U?
$\{\{3,2,9,11\},\{72,7,23,2\},\{5,11,13\}\}$		
$\{\{2,9,11\},\{3,7,72,5\},\{23,18\}\}$		
$\{\{7,9,11\},\{18\},\{3,2,5,23,72\}\}$		
$\{\{3,9,5\},\{7,5,11\},\{2,11,23\}\}$		

6. Hay una dinámica de grupos llamada «grupos de», que consiste en que el organizador grita «hagan grupos de...» y luego menciona un número. Los participantes del juego corren para formar grupos con la cantidad de integrantes dichos por el organizador. Los participantes que no logran acomodarse en algún grupo son descalificados y el juego inicia nuevamente con los competidores no descalificados. El juego continúa hasta que solamente queda un competidor o bien hasta que el organizador termina de mencionar su lista de números. Los ganadores son aquellos que al terminar la serie de números dicha por el organizador, aún quedan activos; o bien solamente queda un único competidor.

Escribe un programa en `Python` que se llame `gruposDe.py` que reciba como entrada dos listas: la colección de números enteros que debe gritar el organizador y los números que representan a los participantes. La salida de la función es un número entero que indica cuántas personas son las ganadoras del juego.
El programa puede terminar en tres casos:

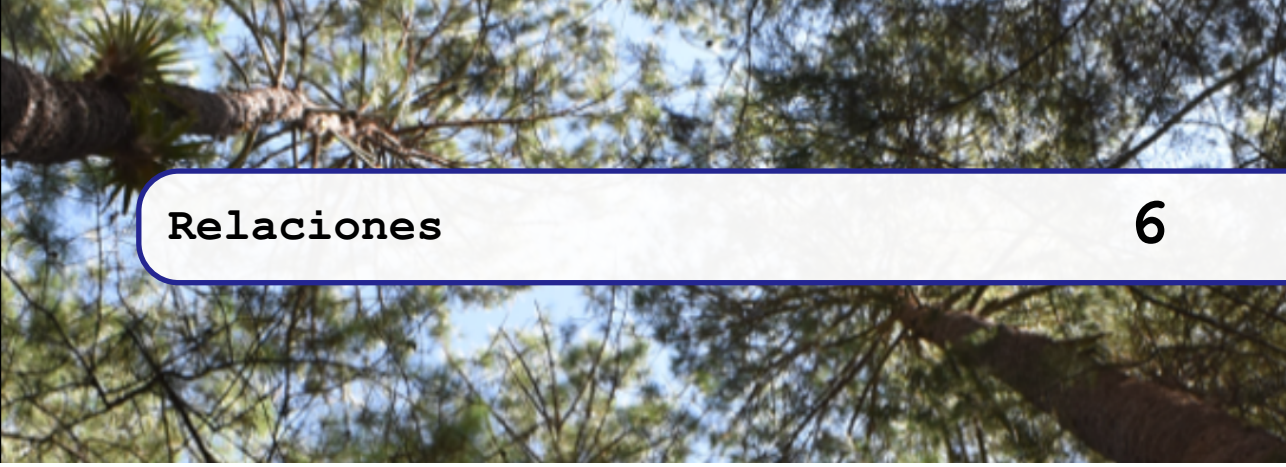
a) Cuando ya no hay más números que el organizador pueda gritar. En este caso el programa debe responder con la cantidad de participantes que quedan.
b) Cuando el número gritado es mayor que la cantidad de participantes. En este caso el programa devuelve 0.
c) Cuando solamente queda un único participante. El programa devuelve 1.

En cualquier otro caso, el programa debe quitar los participantes que no consiguieron entrar en algún grupo y luego repetir el proceso con el siguiente número en la lista de números del organizador.
Ejemplos:

Entrada	Salida	Explicación
`4 5 6 3 2` `1 2 3 4 5 6 7 8 9`	`0`	Se piden grupos de 4, se elimina 1; luego grupos de 5, se eliminan 3; luego grupos de 6, como quedan 5, la respuesta es 0.
`4 5 3 3 2` `1 2 3 4 5 6 7 8 9`	`2`	Se piden grupos de 4, se elimina 1; luego grupos de 5, se eliminan 3; luego grupos de 3, se eliminan 2; de nuevo grupos de 3 y se eliminan 0; finalmente se piden grupos de 2, queda fuera 1. Como ya no hay más números por gritar, el resultado es 2.

III

Relaciones y Funciones



Relaciones 6

6.1 Tuplas

Las tuplas sirven para organizar datos, por ejemplo los registros en una base de datos. En general se pueden organizar objetos en donde se requiera tener diferentes valores guardados de manera ordenada.

Definición 6.1.1 -- Tupla. Una tupla es una colección ordenada de elementos. Se establecen los límites de una tupla con corchetes triangulares $\langle \bullet \rangle$. Lo que se encuentra dentro de los corchetes es la colección ordenada.

Ejemplo 6.1

$\langle a, b, c \rangle$ es una tupla. $\langle 5 \rangle$ es otra tupla.

Hay diferentes notaciones para las tuplas, por ejemplo para denotar la tupla $\langle a, b, c \rangle$ se puede escribir (a, b, c) o también $[a, b, c]$, o incluso cuando no hay confusión entre los elementos de la tupla y los símbolos que la rodean, se puede escribir como una palabra abc o `abc`. Generalmente se escriben separando los elementos de la tupla con una coma y delimitando con corchetes de cualquier tipo excepto las llaves, que son utilizadas para delimitar conjuntos.

Convencionalmente las letras griegas α, β, γ o las letras $a, b, c, \ldots$ suelen utilizarse para representar tuplas, mientras que los elementos generalmente tienen subíndices, como $\alpha \leftarrow\!\shortmid \langle \alpha_1, \alpha_2, \alpha_3 \rangle$, donde α es la tupla y α_1, α_2 y α_3 son literales que representan a los elementos de la tupla y su respectivo orden.

Una n-tupla, donde $n \in \mathbb{N}^*$, es una tupla de n elementos. Cuando $n = 0$ se tiene la tupla $\langle\,\rangle$ que tiene 0 elementos; si $n > 0$ se tiene una tupla de la forma $\langle \alpha_1, \ldots, \alpha_n \rangle$ que tiene n elementos; una tupla que tiene al menos un elemento, también se puede representar en términos de su primer elemento y el resto de ellos, como en $\alpha \leftarrow\!\shortmid \langle \alpha_1 | \alpha' \rangle$,

donde α_1 es una literal que representa al primer elemento de la tupla α, es justo aquel que se encuentra más a la izquierda; y α' es a su vez una tupla que contiene de manera ordenada, todos los elementos de α excepto el primero.

En `Python` las tuplas son un tipo de dato primitivo, son estructuras creadas con base en una lista de elementos y que no pueden ser modificadas. se escriben entre paréntesis y sus índices empiezan desde 0.

```
>>> a = tuple([1,2,3])
>>> a
(1,2,3)
>>> print(type(a))
<class 'tuple'>
>>> a[0]
1
>>>
```

En este libro utilizaremos las listas para modelar tuplas. El tipo `tuple` será utilizado en pocas ocasiones y con propósitos muy particulares.

Ejemplo 6.2

Las siguientes son tuplas:

1. $\langle\,\rangle$, es la tupla sin elementos.
2. $\langle a \rangle$, es llamado «solitón», es una 1-tupla.
3. $\langle a, b \rangle$, es una 2-tupla, también se le conoce como «par ordenado» o «pareja ordenada».
4. $\langle a, b, c \rangle$, es una 3-tupla o triada.

Particularmente para el caso de los pares ordenados, será de mucha ayuda tener un par de funciones que nos permitan obtener el primer elemento del par y el segundo elemento del par.

```
1  def pPar(t:list):
2      """ Primero de par.
3      Obtiene el primer elemento de un par ordenado.
4      """
5      return car(t)
6
7  def sPar(t:list):
8      """ Segundo de par.
9      Obtiene el primero del resto de un par ordenado.
10     """
11     return car(cdr(t))
```

```
>>> pPar(['x', 'y'])
'x'
>>> sPar(['x', 'y'])
'y'
>>>
```

6.1.1 Verificación de una tupla

En `Python` modelamos tuplas usando listas, por lo que, hacer operaciones con tuplas es lo mismo que hacer operaciones con listas. Ahora definiremos un predicado que verifica que un objeto sea una tupla.

Código 6.1: *Verifica una tupla*

```
1  def esTupla(t)-> bool:
2      """ Verifica que un objeto sea una tupla.
3      Devuelve True, si t es precisamente una lista.
4      """
5      return isinstance(t, list)
```

¡Advertencia!: En este libro utilizamos listas para modelar tanto conjuntos como tuplas, por lo que el código fuente debe interpretarse con mucho cuidado, cuándo se trata de un conjunto y cuándo se trata de una tupla.

La siguiente interacción muestra cómo puede darse una mala interpretación al código fuente. Se crea un conjunto, pero se verifica si se trata de una tupla.

```
>>> A = conj(1,2,3) # se crea el conjunto {1,2,3} y se asigna a A.
>>> A
[1, 2, 3]
>>> esTupla(A)
True
>>>
```

6.1.2 Creación de tuplas

Una tupla puede ser creada al dar los elementos de la tupla, el procedimiento `tupla` puede tomar cualquier número de parámetros y crea una tupla con ellos.

$$\texttt{tupla}() \mapsto \langle\,\rangle$$
$$\texttt{tupla}(1,2,3,4) \mapsto \langle 1,2,3,4\rangle$$

Código 6.2: *Creación de tuplas*

```
1 def tupla(*items)-> list:
2     """ Crea una tupla.
3     Recibe una lista no determinada de elementos
4     de cualquier tipo, pero devuelve una lista con ellos.
5     """
6     return list(items)
```

Es importante hacer notar que la aridad de esta función es de al menos 0 elementos, esto significa que se puede invocar la función con 0 o más argumentos.

```
>>> tupla()
[]
>>> tupla(1,2,3)
[1, 2, 3]
>>>
```

6.2 La tupla vacía

La tupla vacía es una tupla que no tiene elementos. La tupla vacía se puede denotar como $\langle\,\rangle$. En otras áreas de las matemáticas discretas como en la teoría de lenguajes formales, una tupla es lo mismo que una palabra y se emplea el símbolo ε para representar a la palabra vacía. En `Python` podemos darle un nombre especial para denotar la tupla vacía como:

```
1 # Se define la tupla vacía
2 tvacia:list = tupla()
```

Al definir un concepto nuevo, es útil definir un predicado que determine `True` si un objeto corresponde con ese nuevo concepto. En este caso crearemos un predicado que determine `True` si el objeto es una tupla vacía, claramente devolverá `False` si el objeto no es una tupla vacía.

Código 6.3: *Verifica la tupla vacía*

```
def esTvacia(t:list)-> bool:
    """ Verifica que una tupla esté vacía.
    Devuelve True si el argumento es la constante tvacia.
    Devuelve False si no lo es.
    """
    return t == tvacia
```

```
>>> esTvacia(4)
False
>>> esTvacia([2,3,4])
False
>>> esTvacia([])
True
>>>
```

6.3 Longitud de una tupla

Definición 6.3.1 -- Longitud de una tupla. Sea α una tupla. La longitud de α se denota $|\alpha|$ y es la cantidad de elementos que se encuentran en α.

En las tuplas importa si hay elementos repetidos, así $\langle a, a\rangle$ es una tupla con dos elementos diferentes en la posición en que se encuentran.

Ejemplo 6.3

Los siguientes ejemplos sirven para ilustrar el concepto de longitud en las tuplas.

- La longitud de la tupla $\langle\,\rangle$ es $|\langle\,\rangle| \mapsto 0$.
- La longitud de la tupla $\langle g\rangle$ es $|\langle g\rangle| \mapsto 1$.
- La longitud de la tupla $\langle g, o, w\rangle$ es $|\langle g, o, w\rangle| \mapsto 3$.
- La longitud de la tupla $\langle g, g, o, w, o, w, w\rangle$ es $|\langle g, g, o, w, o, w, w\rangle| \mapsto 7$.

La longitud de una tupla puede calcularse de manera recursiva de cola, utilizando una variable opcional ℓ con valor por defecto $\ell \leftarrow 0$ como en el siguiente algoritmo:

$$|\alpha, [\ell \leftarrow 0]| \mapsto \begin{cases} \ell & \text{si } \alpha = \langle\,\rangle \\ \\ |\alpha', \ell + 1| & \text{si } \alpha = \langle \alpha_1 | \alpha' \rangle. \end{cases}$$

Código 6.4: *Longitud de una tupla*

```
def tLong(t:list, l:int=0)-> int:
    """ Calcula la longitud de una tupla.
    Versión recursiva que cuenta los elementos en t.
    """
    if esTvacia(t):
        return l
    else:
        return tLong(cdr(t), l+1)
```

Ejemplo 6.4

Utiliza el programa `tLong` para calcular la longitud de las tuplas $\alpha \leftarrow \langle a, s, a\rangle$ y $\beta \leftarrow \langle c, a, m, e, m, a, c\rangle$.

```
>>> alfa = tupla('a','s','a')
>>> beta = tupla('c','a','m','e','m','a','c')
>>> tLong(alfa)
3
>>> tLong(beta)
7
>>>
```

6.3.1 El solitón

Un caso particularmente especial es una tupla llamada «solitón».

Definición 6.3.2 -- Solitón. Un solitón es una tupla que tiene un único elemento. Otro nombre es «tupla unitaria».

Los solitones son especiales porque sirven como base para muchas otras construcciones basadas en listas, por ejemplo en los caminos unitarios en grafos o palabras unitarias en lenguajes formales; también sirven de referencia en procesos iterativos.

Código 6.5: *Verifica solitón*

```
1 def esSoliton(t:list)-> bool:
2     """ Verifica que una tupla sea solitón.
3     Se proporciona una lista y se calcula su longitud,
4     si la longitud es 1, entonces es un solitón.
5     """
6     return tLong(t) == 1
```

```
>>> esSoliton(4)
False
>>> esSoliton([2,3,4])
False
>>> esSoliton([2])
True
>>> esSoliton([tvacia])
True
>>>
```

6.4 Operaciones con tuplas

Las operaciones con tuplas son operaciones que reciben como argumento al menos una tupla. El resultado puede ser de diferente tipo. Podemos clasificar las operaciones con tuplas en dos categorías:

1. **Las operaciones de comparación**, que devuelven un valor booleano. Las operaciones de comparación también son funciones booleanas, ya que devuelven como resultado un valor booleano.
2. **Otras operaciones**, estas operaciones devuelven como resultado una nueva tupla.

Ya antes se han definido algunos predicados relacionados con tuplas, por ejemplo:

- `esTupla` [código 6.1, página 122];
- `esTvacia` [código 6.3, página 124] y
- `esSoliton` [código 6.5, página 125].

Otras operaciones con tuplas permiten crear nuevas tuplas a partir de los argumentos y tienen propósitos específicos.

6.4.1 Tuplas iguales

Definición 6.4.1 -- Tuplas iguales. Si α y β son tuplas, α es igual a β, denotado $\alpha = \beta$, si ambas tuplas tienen los mismos elementos en las mismas posiciones.

Ejemplo 6.5

Las tuplas $\langle 3,6,2,8,2\rangle$ es diferente que la tupla $\langle 2,2,8,6,3\rangle$ porque a pesar de que tienen elementos de igual valor, esto se encuentran en diferente posición. Las tuplas $\langle 3,6,2,8,2\rangle$ y $\langle 3,6,2,8,2\rangle$ son iguales.

Para determinar recursivamente que tuplas son iguales, se procede en cuatro casos:

1. Si exactamente una de las dos tuplas es la tupla vacía, claramente no son iguales.
2. Si ambas tuplas son vacías, claramente son iguales.
3. Si las primeras dos pruebas no sucedieron, significa que ambas tuplas son de la forma $\langle a_1|a'\rangle$. Si el primero de una tupla es igual al primero de la segunda tupla, se verifica recursivamente si el resto de las tuplas son iguales.
4. En cualquier otro caso las tuplas son diferentes.

$$\alpha = \beta \mapsto \begin{cases} \texttt{False} & \text{si } \alpha = \langle\,\rangle \oplus \beta = \langle\,\rangle \\ \texttt{True} & \text{si } \alpha = \langle\,\rangle \wedge \beta = \langle\,\rangle \\ \alpha' = \beta' & \text{si } \alpha_1 = \beta_1 \qquad \#\,\alpha \hookleftarrow \langle \alpha_1|\alpha'\rangle, \beta \hookleftarrow \langle \beta_1|\beta'\rangle \\ \texttt{False} & eoc. \end{cases}$$

Ejemplo 6.6

Los siguientes ejemplos ilustran cada caso:

1. Las tuplas $\alpha \hookleftarrow \langle 1,2,3\rangle$ y $\beta \hookleftarrow \langle\,\rangle$ no son iguales por el caso 1.
2. Las tuplas $\alpha \hookleftarrow \langle\,\rangle$ y $\beta \hookleftarrow \langle\,\rangle$ sí son iguales por el caso 2.
3. Para verificar que $\alpha \hookleftarrow \langle 5,7\rangle$ es igual a $\beta \hookleftarrow \langle 5,7\rangle$, se verifica que $5_\alpha = 5_\beta$ [el subíndice indica de qué tupla se ha obtenido el valor]. Por el caso 3, se debe verificar que $\alpha' \hookleftarrow \langle 7\rangle$ sea igual que $\beta' \hookleftarrow \langle 7\rangle$. Como $7_\alpha = 7_\beta$ son iguales, se debe verificar que $\langle\,\rangle = \langle\,\rangle$, luego por el caso 2, ambas tuplas son iguales.
4. Las tuplas $\alpha \hookleftarrow \langle 5,7\rangle$ y $\beta \hookleftarrow \langle 8,7\rangle$ no son iguales por el caso 4.

Código 6.6: *Igualdad entre tuplas*

```
def tIguales(a:list, b:list)-> bool:
    """ Verifica recursivamente la igualdad entre dos tuplas.
    El resultado es True si son iguales y False en otro caso.
    """
    if ox(esTvacia(a), esTvacia(b)):
        return False
    elif y(esTvacia(a), esTvacia(b)):
        return True
    elif car(a)==car(b):
        return tIguales(cdr(a), cdr(b))
    else:
        return False
```

Ejemplo 6.7

Utiliza la función `tIguales` para comprobar los resultados del ejemplo 6.6.

```
>>> alpha = [1,2,3]
>>> beta = []
>>> tIguales(alpha, beta)
False
>>> alpha = []
>>> tIguales(alpha, beta)
True
>>> alpha = [5,7]
>>> beta = [5,7]
>>> tIguales(alpha, beta)
True
>>> beta = [8,7]
>>> tIguales(alpha, beta)
False
>>>
```

6.4.2 Concatenación tuplas

Definición 6.4.2 -- Concatenación. Si $\alpha \leftarrow \langle a_1,\ldots,a_n \rangle$ y $\beta \leftarrow \langle b_1,\ldots,b_m \rangle$ son dos tuplas, la concatenación de α con β se denota $\alpha \bullet \beta$ y es una nueva tupla:

$$\alpha \bullet \beta \mapsto \begin{cases} \alpha & \text{si } \beta = \langle\,\rangle \\ \beta & \text{si } \alpha = \langle\,\rangle \\ \langle a_1,\ldots,a_n,b_1,\ldots,b_m \rangle & eoc \end{cases}$$

En otras palabras, para obtener la concatenación de dos tuplas, se deben escribir todos los elementos de la primera tupla y en seguida todos los elementos de la segunda tupla.

Ejemplo 6.8

Si $\alpha \leftarrow \langle x,y,z \rangle$ y $\beta \leftarrow \langle 3,7,8 \rangle$, la concatenación de α con β produce la tupla $\langle x,y,z,3,7,8 \rangle$, mientras que la concatenación de β con α produce $\langle 3,7,8,x,y,z \rangle$.

Es un convenio no escrito omitir el símbolo de la concatenación $\bullet$ cuando sea claro, siguiendo el mismo criterio que el producto aritmético. Los paréntesis también tienen el significado de la concatenación, al igual que en el producto aritmético.

Los siguientes ejemplos ilustran la manera en que puede ser escrita la concatenación:

- $\alpha \bullet \beta$ puede ser reescrita como $\alpha\beta$.
- $\alpha \bullet \beta \bullet \gamma$ se puede reescribir quitando el símbolo de la concatenación como $\alpha\beta\gamma$

Se pueden utilizar paréntesis para obligar el orden de las operaciones, así $\alpha\beta\gamma$ puede hacerse:

$(\alpha\beta)\gamma$ lo que significa hacer primero $\alpha \bullet \beta$ y luego concatenar el resultado con γ.

$\alpha(\beta\gamma)$ lo que significa concatenar α con el resultado de $\beta \bullet \gamma$.

La concatenación de tuplas tiene la propiedad asociativa, esto es que para cualesquiera tuplas α, β y γ se cumple que:

$$(\alpha\beta)\gamma = \alpha(\beta\gamma) = \alpha\beta\gamma$$

En general, la concatenación de tuplas no tiene la propiedad conmutativa, esto es que no siempre sucede que $\alpha\beta = \beta\alpha$, salvo en casos muy específicos como cuando ambas tuplas sean iguales.

Código 6.7: *Concatenación de tuplas*

```
1 def tConcat(a:list, b:list)-> list:
2     """ Concatenación.
3     Concatena por la derecha la tupla b a la tupla a.
4     """
5     return a + b
```

Ejemplo 6.9

Utiliza el operador `tConcat` para concatenar las tuplas $\langle x,y,z \rangle$ con $\langle 3,7,8 \rangle$.

```
>>> alpha = tupla('x','y','z')
>>> beta = tupla(3,7,8)
>>> tConcat(alpha, beta)
['x', 'y', 'z', 3, 7, 8]
>>>
```

La longitud de la tupla generada por la concatenación de dos tuplas, es el resultado de sumar la longitud de una tupla con la longitud de la otra, por lo que si $\alpha \leftarrowtail \langle a_1, \ldots, a_n \rangle$ y $\beta \leftarrowtail \langle b_1, \ldots, b_m \rangle$, $|\alpha \bullet \beta|$ es

$$|\langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle| \mapsto n + m$$

6.4.3 Escribir por la derecha de una tupla

Esta operación es motivo de discusión porque en algunos lenguajes de programación, las tuplas son objetos inmutables, esto significa que una vez que ha sido definida una tupla, esta no puede modificarse. Otros lenguajes no son tan estrictos y permiten que las tuplas puedan ser modificadas.

La operación de escribir por la derecha en una tupla, tiene el propósito de generar [o modificar] una tupla anexando por la derecha un nuevo elemento. Así por ejemplo, si se tiene una tupla $\langle 1, 2, 3 \rangle$ y se desea escribir por la derecha el nuevo elemento 4, el resultado es la tupla $\langle 1, 2, 3, 4 \rangle$.

Definición 6.4.3 -- Agregar por derecha. Sea $\alpha \leftarrowtail \langle a_1, \ldots, a_n \rangle$ una tupla y e un elemento cualquiera. La operación `tEscribe` toma la tupla α y le escribe por la derecha el elemento e, produciendo

$$\texttt{tEscribe}(\alpha, e) \mapsto \langle a_1, \ldots, a_n, e \rangle$$

El resultado de esta operación es una tupla que tiene a e como último elemento [en el extremo derecho]. Para lograr esto simplemente se crea un solitón con e luego, ya teniendo las dos tuplas α y $\langle e \rangle$, solo es cuestión de concatenarlas en el orden adecuado:

$$\begin{aligned} \texttt{tEscribe}(\alpha, e) &= \langle a_1, \ldots, a_n \rangle \bullet \texttt{tupla}(e) \\ &= \langle a_1, \ldots, a_n \rangle \bullet \langle e \rangle \\ &= \langle a_1, \ldots, a_n, e \rangle \end{aligned}$$

Ejemplo 6.10

Una institución dedicada a investigar cuestiones de salud, ha creado una base de datos en la que cada dato contiene información de personas. La información solicitada es sobre la identidad, sexo y edad. Por lo que con cada persona entrevistada se crea una tupla con su información. La siguiente tabla muestra los datos de algunas personas entrevistadas.

Nombre	Sexo	Edad
Ana	F	13
Juan	M	18
Sebastián	M	30

Después de la pandemia de COVID19, hubo la necesidad de agregar nueva información. Se desea saber si las personas han sido vacunadas, por lo que se agrega esta nueva información y se ha solicitado a cada persona que aporte esa información.

Nombre	Sexo	Edad	Vacunado
Ana	F	13	False
Juan	M	18	True
Sebastián	M	30	True

Código 6.8: *Escribe por derecha en una tupla*

```
def tEscribe(t:list,e)-> list:
    """ Escribe por derecha.
    Escribe el elemento e al final de la tupla t.
    """
    if esTupla(t):
        return tConcat(t,tupla(e))
    else:
        return tConcat(tupla(t), tupla(e))
```

Ejemplo 6.11

Utiliza la función `tEscribe` para:

1. Escribir `"x"` a la tupla vacía.
2. Escribir `"y"` al resultado anterior.
3. Escribir `"z"` al resultado anterior.

```
>>> tEscribe(tvacia, 'x')
['x']
>>> tEscribe(['x'], 'y')
['x', 'y']
>>> tEscribe(['x', 'y'], 'z')
['x', 'y', 'z']
>>>
```

6.4.4 Dividir una lista en la posición *k*

Esta operación toma como entrada una tupla y devuelve un par ordenado [tupla con dos elementos], cuyos elementos son tuplas.

Definición 6.4.4 -- Divide una tupla. Si $\alpha \leftarrowtail \langle a_1,\ldots,a_n \rangle$ es una tupla con $n \in \mathbb{Z}$ elementos, la operación `tKdivide` divide la tupla α en dos y crea una nueva tupla con las partes.

$$\texttt{tKdivide}(\alpha,k) \mapsto \begin{cases} \langle \langle\,\rangle, \alpha \rangle & \text{si } k < 0 \\ \langle \langle a_1,\ldots,a_k \rangle, \langle a_{k+1},\ldots,a_n \rangle \rangle & \text{si } 0 \leq k \leq n \\ \langle \alpha, \langle\,\rangle \rangle & \text{si } k > n \end{cases}$$

Ejemplo 6.12

Sea $\alpha \leftarrowtail \langle 4,6,2,8,8,9,3 \rangle$ una tupla. Al dividir la tupla en la posición 4 se obtiene:

$$\begin{aligned} \texttt{tKdivide}(\alpha,4) &= \texttt{tKdivide}(\langle 4,6,2,8,8,9,3 \rangle, 4) \\ &\mapsto \langle \langle 4,6,2,8 \rangle, \langle 8,9,3 \rangle \rangle \end{aligned}$$

El programa en `Python` sigue fielmente la definición de la función.

Código 6.9: *Dividir una tupla*

```
def tKdivide(t:list, k:int)-> list:
    """ Divide una lista en la posición k.
    Devuelve una tupla con las los partes de la tupla original.
    """
    if k<0:
        return tupla([],t)
    if k>tLong(t):
        return tupla(t,[])
    else:
        return tupla(t[:k], t[k:])
```

Al dividir una tupla se obtiene un par ordenado, cada elemento es una parte de la tupla original. Recuerda que el primero de una tupla es el que se encuentra más a la izquierda. La operación `car` [código 2.1, página 50] obtiene el primer elemento de una tupla. Para obtener el segundo elemento se puede hacer utilizando una combinación de `car` y `cdr`.

Ejemplo 6.13

Una empresa tiene en una base de datos, una tabla con la estructura T:

Índice	Nombre	Apellido	TipoContrato	FechaIngreso

Al hacer una reestructuración de la tabla, podían omitir el campo `FechaIngreso` y requerían agregar un nuevo campo en dependencia del tipo de contrato. Si el tipo de contrato es `eventual` se debía agregar el campo `MaxTC` que se refiere al máximo tiempo de contrato, expresado en meses; pero si el tipo de contratación es `base`, entonces se debe agregar el campo `Ant` que se refiere al tiempo transcurrido desde la basificación, expresado en años.

La reestructuración se debe hacer en pasos. El primer paso es dividir la tabla en la posición 4. Luego hacer dos nuevas tablas, agregando el campo adecuado.

1. $T \leftarrow \langle$ Índice, Nombre, Apellido, TipoContrato, FechaIngreso $\rangle$
2. $TD \leftarrow$ tKdivide$(T, 4)$
   # $TD \leftarrow \langle \langle$ Índice, Nombre, Apellido, TipoContrato $\rangle, \langle$ FechaIngreso $\rangle \rangle$
3. $T_a \leftarrow$ car(TD)
   # $T_a \leftarrow \langle$ Índice, Nombre, Apellido, TipoContrato $\rangle$
4. $T_b \leftarrow$ tEscribe$(Ta$, MaxTC$)$
   # $T_b \leftarrow \langle$ Índice, Nombre, Apellido, TipoContrato, MaxTC $\rangle$
5. $T_c \leftarrow$ tEscribe$(Ta$, Ant$)$
   # $T_c \leftarrow \langle$ Índice, Nombre, Apellido, TipoContrato, Ant $\rangle$

6.4.5 Escribir en alguna posición de la tupla

Esta acción permite modificar una tupla $t \leftarrow \langle a_1, \ldots, a_n \rangle$ escribiendo un elemento e en cualquiera de las n posiciones, generando una tupla de longitud $n+1$.

La función `tEscribeEnPos` toma un elemento cualquiera e y lo inserta en la tupla t en una posición k, con $0 \leq k \leq |t|$, generando una tupla de longitud $n+1$. Si $\alpha \leftarrow \langle a_1, \ldots, a_n \rangle$, e es un elemento y k un número, se tiene:

$$\begin{aligned} \texttt{tEscribeEnPos}(\alpha, e, k) &= \texttt{tEscribeEnPos}(\langle a_1, \ldots, a_n \rangle, e, k) \\ &\mapsto \langle a_1, \ldots, a_{k-1}, e, a_k, \ldots, a_n \rangle \end{aligned}$$

Código 6.10: *Escribe un elemento en una tupla dando su posición*

```
def tEscribeEnPos(t,e,k)-> list:
    """ Escribe un elemento en una posición dada.
    Devuelve una nueva lista, incluyendo el elemento insertado.
    """
    Ta = tKdivide(t, k-1)
    Tb = car(Ta)
    Tc = car(cdr(Ta))
    return tConcat(tEscribe(Tb, e), Tc)
```

```
>>> tEscribeEnPos([1,2,3,4,5,6,7,8], 'x', 1)
['x', 1, 2, 3, 4, 5, 6, 7, 8]
>>> tEscribeEnPos([1,2,3,4,5,6,7,8], 'x', 8)
[1, 2, 3, 4, 5, 6, 7, 'x', 8]
>>> tEscribeEnPos([1,2,3,4,5,6,7,8], 'x', 0)
['x', 1, 2, 3, 4, 5, 6, 7, 8]
>>>
```

6.4.6 Tupla inversa

Definición 6.4.5 La tupla inversa de una tupla α, se denota α^{-1}, que es definida como sigue:

$$\alpha^{-1} \mapsto \begin{cases} \langle\,\rangle & \text{si } \alpha = \langle\,\rangle \\ \langle a_n,\ldots,a_1\rangle & \text{si } \alpha = \langle a_1,\ldots,a_n\rangle \end{cases}$$

La tupla inversa de una tupla es una tupla que contiene los mismos elementos que la tupla original, pero colocados en orden inverso.

Ejemplo 6.14

α	α^{-1}
$\langle\,\rangle$	$\langle\,\rangle$
$\langle 1\rangle$	$\langle 1\rangle$
$\langle 1,2\rangle$	$\langle 2,1\rangle$
$\langle 1,2,3\rangle$	$\langle 3,2,1\rangle$

6.4.7 La concatenación generalizada

La operación de concatenación puede ser extendida para crear un operador de concatenación que trabaje con cero, una o más tuplas. Esta operación la llamaremos «concatenación generalizada». Suponiendo que $\alpha_1,\ldots,\alpha_k$ son tuplas, la concatenación generalizada de todas esas tuplas se puede escribir:

$$\prod_{i=1}^{k} \alpha_i,$$

lo que significa concatenar todas las tuplas α_i, desde $i \leftarrow 1$ hasta $i \leftarrow k$; esto es $\alpha_1 \bullet \cdots \bullet \alpha_k$. Sin embargo la operación de concatenación fue definida con aridad 2, por lo que la concatenación generalizada se debe resolver haciendo la concatenación a pares.

$$(\ldots((\alpha_1 \bullet \alpha_2)\bullet\alpha_3)\bullet\ldots)\bullet\alpha_k$$

Ejemplo 6.15

Calcula la concatenación de las tuplas $\alpha_1 \leftarrow \langle 3,7,3\rangle, \alpha_2 \leftarrow \langle$ `"Pao"`$,7,$`True`$\rangle$ y $\alpha_3 \leftarrow \langle 10.5,13.2\rangle$ en el orden en que aparecen.

$$\begin{aligned} \prod_{i=1}^{k} \alpha_i &= \alpha_1 \bullet \alpha_2 \bullet \alpha_3 \\ &= (\alpha_1 \bullet \alpha_2)\bullet\alpha_3 \\ &= (\langle 3,7,3\rangle \bullet \langle \texttt{"Pao"},7,\texttt{True}\rangle)\bullet\alpha_3 \\ &= \langle 3,7,3,\texttt{"Pao"},7,\texttt{True}\rangle \bullet \alpha_3 \\ &= \langle 3,7,3,\texttt{"Pao"},7,\texttt{True}\rangle \bullet \langle 10.5,13.2\rangle \\ &\mapsto \langle 3,7,3,\texttt{"Pao"},7,\texttt{True},10.5,13.2\rangle \end{aligned}$$

El procedimiento salta a la vista. Primero se concatenan las primeras dos tuplas, el resultado se concatena con la tercera, el procedimiento continúa hasta concatenar el penúltimo resultado con la última tupla. Claramente si no hay tuplas que concatenar, el resultado es la tupla vacía.

Código 6.11: *Concatenación generalizada*

```
def TConcat(*T)-> list:
    T = tupla(*T)
    if esTvacia(T):
        return tvacia
    if esSoliton(T):
        return car(T)
    else:
        x = tConcat(car(T), car(cdr(T)))
        r = tConcat(tupla(x), cdr(cdr(T)))
        return TConcat(*r)
```

Ejemplo 6.16

Utiliza la función `TConcat` para concatenar las tuplas α_1, α_2 y α_3 del ejercicio 6.15.

```
>>> alpha1 = tupla(3,7,3)
>>> alpha2 = tupla('Pao', 7, True)
>>> alpha3 = tupla(10.5, 13.2)
>>> TConcat(alpha1, alpha2, alpha3)
[3, 7, 3, 'Pao', 7, True, 10.5, 13.2]
>>>
```

6.5 Producto cartesiano

El producto cartesiano es una operación que tiene como origen el sistema de referencia que fue ideado por René Descartes que dio origen a la Geometría Analítica.

6.5.1 Producto cartesiano de dos conjuntos

Definición 6.5.1 -- Producto Cartesiano. Sean A y B dos conjuntos. El **producto cartesiano** de los conjuntos A y B se escribe $A \times B$, y es el conjunto

$$\{\langle a, b\rangle | a \in A \wedge b \in B\}.$$

Así $A \times B$ es un conjunto de tuplas, donde cada tupla se construye con su primer elemento tomado del conjunto A y el segundo es tomado del conjunto B. Sin embargo una característica sumamente importante es que sigue siendo un conjunto.

Ejemplo 6.17

Supongamos que $A = \{\texttt{ana}, \texttt{mar}, \texttt{fer}\}$ es un conjunto de nombres y $B = \{\texttt{fresa}, \texttt{chocolate}\}$ el conjunto de sabores de helados. El producto cartesiano de A con B se escribe $A \times B$ y es el conjunto

$$A \times B = \{\langle \texttt{ana}, \texttt{fresa}\rangle, \langle \texttt{ana}, \texttt{chocolate}\rangle, \langle \texttt{mar}, \texttt{fresa}\rangle, \langle \texttt{mar}, \texttt{chocolate}\rangle, \langle \texttt{fer}, \texttt{fresa}\rangle, \langle \texttt{fer}, \texttt{chocolate}\rangle\}$$

Si $A \leftarrow \{a_1, \ldots, a_n\}$ y $B \leftarrow \{b_1, \ldots, b_n\}$ son dos conjuntos, el producto cartesiano $A \times B$ se puede representar mediante una tabla, donde los renglones son identificados con cada elemento del primer conjunto y las columnas se identifican con los elementos del otro conjunto.

$A \times B$	b_1	b_2	$\ldots$	b_m
a_1	$\langle a_1, b_1\rangle$	$\langle a_1, b_2\rangle$	$\ldots$	$\langle a_1, b_m\rangle$
a_2	$\langle a_2, b_1\rangle$	$\langle a_2, b_2\rangle$	$\ldots$	$\langle a_2, b_m\rangle$
$\vdots$	$\vdots$	$\vdots$	$\ldots$	$\vdots$
a_n	$\langle a_n, b_1\rangle$	$\langle a_n, b_2\rangle$	$\ldots$	$\langle a_n, b_m\rangle$

En general $A \times B \neq B \times A$, esto es porque en $A \times B$, el primer elemento de cada par ordenado pertenece al conjunto A y el segundo elemento de cada par pertenece a B; en $B \times A$ los primeros elementos pertenecen a B mientras que los segundos elementos pertenecen al conjunto A.

Código 6.12: *El producto cartesiano de dos conjuntos*

```
def pCart(A:list, B:list)-> list:
    """
    Calcula el producto cartesiano de dos conjuntos.
    """
    PC = enCada(lambda a: enCada(lambda b: tEscribe(a,b), B), A)
    return TConcat(*PC)
```

Ejemplo 6.18

Utiliza el programa `pCart` para calcular $\{\texttt{ana}, \texttt{mar}, \texttt{fer}\} \times \{\texttt{fresa}, \texttt{chocolate}\}$.

```
>>> pCart(['ana','mar','fer'], ['fresa','chocolate'])
[['ana', 'fresa'], ['ana', 'chocolate'], ['mar', 'fresa'], ['mar', 'chocolate'],
['fer', 'fresa'], ['fer', 'chocolate']]
>>>
```

6.5.2 Cardinalidad del producto cartesiano

En la tabla anterior podemos observar que si $|A| = n$ y $|B| = m$, se forman m pares por cada uno de los n elementos de A, esto significa que

$$|A \times B| = |A| \cdot |B| = nm$$

Eso si, $|A \times B| = |B \times A|$, porque ahora solamente consideramos el producto natural de las cardinalidades $|A \times B| = |A| \cdot |B| = |B| \cdot |A| = |B \times A|$

Ejemplo 6.19

¿Cuántos pares se pueden lograr con el producto cartesiano de un conjunto $A = \{x, y, z\}$ con otro conjunto $B = \{1, 2, 6, 3, 8\}$? Como $|A| = 3$ y $|B| = 5$, $|A \times B| = 3 \cdot 5 = 15$. El producto cartesiano es el siguiente:

$A \times B$	1	2	6	3	8
x	$\langle x,1 \rangle$	$\langle x,2 \rangle$	$\langle x,6 \rangle$	$\langle x,3 \rangle$	$\langle x,8 \rangle$
y	$\langle y,1 \rangle$	$\langle y,2 \rangle$	$\langle y,6 \rangle$	$\langle y,3 \rangle$	$\langle y,8 \rangle$
z	$\langle z,1 \rangle$	$\langle z,2 \rangle$	$\langle z,6 \rangle$	$\langle z,3 \rangle$	$\langle z,8 \rangle$

```
>>> A = conj('x', 'y', 'z')
>>> B = conj(1,2,6,3,8)
>>> AxB = pCart(A, B)
>>> card(AxB)
15
>>>
```

¿Qué pasaría si $A = \varnothing$ o $B = \varnothing$? Supongamos que uno de ellos es vacío, digamos A. De acuerdo con lo anterior $|A \times B| = |A| \cdot |B| = 0 \cdot m = 0$, si B fuera el conjunto vacío se obtiene también que $|A \times B| = 0$. La conclusión es que si $|A \times B| = 0$ significa que alguno o ambos conjuntos es vacío; también, si $|A \times B| = 0$ entonces $A \times B = \varnothing$ es una proposición verdadera.

Ejemplo 6.20

Si $A = \varnothing$ y $B = \{w, x, y, z\}$, $|A \times B| = 0 \cdot 4 = 0$. Así $A \times B = \varnothing$.

```
>>> A = conj()
>>> B = conj('w','x','y','z')
>>> AxB = pCart(A,B)
>>> card(AxB)
0
>>> AxB
[]
>>>
```

El producto cartesiano de un conjunto A consigo mismo, se puede escribir $A \times A$ claro, pero también se escribe A^2. Esta notación que utiliza potencias es frecuentemente usada en varias aplicaciones, entre ellas cuando se representa el plano cartesiano bidimensional $\mathbb{Z}^2$; también se usa con el conjunto de números reales $\mathbb{R}$, donde el plano cartesiano se escribe como $\mathbb{R}^2$ lo que significa $\mathbb{R} \times \mathbb{R}$.

6.5.3 Extensión del producto cartesiano

La noción de producto cartesiano de dos conjuntos se puede extender al considerar cualquier número de conjuntos no vacíos. La idea es simple:

1. El producto cartesiano aplicado a ningún conjunto es el conjunto vacío.
2. El producto cartesiano de un conjunto, es el conjunto de tuplas unitarias generadas con cada uno de los elementos de ese conjunto.
3. Con $n \geq 1$ conjuntos no vacíos, el producto cartesiano se conforma de tuplas de longitud n, generadas con elementos de cada conjunto:

$$\begin{aligned} \prod_{i=1}^{n} A_i = & \quad A_1 \times A_2 \times \cdots \times A_n \\ = & \quad \{\langle a_1, a_2, \ldots, a_n \rangle | a_1 \in A_1 \wedge a_2 \in A_2 \wedge \cdots \wedge a_n \in A_n\} \end{aligned}$$

Ejemplo 6.21

Para los siguientes ejemplos considera los conjuntos $A_1 \leftarrow \{a, b, c\}$, $A_2 \leftarrow \{4, 5\}$ y $A_3 \leftarrow \{x, y, z\}$ para calcular $\prod_{i=1}^{\ell} A_i$, con $\ell = 1, 2, 3$:

1. $\prod_{2} A_1 \mapsto \{\langle a \rangle, \langle b \rangle, \langle c \rangle\}$.
2. $$\begin{aligned} \prod_{i=1} A_i \quad & = \{\langle a_1, a_2 \rangle | a_1 \in A_1 \wedge a_2 \in A_2\} \\ & \mapsto \{\langle a, 4 \rangle, \langle a, 5 \rangle, \langle b, 4 \rangle, \langle b, 5 \rangle, \langle c, 4 \rangle, \langle c, 5 \rangle\} \end{aligned}$$
3. $$\begin{aligned} \prod_{i=1}^{3} A_i \quad & = \{\langle a_1, a_2, a_3 \rangle | a_1 \in A_1 \wedge a_2 \in A_2 \wedge a_3 \in A_3\} \\ & \mapsto \{\langle a, 4, x \rangle, \langle a, 4, y \rangle, \langle a, 4, z \rangle, \langle a, 5, x \rangle, \langle a, 5, y \rangle, \langle a, 5, z \rangle, \\ & \quad \langle b, 4, x \rangle, \langle b, 4, y \rangle, \langle b, 4, z \rangle, \langle b, 5, x \rangle, \langle b, 5, y \rangle, \langle b, 5, z \rangle, \\ & \quad \langle c, 4, x \rangle, \langle c, 4, y \rangle, \langle c, 4, z \rangle, \langle c, 5, x \rangle, \langle c, 5, y \rangle, \langle c, 5, z \rangle\} \end{aligned}$$

Ejemplo 6.22

Si $L = \{\texttt{AM}, \texttt{4O}, \texttt{M7}, \texttt{Y4*}\}$ es el conjunto de líneas aéreas, y por otro lado $A = \{\texttt{AGU}, \texttt{ESE}, \texttt{CDI}\}$ son un conjunto de ciudades. El conjunto que modela el concepto de que una línea aérea vuela de una ciudad a otra, se puede establecer mediante 3–tuplas $\langle l, o, d \rangle$, donde l es una línea aérea, o es una

ciudad de origen tomada del conjunto A y d es una ciudad de destino tomada del mismo conjunto A, lo que se modela mediante el producto cartesiano

$$\begin{aligned} L \times A \times A &= \{\texttt{AM}, \texttt{4O}, \texttt{M7}, \texttt{Y4*}\} \times \{\texttt{AGU}, \texttt{ESE}, \texttt{CDI}\} \times \{\texttt{AGU}, \texttt{ESE}, \texttt{CDI}\} \\ &\mapsto \{\langle \texttt{AM},\texttt{AGU},\texttt{AGU}\rangle, \langle \texttt{AM},\texttt{AGU},\texttt{ESE}\rangle, \langle \texttt{AM},\texttt{AGU},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{AM},\texttt{ESE},\texttt{AGU}\rangle, \langle \texttt{AM},\texttt{ESE},\texttt{ESE}\rangle, \langle \texttt{AM},\texttt{ESE},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{AM},\texttt{CDI},\texttt{AGU}\rangle, \langle \texttt{AM},\texttt{CDI},\texttt{ESE}\rangle, \langle \texttt{AM},\texttt{CDI},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{4O},\texttt{AGU},\texttt{AGU}\rangle, \langle \texttt{4O},\texttt{AGU},\texttt{ESE}\rangle, \langle \texttt{4O},\texttt{AGU},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{4O},\texttt{ESE},\texttt{AGU}\rangle, \langle \texttt{4O},\texttt{ESE},\texttt{ESE}\rangle, \langle \texttt{4O},\texttt{ESE},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{4O},\texttt{CDI},\texttt{AGU}\rangle, \langle \texttt{4O},\texttt{CDI},\texttt{ESE}\rangle, \langle \texttt{4O},\texttt{CDI},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{M7},\texttt{AGU},\texttt{AGU}\rangle, \langle \texttt{M7},\texttt{AGU},\texttt{ESE}\rangle, \langle \texttt{M7},\texttt{AGU},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{M7},\texttt{ESE},\texttt{AGU}\rangle, \langle \texttt{M7},\texttt{ESE},\texttt{ESE}\rangle, \langle \texttt{M7},\texttt{ESE},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{M7},\texttt{CDI},\texttt{AGU}\rangle, \langle \texttt{M7},\texttt{CDI},\texttt{ESE}\rangle, \langle \texttt{M7},\texttt{CDI},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{Y4*},\texttt{AGU},\texttt{AGU}\rangle, \langle \texttt{Y4*},\texttt{AGU},\texttt{ESE}\rangle, \langle \texttt{Y4*},\texttt{AGU},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{Y4*},\texttt{ESE},\texttt{AGU}\rangle, \langle \texttt{Y4*},\texttt{ESE},\texttt{ESE}\rangle, \langle \texttt{Y4*},\texttt{ESE},\texttt{CDI}\rangle, \\ &\quad \langle \texttt{Y4*},\texttt{CDI},\texttt{AGU}\rangle, \langle \texttt{Y4*},\texttt{CDI},\texttt{ESE}\rangle, \langle \texttt{Y4*},\texttt{CDI},\texttt{CDI}\rangle\} \end{aligned}$$

La cantidad de tuplas obtenidas se calcula mediante $|L \times A \times A|$ se obtiene del producto $|L| \cdot |A| \cdot |A|$, que es $4 \cdot 3 \cdot 3 \mapsto 36$.

$$\texttt{PCart}(\{C_0|C'\}) \mapsto \begin{cases} \varnothing & \text{si } C = \varnothing \\ C_0 & \text{si } |C| = 1 \wedge \forall(\lambda e \in C \bullet \texttt{esTupla}(e)) \\ \texttt{enCada}(\lambda t \in C_0 \bullet \texttt{tupla}(t)) & \text{si } |C| = 1. \\ \texttt{PCart}\left(\texttt{pCart}\left(C_0, C_0'\right) \cup (C')'\right) & \text{si } C' = \{C_0'|(C')'\} \end{cases}$$

Ejemplo 6.23

Los siguientes ejemplos ilustran cada caso del algoritmo `PCart`:

1. $\texttt{PCart}(\,) \mapsto \varnothing$, porque $C = \varnothing$.
2. $\texttt{PCart}(\{\langle 1\rangle, \langle 2\rangle\}) \mapsto \{\langle 1\rangle, \langle 2\rangle\}$, porque $|C| = 1 \wedge \forall(\lambda e \in C \bullet \texttt{esTupla}(e))$.
3. $\texttt{PCart}(\{1,2\}) \mapsto \{\langle 1\rangle, \langle 2\rangle\}$, porque $|C| = 1$.
4. $$\begin{aligned} \texttt{PCart}(\{1,2\}, \{x,y\}) &= \texttt{PCart}((\{1,2\} \times \{x,y\}) \cup \varnothing) \quad \texttt{\# Caso 4} \\ &= \texttt{PCart}(\{\langle 1,x\rangle, \langle 1,y\rangle, \langle 2,x\rangle, \langle 2,y\rangle\}) \\ &\mapsto \{\langle 1,x\rangle, \langle 1,y\rangle, \langle 2,x\rangle, \langle 2,y\rangle\} \quad \texttt{\# Caso 2} \end{aligned}$$

6.6 Relaciones binarias

6.6.1 Definición de relación binaria

Definición 6.6.1 -- Relación binaria. Sean A y B dos conjuntos no vacíos, una relación binaria R de elementos de A con elementos de B se escribe $R : A \to B$, es un conjunto de pares ordenados $\langle a, b\rangle$, con $a \in A$ y $b \in B$. Formalmente es definida como $R : A \to B \subseteq A \times B$.

La expresión $R : A \to B$ se conoce como la **firma** de la relación y es un identificador, no una operación. En $R : A \to B$, R es el nombre de la relación; $A \to B$ establece que elementos del conjunto A [posiblemente todos o quizá ninguno], se relacionan con elementos de del conjunto B [posiblemente todos o quizá ninguno].

Cuando en una relación $R : A \to B$ ocurre que $A = B$, es decir $R : A \to A$, podemos decir que la relación está definida «sobre» el conjunto A. Cuando no hay confusión acerca de los conjuntos A y B que son considerados, se puede escribir simplemente R y omitir el resto de la notación de la relación $R : A \to B$.

Como $R : A \to B \subseteq A \times B$, entonces $|R : A \to B| \leq |A \times B|$.

La importancia de las relaciones binarias como subconjuntos del producto cartesiano, se debe a la interpretación que se les da a los pares considerados.

Ejemplo 6.24

La relación «es hijo de» establece una relación binaria entre dos conjuntos de personas. Si $A = \{\texttt{pedro}, \texttt{juana}, \texttt{maricela}, \texttt{antonio}\}$ y $B = \{\texttt{gustavo}, \texttt{brenda}, \texttt{pablo}\}$, la relación $R : A \to B$ que representa la relación *"a es hijo de b"* contiene pares de la forma $\langle a, b \rangle$, donde $a \in A$ y $b \in B$, la relación R es definida extensionalmente como

$$R \leftarrow \{\langle \texttt{pedro}, \texttt{gustavo} \rangle, \langle \texttt{maricela}, \texttt{pablo} \rangle, \langle \texttt{juana}, \texttt{brenda} \rangle\}$$

Donde cada par $\langle a, b \rangle \in R$ debe ser interpretado como «a es hijo de b» y establece que por ejemplo, pedro es hijo de gustavo ya que $\{\texttt{pedro}, \texttt{gustavo}\} \in R$; pero no se puede interpretar que gustavo es hijo de pedro porque $\langle \texttt{gustavo}, \texttt{pedro} \rangle \notin R$.

En una relación R, si $\langle a, b \rangle \in R$ también se puede escribir aRb; si por el contrario $\langle a, b \rangle \notin R$, se puede escribir $a \not R b$.

Del mismo modo que los conjuntos, las relaciones pueden ser definidas de manera extensional o de manera intencional.

Una relación $R : A \to B$ definida de manera extensional es una lista explícita de pares ordenados, como en

$$R \leftarrow \{\langle 1, 3 \rangle, \langle 1, 6 \rangle, \langle 1, 4 \rangle, \langle 3, 4 \rangle, \langle 3, 6 \rangle, \langle 4, 6 \rangle\},$$

en este tipo de relaciones, a menos que se indique otra cosa, el conjunto A se supone que se conforma de todas las primeras entradas de las tuplas, el conjunto B se compone de todas las segundas entradas.

Cuando la relación $R : A \to B$ se define de manera intencional, se deben definir el conjunto $A \times B$ de donde serán tomados los pares y también un predicado que permita seleccionar los pares que pertenecen a la relación, por ejemplo:

$$S \leftarrow \{\langle x, y \rangle \in \{1, 3, 4, 6\}^2 \mid x < y\},$$

la relación anterior establece que un par $\langle x, y \rangle$ tendrá como primera entrada un elemento de $\{1, 3, 4, 6\}$ y como segunda entrada aquel elemento de $\{1, 3, 4, 6\}$ que sea mayor que su primera entrada, así que S es la relación

$$S \leftarrow \{\langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 1, 6 \rangle, \langle 3, 4 \rangle, \langle 3, 6 \rangle, \langle 4, 6 \rangle\}.$$

Para verificar que un conjunto de pares ordenados es una relación R que asocia elementos de un conjunto A con elementos de otro conjunto B, podemos verificar que los pares de la relación sean un subconjunto del producto cartesiano $A \times B$.

Código 6.13: *Verifica que una lista de pares sea una relación*

```
1  def esRel(R:list, A:list, B:list)-> list:
2      """
3      Determina si una lista de pares es una relación válida.
4      """
5      return esSubc(R, pCart(A, B))
```

Ejemplo 6.25

Utiliza la herramienta `esRel` para determinar si $R \hookleftarrow \{\langle 1,3\rangle, \langle 1,4\rangle, \langle 1,6\rangle, \langle 3,4\rangle, \langle 3,6\rangle, \langle 4,6\rangle\}$ es una relación en $\{1,2,3,4,5,6\}^2$.

```
>>> A = conj(1,2,3,4,5,6)
>>> AxA = pCart(A,A)
>>> R = conj(tupla(1,3), tupla(1,4), tupla(1,6), tupla(3,4), tupla(3,6), tupla(4,6))
>>> esRel(R, A, A)
True
>>>
```

6.6.2 Partes de una relación

Una relación se conforma de las siguientes partes [ver figura 6.1]:

1. El **identificador**, en el caso de $R : A \to B$ el identificador o el nombre de la relación es R. Otro ejemplo es la relación $\texttt{esPadreDe} : A \to B$, aquí el nombre de la relación es `esPadreDe`.
2. El **conjunto de referencia** de la relación. En una relación $R : A \to B$, el conjunto de referencia es el conjunto A, que contiene los valores que podrían ser relacionados, aunque es posible que no todos participen en la relación.
3. El **dominio** de la relación R, denotado $\texttt{dom}(R)$, es el subconjunto del conjunto de referencia definido por $\{x \in A | \exists b \in B : \langle x, b\rangle \in R\}$. Así $\texttt{dom}(R) \subseteq A$.
4. El **codominio** de la relación $R : A \to B$, es el conjunto B, que contiene todos los posibles valores que se pueden relacionar con elementos del dominio.
5. El **rango** de la relación $R : A \to B$ es denotado $\texttt{ran}(R)$ y es un subconjunto de B, precisamente el conjunto $\{y \in B | \exists a \in A : \langle a, y\rangle \in R\}$. Así $\texttt{ran}(R) \subseteq \texttt{cod}(R)$.

En una relación $R : A \to B$ dada de forma extensional, es decir, enlistando todos y cada uno de los pares de la relación, el dominio de la relación coincide con el conjunto de referencia, que es el conjunto que contiene a todos los primeros elementos de cada par. El codominio por su parte, coincide con el rango, que es el conjunto de los segundos elementos de cada par.

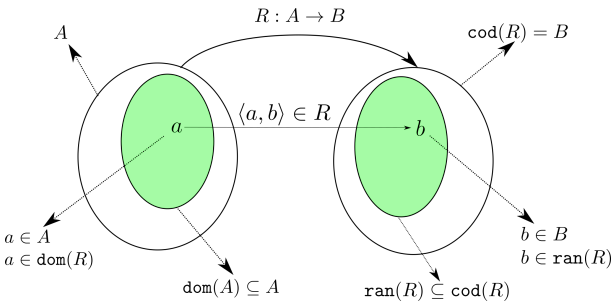


Figura 6.1: *Partes de una relación*

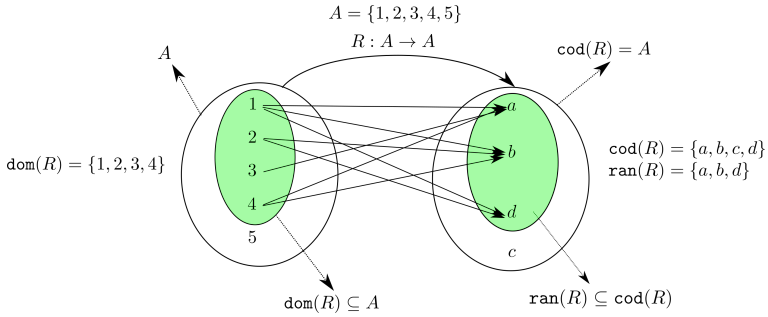
Ejemplo 6.26

Sean $A = \{1,2,3,4,5\}$, $B = \{a,b,c,d\}$ dos conjuntos y la relación $R \hookleftarrow \{\langle 1,a\rangle, \langle 1,b\rangle, \langle 1,d\rangle, \langle 2,b\rangle, \langle 2,d\rangle, \langle 3,a\rangle, \langle 4,a\rangle, \langle 4,b\rangle\}$

$\texttt{dom}(R) = \{1,2,3,4\}$

$\texttt{ran}(R) = \{a,b,d\}$

$\texttt{cod}(R) = B$



Código 6.14: *Dominio y rango de una relación*

```
def dom(R:list)-> list:
    """
    Obtiene el dominio de una relación.
    """
    l = tLong(car(R))
    D = conj(*enCada(lambda t: car(tKdivide(t, l-1)), R))
    D = [car(x) if esUnit(x) else x for x in D]
    return D


def ran(R:list)-> list:
    """
    Obtiene el rango de una relación.
    """
    l = tLong(car(R))
    C = conj(*enCada(lambda t: car(car(cdr(tKdivide(t, l-1))))), R))
    return C
```

6.6.3 Representaciones de las relaciones binarias

Las relaciones binarias pueden ser representadas de diferentes maneras, aquí te muestro algunas. Para ilustrarlas supongamos los siguientes conjuntos $A \leftarrow\!\shortmid \{1,2,3,4\}$, $B \leftarrow\!\shortmid \{a,b,c,d\}$ y una relación $R : A \to B$ definida de manera extensional como la relación $R \leftarrow\!\shortmid \{\langle 1,a\rangle, \langle 1,b\rangle, \langle 1,d\rangle, \langle 2,b\rangle, \langle 2,d\rangle, \langle 3,a\rangle, \langle 4,a\rangle, \langle 4,b\rangle\}$.

1. En forma de una **lista explícita** de pares, como

$$\{\langle 1,a\rangle, \langle 1,b\rangle, \langle 1,d\rangle, \langle 2,b\rangle, \langle 2,d\rangle, \langle 3,a\rangle, \langle 4,a\rangle, \langle 4,b\rangle\}$$

esta misma lista se escribe en `Python` usando las definiciones `conj` [código 3.2, página 71] y `tupla` [definición 6.1.1, página 121] como:

```
conj(tupla(1,'a'), tupla(1,'b'), tupla(1,'d'), tupla(2,'b'), tupla(2,'d'),
       tupla(3,'a'), tupla(4,'a'), tupla(4,'b'))
```

En esta representación, cada par se construye con un elemento del conjunto A y un elemento del conjunto B de la relación. Es la manera más explícita de indicar qué elemento del dominio está relacionado con qué elemento del rango. Estrictamente hablando, una relación R descrita de este modo tiene una firma

$$R : \texttt{dom}(R) \to \texttt{ran}(R).$$

Una desventaja de esta representación es la necesidad de escribir explícitamente cada par de la relación, cuando hay otras maneras de resumir un poco la notación. Sin embargo esta es la representación más apegada a la definición matemática.
Otra desventaja es que en una relación R definida sobre los conjuntos A y B, pueden haber elementos de A que no aparecen en el dominio y elementos de B que no aparecen en el rango de la relación.
Por ejemplo, la relación $R : \{1,2,3,4,5\} \to \{1,2,3,4,5\}$, donde

$$R \leftarrow\!\shortmid \{\langle x, y\rangle \in \{1,2,3,4,5\}^2 \mid y \leftarrow\!\shortmid 2 \text{ si } \texttt{par}(x) \ \textit{eoc}\ y \leftarrow\!\shortmid 1\}.$$

Donde se tienen los siguientes pares:

$$R \leftarrow\!\shortmid \{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,1\rangle, \langle 4,2\rangle, \langle 5,1\rangle\}$$

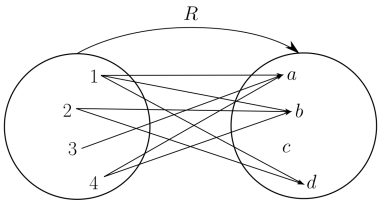
que tiene $\texttt{dom}(R) \mapsto \{1,2,3,4,5\}$ pero $\texttt{ran}(R) \mapsto \{1,2\}$.

2. En forma de **matriz de pertenencia**. En esta manera, con la información de la relación $R : A \to B$, donde $A \leftarrow\!\shortmid \{a_1, \ldots, a_i, \ldots, a_n\}$ y $B \leftarrow\!\shortmid \{b_1, \ldots, b_j, \ldots, b_m\}$ se crea una matriz de orden $n \times m$, donde cada renglón de la matriz, contiene información de elementos del conjunto A y cada columna informa sobre elementos del conjunto B. Así, cada entrada $\langle i, j\rangle$ en la matriz, con $1 \leq i \leq |A|$ y $1 \leq j \leq |B|$, es un valor 1 o 0 y representa la pertenencia de un par ordenado de la relación. En la entrada $\langle i, j\rangle$ de la matriz se coloca un 1 si el par $\langle a_i, b_j\rangle$ pertenece a la relación y un 0 si ese par no pertenece a la relación. Para una mejor visualización, frecuentemente se coloca un $\bullet$ en lugar de un 0.

$$R \leftarrow\!\shortmid \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 1 & 1 & \bullet & 1 \\ \bullet & 1 & \bullet & 1 \\ 1 & \bullet & \bullet & \bullet \\ 1 & 1 & \bullet & \bullet \end{pmatrix} \end{array}$$

El dominio de la relación está conformado por los elementos de A que aportan un renglón no-cero [al menos un 1 en el renglón]; y el rango de la relación lo constituyen aquellos elementos de B que aportan una columna no-cero en la matriz.
Un inconveniente de esta forma de representación es que puede generar matrices de gran tamaño pero con pocos unos, es decir, relaciones con conjuntos A y B muy grandes, pero con $\texttt{dom}(R)$ y $\texttt{ran}(R)$ muy pequeños, ocasionando matrices con muchos ceros y pocos unos, ocupando espacio en memoria y tiempo de procesamiento innecesarios.

3. En forma de **diagramas sagitales**. Los diagramas sagitales se utilizan cuando las relaciones son más bien pequeñas y se pueden dibujar en una hoja. La manera es colocar en un círculo a la izquierda los elementos del dominio de la relación, a la derecha en otro círculo los elementos del codominio, y uniendo con una flecha un elemento del dominio con un elemento del codominio. El nombre de la relación encabeza una flecha que sale del conjunto de la izquierda y llega al conjunto de la derecha.
En esta representación el conjunto dibujado a la izquierda es el dominio de la relación, mientras que el de la derecha es el rango.



El inconveniente es que al dibujar las líneas que representan las relaciones, se van cruzando unas con otras, generando confusión visual.

4. En forma de **listas de relaciones**. Aquí se crea una lista de listas. Cada lista interior contiene en primer lugar un elemento del dominio, y en seguida todos los elementos del rango de la relación con los que el primer elemento tiene relación.

$$\langle\langle 1,a,b,d\rangle,\langle 2,b,d\rangle,\langle 3,a\rangle,\langle 4,a,b\rangle\rangle$$

En `Python` se ve también como una lista de listas:

```
[[1,'a','b','d'], [2,'b','d'], [3,'a'], [4,'a','b']]
```

Este formato es bastante útil, porque se puede escribir un archivo de texto sin formato y escribir en cada línea una de estas listas, por ejemplo la relación R se escribe en un archivo de texto llamado [por ejemplo] `R.dat`:

R.dat
```
1 a b d
2 b d
3 a
4 a b
```

Luego mediante la ayuda de un programa que tome el archivo `R.dat`, la información se convierte a una lista de listas o en una lista de pares o una matriz de pertenencia, todo depende de cómo sea más conveniente.

6.7 Imagen de una relación

Definición 6.7.1 -- Imagen de un elemento. La imagen de un elemento a en el dominio de una relación $R : A \to B$, está definido como

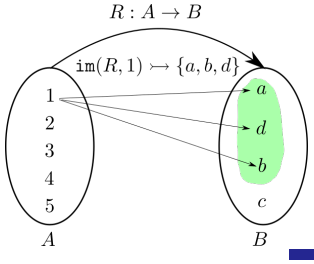
$$\texttt{im}(R,a) = \{b \in \texttt{ran}(R) | aRb\}$$

La imagen de un elemento del dominio siempre es un conjunto, posiblemente $\varnothing$.

Ejemplo 6.27

Supongamos que $A \leftarrowtail \{1,2,3,4,5\}$, $B \leftarrowtail \{a,b,c,d\}$ son conjuntos y una relación $R \leftarrowtail \{\langle 1,a\rangle,\langle 1,b\rangle,\langle 1,d\rangle,\langle 2,b\rangle,\langle 2,d\rangle,\langle 3,a\rangle,\langle 4,a\rangle,\langle 4,b\rangle\}$.

$\texttt{im}(R,1) = \{a,b,d\}$, *mostrado en la figura.*
$\texttt{im}(R,2) = \{b,d\}$.
$\texttt{im}(R,3) = \{a\}$.
$\texttt{im}(R,4) = \{a,b\}$.
$\texttt{im}(R,5) = \varnothing$.



Para escribir un programa en Python que obtenga la imagen de un elemento a del dominio de una relación R, seguiremos el siguiente algoritmo:

1. Tomaremos l como la longitud de las tuplas, se espera que todas las tuplas sean de la misma longitud, pues R es una lista de pares ordenados.
2. Calcularemos un conjunto de imágenes C, que se obtiene de tomar cada tupla en donde aparezca a como primer elemento.
3. De cada tupla en C, obtener cada último elemento y hacer un conjunto.

Código 6.15: *Imagen de un elemento del dominio*

```
1 def im(R:list, a)-> list:
2     """
3     Calcula la imagen de un elemento del dominio de la relación.
4     """
5     l = tLong(car(R))-1
6     if neg(esTupla(a)): a = tupla(a)
7     C = subc(lambda t: a == car(tKdivide(t, l)), R)
8     D = conj(*enCada(lambda t: car(car(cdr(tKdivide(t, l)))), C))
9     return D
```

Nota: las líneas 6 y 7 del programa, tendrán sentido después del tema de relaciones n-arias [página 143].

Ejemplo 6.28

Usa la función im para calcular la imagen de cada elemento del dominio en el ejemplo 6.27.

```
>>> R = [[1,'a'], [1,'b'], [1,'d'],[2,'b'],[2,'d'],[3,'a'],[4,'a'],[4,'b']]
>>> im(R,1)
['a', 'b', 'd']
>>> im(R,2)
['b', 'd']
>>> im(R,3)
['a']
>>> im(R,4)
['a', 'b']
>>> im(R,5)
[]
>>>
```

La imagen de un elemento del dominio es un concepto que también se puede extender a la **imagen de un conjunto de elementos** del siguiente modo:

Definición 6.7.2 -- Imagen de un conjunto. Sea $R : A \to B$ una relación y $D \hookleftarrow \{d_1, d_2, \ldots, d_k\}$ un conjunto donde $D \subseteq A$, la imagen $\mathtt{Im}(R, D)$ se define como

$$\begin{aligned} \mathtt{Im}(R, D) &= \mathtt{Im}(R, \{d_1, d_2, \ldots, d_k\}) \\ &= \mathtt{im}(R, d_1) \cup \mathtt{im}(R, d_2) \cup \cdots \cup \mathtt{im}(R, d_k) \\ &\mapsto \bigcup_{j=1}^{k} \mathtt{im}\left(R, d_j\right) \end{aligned}$$

Ejemplo 6.29

Considera la relación $S : C \to C$ definida sobre el conjunto $C \hookleftarrow \{1,2,3,4,5,6,7,8,9\}$ siguiente:

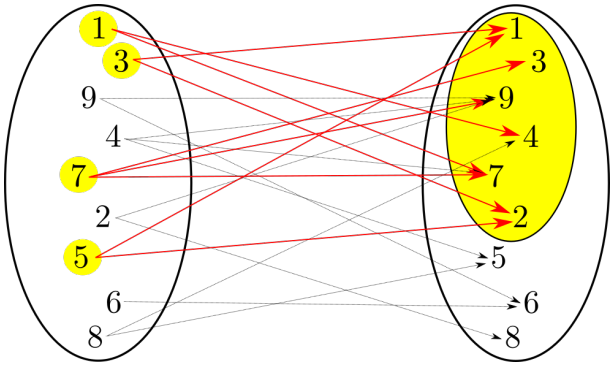
$$\begin{aligned} S \quad \hookleftarrow \{&\langle 1,4\rangle, \langle 1,7\rangle, \langle 2,8\rangle, \langle 2,9\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,5\rangle, \langle 4,7\rangle, \langle 4,9\rangle, \langle 5,1\rangle, \langle 5,2\rangle, \langle 6,6\rangle, \\ &\langle 7,3\rangle, \langle 7,7\rangle, \langle 7,9\rangle, \langle 8,4\rangle, \langle 8,5\rangle, \langle 9,6\rangle, \langle 9,9\rangle\} \end{aligned}$$

Para el objetivo de este ejemplo, es mejor representar la relación S como la siguiente lista de relaciones, esto es porque un elemento e del dominio de la relación encabeza una lista que tiene como resto, justamente la imagen de e:

$$S \leftarrow\!\!\!\shortmid \langle\langle 1,4,7\rangle,\langle 2,8,9\rangle,\langle 3,1,2\rangle,\langle 4,5,7,9\rangle,\langle 5,1,2\rangle,\langle 6,6\rangle,\langle 7,3,7,9\rangle,\langle 8,4,5\rangle,\langle 9,6,9\rangle\rangle$$

La imagen de $\{1,3,5,7\} \subset \mathtt{dom}(S)$ es:

$$\begin{aligned}\mathtt{Im}(S,\{1,3,5,7\}) &= \bigcup_{e\in\{1,3,5,7\}} \mathtt{im}(S,e)\\ &= \mathtt{im}(S,1)\cup\mathtt{im}(S,3)\cup\mathtt{im}(S,5)\cup\mathtt{im}(S,7)\\ &= \{4,7\}\cup\{1,2\}\cup\{1,2\}\cup\{3,7,9\}\\ &\mapsto \{4,7,1,2,3,9\}\end{aligned}$$



Código 6.16: *Imagen de un subconjunto de elementos del dominio*

```
def Im(R:list, A:list)-> list:
    """
    Calcula la imagen de un conjunto de elementos del dominio.
    """
    return Union(*enCada(lambda e:im(R, e), A))
```

Ejemplo 6.30

Considera la relación $R \leftarrow\!\!\!\shortmid \{\langle 5,17\rangle,\langle 4,6\rangle,\langle 6,7\rangle,\langle 15,13\rangle,\langle 8,2\rangle,\langle 2,19\rangle,\langle 13,6\rangle,\langle 15,0\rangle,\langle 18,10\rangle,\langle 6,7\rangle,\langle 5,18\rangle,\langle 1,15\rangle,\langle 3,13\rangle,\langle 7,16\rangle,\langle 2,3\rangle,\langle 12,1\rangle,\langle 5,14\rangle,\langle 16,4\rangle,\langle 18,1\rangle,\langle 1,18\rangle,\langle 2,11\rangle,\langle 18,7\rangle\}$

Donde:

$\mathtt{dom}(R) \mapsto \{4,8,13,15,6,3,7,12,5,16,1,2,18\}$

$\mathtt{ran}(R) \mapsto \{17,2,19,6,0,10,15,13,16,3,14,4,1,18,11,7\}$

Utiliza la función `Im` para calcular la imagen del conjunto $\{4,18,12,9,3\}$.

```
>>> R = [5,17],[4,6],[6,7],[15,13],[8,2],[2,19],[13,6],[15,0],[18,10],
[6,7],[5,18],[1,15],[3,13],[7,16],[2,3],[12,1],[5,14],[16,4],[18,1],[1,18],[2,11],[18,7]
>>> C = conj(4,12,18,9,3)
>>> Im(R,C)
[10, 1, 7, 13, 6]
>>>
```

6.8 Relaciones *n*-arias

Las relaciones n-arias son una extensión natural de las relaciones binarias que se llaman «binarias» porque sus elementos son elementos del producto cartesiano de dos conjuntos. Tomando esto en cuenta definimos una relación n-aria como:

Definición 6.8.1 -- Relación n-aria. Sean $A_1, A_2, \ldots, A_n$ conjuntos no vacíos con $n \geq 2$. Una **relación n-aria** sobre estos conjuntos es un subconjunto de $A_1 \times \ldots \times A_n$ y es definida como

$$\left\{\langle a_1, \ldots, a_n \rangle \in \prod_{i=1}^{n} A_i | a_1 \in A_1, \ldots, a_n \in A_n\right\}.$$

Cada elemento de la relación es una n-tupla de la forma $\langle n_1, \ldots, n_a \rangle$. La firma de la relación n-aria se escribe $R : A_1 \times \ldots \times A_{n-1} \to A_n$, donde R es el nombre de la relación; el dominio es subconjunto del producto cartesiano de tales conjuntos; y el rango es subconjunto de A_n. Cada n-tupla en la relación, contiene información del elemento del dominio y el elemento del rango que están siendo relacionados. Por conveniencia, la relación n-aria se puede ver como una relación binaria que contiene pares ordenados de la forma

$$\langle \langle a_1, \ldots, a_{n-1} \rangle, a_n \rangle,$$

donde cada tupla $\langle a_1, a_2, \ldots, a_{n-1} \rangle$ es un elemento del domino de la relación y a_n pertenece al rango de la relación, sin embargo, para facilitar la lectura se ha convenido en escribir $\langle a_1, \ldots, a_{n-1}, a_n \rangle$, obviando que la tupla $\langle a_1, \ldots, a_{n-1} \rangle$ es un elemento del dominio y a_n es un elemento del rango de la relación.

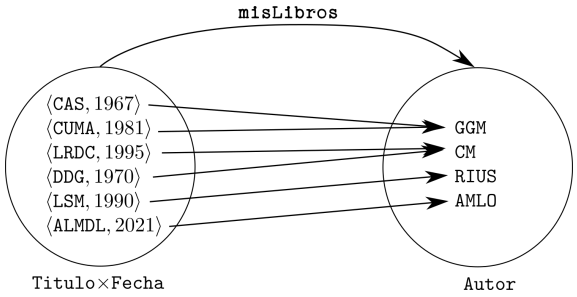
Ejemplo 6.31

Considera la relación 3-aria que se ilustra en la siguiente tabla de libros:

Título	**Fecha**	**Autor**
Cien años de soledad [CAS]	1967	Gabriel García Márquez [GGM]
Crónica de una muerte anunciada [CUMA]	1981	Gabriel García Márquez [GGM]
Los rituales del caos [LRDC]	1995	Carlos Monsiváis [CM]
Días de guardar [DDG]	1970	Carlos Monsiváis [CM]
Los super machos [LSM]	1990	Eduardo del Rio [RIUS]
A la mitad del camino [ALMDC]	2021	Andrés Manuel López Obrador [AMLO]

Cada renglón de la tabla representa una 3-tupla en la relación que arbitrariamente podemos llamar `misLibros` y establece con la firma $\texttt{misLibros} : \texttt{Titulo} \times \texttt{Fecha} \to \texttt{Autor}$ que está definida explícitamente con las siguientes tuplas:

$$\texttt{misLibros} \hookleftarrow \Big\{\langle \texttt{CAS}, 1967, \texttt{GGM}\rangle \langle \texttt{CUMA}, 1981, \texttt{GGM}\rangle, \langle \texttt{LRDC}, 1995, \texttt{CM}\rangle, \langle \texttt{DDG}, 1970, \texttt{CM}\rangle, \langle \texttt{LSM}, 1990, \texttt{RIUS}\rangle, \langle \texttt{ALMDC}, 2021, \texttt{AMLO}\rangle\Big\}$$



6.8.1 Imagen de un elemento del dominio en relaciones *n*-arias

La generalización de la imagen de un elemento del dominio tiene en esencia los mismos elementos que en la definición para el caso binario [definición 6.7.1, página 140], pero ahora se considera como un elemento a la tupla que contiene los primeros $n-1$ elementos, siendo el último de la tupla el elemento del rango que es el que pertenecerá a la imagen.

Considerando una relación n-aria $R : A_1 \times \cdots \times A_{n-1} \to A_n$:

$$\texttt{im}(R, \langle a_1, \ldots, a_{n-1} \rangle) = \{a_n \in A_n | \langle a_1, \ldots, a_{n-1}, a_n \rangle \in R\}$$

La implementación encierra unos retos porque la función `im` que calcula la imagen de un elemento del dominio de una relación, requiere la lista R de las tuplas de la relación y el elemento del dominio a del cual se desea calcular su imagen que bien puede ser una n-tupla o no.

El algoritmo es como sigue:

```
im(R, a)
    Requiere:
    R : conjunto de n-tuplas
    a : un elemento del dominio de R
    Devuelve:
    un conjunto

    C ↤ Conjunto de las tuplas en R cuyo primer elemento es a
    D ↤ Conjunto con las últimas entradas de cada tupla en C
    devolver D
```

Para determinar el primer elemento de una tupla, tomaremos ℓ como la longitud de la tupla menos 1, que es la longitud de la tupla en el dominio de la relación. Se utilizarán las herramientas de tuplas creadas al inicio de este capítulo, en particular `tKdivide` [código 6.9, página 129] que toma una tupla t y un número k y devuelve un par ordenado con dos tuplas, una de ellas contiene los primeros k elementos de la tupla original y la otra tupla contiene el resto de los elementos. Así `tKdivide`(t, ℓ) divide la tupla t en dos partes, la primera que contiene el elemento del dominio y la segunda un elemento del rango de la relación, por ejemplo `tKDivide`$(\langle 1,2,3,4 \rangle, 3) \mapsto \langle \langle 1,2,3 \rangle, \langle 4 \rangle \rangle$.

Para obtener el elemento del dominio de una tupla t, tomaremos la primera tupla de la respuesta generada por `tKdivide`, para esto utilizamos `car` [código 2.1, página 50]. Formamos el conjunto C con las tuplas que tienen una primera parte igual al elemento a requerido. Como a debe ser comparada con una tupla aunque a puede ser o no una tupla, es necesario convertir el parámetro a en una tupla $\langle a \rangle$ si es necesario.

Para obtener el conjunto de respuesta D, se debe obtener el elemento final de cada tupla en C, para lo cual utilizamos una combinación de `car` y `cdr`.

Finalmente habrá que hacer un conjunto con los elementos obtenidos, pues cabe la posibilidad de tener elementos repetidos.

Ejemplo 6.32

Utiliza la función `im` para obtener la imagen de $\langle 1,3 \rangle$ en la relación $R \leftarrow \{\langle 1,3,'b' \rangle, \langle 1,2,'c' \rangle, \langle 2,3,'a' \rangle, \langle 4,4,'a' \rangle, \langle 3,4,'a' \rangle, \langle 2,4,'a' \rangle, \langle 1,3,'a' \rangle, \langle 1,3,'b' \rangle, \langle 1,1,'b' \rangle, \langle 3,3,'b' \rangle\}$

```
>>> R = [[1, 3, 'b'], [1, 2, 'c'], [2, 3, 'a'], [4, 4, 'a'], [3, 4, 'a'], [2, 4, 'a'],
 [1, 3, 'a'], [1, 3, 'b'], [1, 1, 'b'], [3, 3, 'b']]
>>> im(R,[1,3])
['a', 'b']
>>>
```

Ejercicios

1. Considera los conjuntos $R \leftarrow \{1,\ldots,10\}$ y $S \leftarrow \{0,1,2\}$ y escribe la lista de las tuplas que corresponden a la relación $T : R \to S$ siguiente:

$$T \leftarrow \{\langle r,s\rangle \in R \times S \,|\, s = r \bmod 3\}$$

2. Escribe la representación en forma de lista de pares de la siguiente matriz de pertenencia.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & \cdot & \cdot & 1 & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & 1 & 1 \\ 1 & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 & 1 \\ 1 & 1 & 1 & \cdot & \cdot & 1 \\ \cdot & 1 & \cdot & 1 & 1 & \cdot \end{pmatrix} \end{array}$$

3. Una tupla no vacía $\alpha \leftarrow \langle a_1,\ldots,a_n\rangle$ tiene sus elementos ordenados, el subíndice indica la posición en la que se encuentra tal elemento en la tupla, por ejemplo cuando $\alpha \leftarrow \langle f,g,h,i,j\rangle$, el elemento que se encuentra en la posición 3 es la letra h. Escribe una definición matemática para una operación con tuplas en las que se desea quitar el i-ésimo elemento de la tupla. El resultado debe ser una tupla en donde no se encuentre el elemento que estaba en la posición i, y su lugar lo ocupe aquel elemento en la posición $i+1$.
   Por ejemplo, en α al quitar el elemento de la posición 3 resulta $\langle f,g,i,j\rangle$. Tratar de quitar un elemento en una posición que no existe en la tupla, resulta en una tupla sin cambios.
4. Considera los conjuntos $A \leftarrow \{1,\ldots,100\}$ y $B \leftarrow \{1,\ldots,100\}$. Calcula el dominio y el rango en las siguientes relaciones:
   a) $R_1 \leftarrow \{\langle x,y\rangle \in A \times B \,|\, x > y\}$
   b) $R_2 \leftarrow \{\langle x,y\rangle \in A \times B \,|\, x^2 < 2y\}$
   c) $R_3 \leftarrow \{\langle 63,1\rangle,\langle 73,52\rangle,\langle 59,93\rangle,\langle 36,10\rangle,\langle 12,61\rangle,\langle 17,38\rangle,\langle 43,90\rangle,\langle 50,63\rangle\}$
   d) $R_4 \leftarrow \{\langle 14,45\rangle,\langle 99,94\rangle,\langle 50,84\rangle,\langle 54,61\rangle,\langle 51,93\rangle,\langle 76,28\rangle,\langle 25,70\rangle,\langle 73,42\rangle\}$
5. Escribe una función recursiva llamada `tInversa`, cuyo objetivo es obtener la inversa de una tupla. La función debe recibir de manera obligatoria una tupla `t` y devuelve una tupla. *Ayuda:* Para hacer el procedimiento recursivo óptimo, puedes definir un parámetro opcional que lleve el resultado a medida que se vaya generando la respuesta. Las funciones auxiliares que puedes utilizar son `car`, `cdr` y `tEscribeEnPos`.

```
>>> tInversa([])
[]
>>> tInversa([1])
[1]
>>> tInversa([1,2])
[2, 1]
>>> tInversa([1,2,3])
[3, 2, 1]
>>>
```

6. Escribe una función llamada `paresArels` que servirá para transformar la representación de lista de pares de una relación a su respectiva representación como lista de relaciones [página 140]. La función debe recibir una lista de pares `Lp` y devolver una lista de relaciones.

```
>>> S = [[1,4],[1,7],[2,8],[2,9],[3,1],[3,2],[4,5],[4,7],[4,9],[5,1],[5,2],[6,6],
[7,3],[7,7],[7,9],[8,4],[8,5],[9,6],[9,9]]
>>> paresArels(S)
[[8, 4, 5],
 [1, 4, 7],
 [2, 8, 9],
 [3, 1, 2],
 [7, 3, 7, 9],
 [4, 5, 7, 9],
 [5, 1, 2],
 [6, 6],
 [9, 6, 9]]
>>>
```

7. Escribe una función llamada `paresAmat` que servirá para transformar la representación de lista de pares de una relación a su respectiva representación como matriz de pertenencia [página 140]. La función debe recibir una lista de pares `Lp` y devolver una lista de listas que contengan 1 o 0. Nota: Con el dominio y rango iguales, debes tener cuidado de utilizar el mismo orden de los elementos,

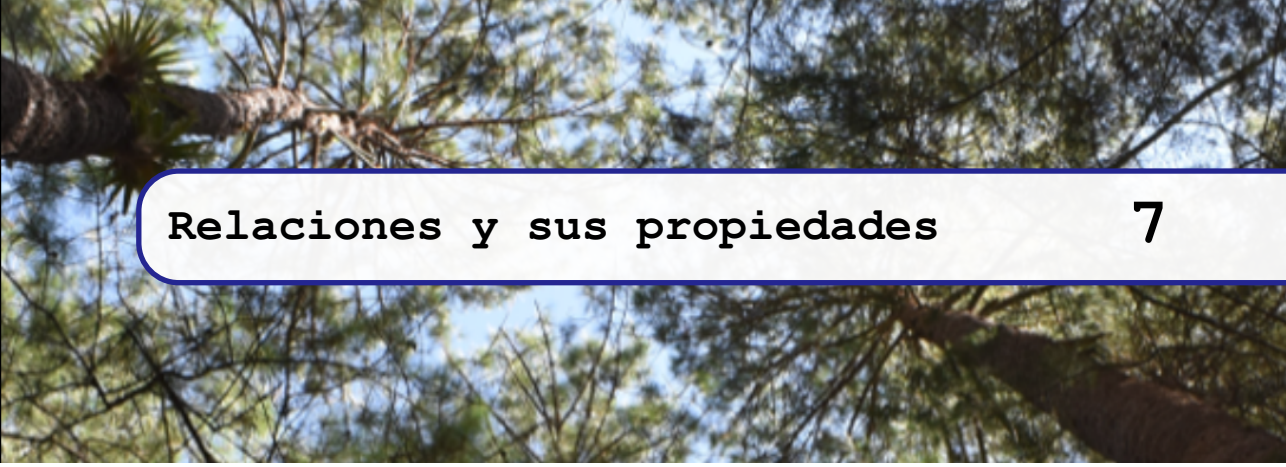
```
>>> S = [[1,4],[1,7],[2,8],[2,9],[3,1],[3,2],[4,5],[4,7],[4,9],[5,1],[5,2],[6,6],
[7,3],[7,7],[7,9],[8,4],[8,5],[9,6],[9,9]]
>>> paresAmat(S)
[[0, 0, 0, 1, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 1, 1],
 [1, 1, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 1, 0, 1],
 [1, 1, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0, 0, 1, 0, 1],
 [0, 0, 0, 1, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 1, 0, 0, 1]]
>>>
```

8. Escribe una función en `Python` que se llame `txtApares` y que reciba una cadena de texto `nomarch` que indica el nombre de un archivo de texto sin formato que contenga la descripción de una relación en forma de lista de relaciones [página 140]. El programa debe devolver una lista de pares si el archivo ha sido leído sin problema y `False` si hubo algo que haya impedido la lectura correcta.

rel.txt

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

```
>>> txtApares("rel.txt")
[[8, 4], [8, 5], [1, 4], [1, 7], [2, 8], [2, 9], [3, 1], [3, 2], [7, 3],
 [7, 7], [7, 9], [4, 5], [4, 7], [4, 9], [5, 1], [5, 2], [6, 6], [9, 6],
 [9, 9]]
>>>
```



Relaciones y sus propiedades 7

7.1 Propiedades intrínsecas

En esta sección se revisan las propiedades que son consideradas esenciales en las relaciones sobre un conjunto, estas relaciones están definidas sobre un conjunto A que actúa como el dominio y codominio de la relación. Cuando la relación está dada en forma de un conjunto de pares ordenados, el conjunto de referencia A debe ser considerado como $\mathtt{dom}(R) \cup \mathtt{cod}(R)$.

7.1.1 Reflexividad

Definición 7.1.1 -- Reflexividad. Decimos que una relación $R : A \to A$ es **reflexiva** si cumple $\forall a \in A : \langle a, a \rangle \in R$.

Así, en una relación reflexiva, cada elemento del conjunto de referencia se relaciona consigo mismo.

Ejemplo 7.1

Las siguientes colecciones de pares son relaciones reflexivas:

1. $\{\langle 1,1\rangle, \langle 1,2\rangle, \langle 2,2\rangle\}$; el conjunto de referencia coincide con el dominio, que es $\mathtt{dom}(R) = \{1,2\}$ y notamos que $1R1$ y $2R2$.
2. $\{\langle a,b\rangle, \langle b,c\rangle, \langle a,a\rangle, \langle b,b\rangle, \langle c,c\rangle\}$; el dominio es $\mathtt{dom}(R) = \{a,b,c\}$, notamos que aRa, bRb y cRc, aunque haya otros elementos relacionados.
3. $\{\langle a,a\rangle, \langle b,b\rangle, \langle c,c\rangle\}$.

Los conjuntos de pares siguientes, no son relaciones reflexivas:

1. $\{\langle a,a\rangle, \langle b,b\rangle, \langle c,c\rangle, \langle c,d\rangle\}$; suponiendo que A es la unión del dominio con el rango, tenemos que $A \mapsto \{a,b,c,d\}$, por lo que esperamos tener los cuatro pares reflexivos, pero hace falta $\langle d,d\rangle$.
2. $\{\langle 4,5\rangle, \langle 5,6\rangle, \langle 6,7\rangle, \langle 7,4\rangle\}$.

Ejemplo 7.2

Considera la relación «Un alumno está relacionado con otro alumno si tienen el mismo número de credencial». Como cada alumno tiene una credencial con un número, y en principio este número es único para cada estudiante, cada alumno está relacionado con él mismo.

Código 7.1: *Verifica que una relación sea reflexiva*

```
def esRef(R:list, D:list=None)-> bool:
    """
    Determina si una relación es reflexiva
    """
    if D is None: D = union(dom(R), ran(R))
    return paraTodo(lambda a: en(tupla(a, a), R), D)
```

Ejemplo 7.3

Utiliza la herramienta `esRef` para determinar si las relaciones $R \leftarrow \{\langle 1,1\rangle,\langle 1,2\rangle,\langle 2,2\rangle,\langle 2,4\rangle,\langle 3,1\rangle,\langle 3,3\rangle,\langle 4,3\rangle,\langle 4,4\rangle\}$ y $S \leftarrow \{\langle 1,4\rangle,\langle 2,4\rangle,\langle 4,3\rangle,\langle 3,2\rangle,\langle 2,5\rangle,\langle 5,3\rangle,\langle 1,3\rangle,\langle 2,1\rangle,\langle 3,1\rangle,\langle 4,5\rangle,\langle 1,2\rangle,\langle 2,3\rangle\}$ son reflexivas o no.

```
>>> R = [[1,1], [1,2], [2,2], [2,4], [3,1], [3,3], [4,3], [4,4]]
>>> esRef(R)
True
>>> S = [[1, 4], [2, 4], [4, 3], [3, 2], [2, 5], [5, 3], [1, 3], [2, 1], [3, 1], [4, 5],
     [1, 2], [2, 3]]
>>> esRef(S)
False
>>>
```

Observamos que en la relación R, todos los elementos del dominio están relacionados con ellos mismos, así que la proposición `esRef(R)` es `True`. En el caso de la relación S, observamos que tanto el dominio como el rango son iguales, pero hacen falta los pares de la forma $\langle a,a\rangle$ con $a \in \texttt{dom}(S)$, por lo que `esRef(S)` es una proposición `False`.

7.1.2 Irreflexividad

Definición 7.1.2 -- Irreflexividad. Una relación $R : A \to A$ es **irreflexiva** si cumple con $\forall a \in A : a \not R a$.

La propiedad de irreflexividad la tienen las relaciones en donde no hay pares en los que un elemento esté relacionado consigo mismo.

Ejemplo 7.4

Las siguientes relaciones tienen la propiedad de irreflexividad:

1. La relación entre las personas definida como «x es padre biológico de y», es una relación irreflexiva, pues nadie puede ser padre biológico de sí mismo.
2. La relación $R : \mathbb{Z} \to \mathbb{Z}$ donde aRb si $a > b$. Claramente un número entero a no es mayor que si mismo, por lo que es una relación irreflexiva.

Ejemplo 7.5

Los siguientes ejemplos son relaciones irreflexivas:

1. $\{\langle 1,2\rangle,\langle 3,2\rangle,\langle 4,2\rangle,\langle 3,1\rangle,\langle 2,4\rangle\}$ en el dominio $\{1,2,3,4\}$. Observa que no hay pares en que las entradas sean las mismas.
2. $\{\langle 3,2\rangle,\langle 2,4\rangle,\langle 1,4\rangle,\langle 3,1\rangle\}$ en el mismo dominio $\{1,2,3,4\}$.

Los siguientes ejemplos no son relaciones irreflexivas:

1. $\{\langle 1,2\rangle,\langle 3,2\rangle,\langle 4,2\rangle,\langle 3,1\rangle,\langle 2,2\rangle\}$ en el dominio $\{1,2,3,4\}$, porque la propiedad $a \not R a$ no se cumple para todos los elementos del dominio.
2. $\{\langle x,y\rangle \in \{1,2,3,4\}^2 \,|\, y = x^2 \bmod 4\}$ no es una relación irreflexiva, pues los pares en la relación formal el conjunto $\{\langle 1,1\rangle,\langle 2,0\rangle,\langle 3,1\rangle,\langle 4,0\rangle\}$ y encontramos que $1R1$.

Comparando las matrices de pertenencia de una relación reflexiva y una relación irreflexiva, se observa que las relaciones reflexivas tienen un 1 sobre cada elemento de la diagonal principal, mientras que una relación irreflexiva tiene 0 en cada elemento de la diagonal principal.

Reflexividad

$$\begin{bmatrix} 1 & \circ & \dots & \circ \\ \circ & 1 & \dots & \circ \\ \vdots & \vdots & \dots & \vdots \\ \circ & \circ & \dots & 1 \end{bmatrix}$$

Irreflexividad

$$\begin{bmatrix} 0 & \circ & \dots & \circ \\ \circ & 0 & \dots & \circ \\ \vdots & \vdots & \dots & \vdots \\ \circ & \circ & \dots & 0 \end{bmatrix}$$

Código 7.2: *Verifica que una relación sea irreflexiva*

```
def esIrref(R:list, D:list=None)-> bool:
    """
    Determina si una relación es irreflexiva
    """
    if D is None: D = union(dom(R), ran(R))
    return paraTodo(lambda a: neg(en(tupla(a, a), R)), D)
```

Ejemplo 7.6

Utiliza la función `esIrref` para determinar si $R \leftrightarrowtail \{\langle x,y\rangle \in \{2,4,6\}^2 \,|\, x \neq y\}$ es irreflexiva.

Observa que R es un subconjunto de $\{2,4,6\} \times \{2,4,6\}$, por lo que podemos escribir:

```
>>> R = subc(lambda p:car(p)!=car(cdr(p)), pCart([2,4,6], [2,4,6]))
>>> esIrref(R)
True
>>> R
[[2, 4], [2, 6], [4, 2], [4, 6], [6, 2], [6, 4]]
>>>
```

Observamos que en R no hay elemento alguno que esté relacionado consigo mismo.

7.1.3 Simetría

Definición 7.1.3 -- Simetría. Una relación $R : A \to A$ es **simétrica** si cumple

$$\forall \langle a,b\rangle \in R : \langle b,a\rangle \in R.$$

En una relación simétrica los pares que forman la relación, involucran elementos que están relacionados uno con el otro.

Ejemplo 7.7

Los siguientes ejemplos son de relaciones simétricas

1. Los empleados son relacionados del siguiente modo. El empleado a está relacionado con b, si ambos han trabajado en el mismo proyecto. Así cuando aRb, también bRa.
2. Sea $R \leftrightarrowtail \{\langle a,b\rangle \in \mathbb{Z}^2 \,|\, a = b\}$. En esta relación un elemento está relacionado con otro siempre que sean iguales, por lo que son de la forma $\langle a,a\rangle$. Esta relación, además de ser simétrica, también es reflexiva.

3. Si A es el conjunto de rectas en el plano, decimos que una recta a está relacionada con otra recta b, si a es paralela a b.

Si vemos la relación $R : A \to A$ como una matriz de pertenencia $M_{[n\times n]}$, en una relación con la propiedad de simetría, se puede observar que el valor en las posiciones $\langle i,i\rangle$, con $1 \le i \le n$, pueden ser 0 o 1; y el valor en alguna posición $\langle i,j\rangle$ es el mismo que el valor en la posición $\langle j,i\rangle$.

$$M \mapsto \begin{array}{c} \\ 1 \\ \\ i \\ \\ j \\ \\ n \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & & i & & j & & n \end{array} \\ \begin{pmatrix} \circ & \dots & \circ & \dots & \circ & \dots & \circ \\ \vdots & & \vdots & & \vdots & & \vdots \\ \circ & \dots & v_{i,i} & \dots & \textcircled{$v_{i,j}$} & \dots & \circ \\ \vdots & & \vdots & & \vdots & & \vdots \\ \circ & \dots & \textcircled{$v_{j,i}$} & \dots & \circ & \dots & \circ \\ \vdots & & \vdots & & \vdots & & \vdots \\ \circ & \dots & \circ & \dots & \circ & \dots & \circ \end{pmatrix} \end{array}$$

Código 7.3: *Verifica que una relación sea simétrica*

```
def esSim(R:list)-> bool:
    """
    Determina si una relación es simétrica
    R: list
    """
    return paraTodo(lambda t:en(tInversa(t), R), R)
```

Ejemplo 7.8

Utiliza la función `esSim` para verificar que las relaciones:

$$R \mapsto \{\langle 3,1\rangle,\langle 3,5\rangle,\langle 1,2\rangle,\langle 4,5\rangle,\langle 5,3\rangle,\langle 3,3\rangle,\langle 4,4\rangle,\langle 5,4\rangle,\langle 1,4\rangle,\langle 1,3\rangle,\langle 2,1\rangle,\langle 4,1\rangle\}$$

$$S \mapsto \{\langle 4,4\rangle,\langle 4,5\rangle,\langle 5,4\rangle,\langle 3,1\rangle,\langle 1,4\rangle,\langle 3,5\rangle,\langle 5,2\rangle,\langle 5,3\rangle,\langle 1,5\rangle,\langle 3,3\rangle,\langle 3,4\rangle,\langle 2,5\rangle\}$$

$$M_R \mapsto \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & \cdot & \cdot \\ 1 & \cdot & \cdot & \cdot & 1 \\ 1 & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot \end{pmatrix} \end{array} \qquad M_S \mapsto \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & 1 & 1 \\ 1 & 1 & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot \end{pmatrix} \end{array}$$

M_R es simétrica respecto a su diagonal principal, aunque no todos los elementos de la diagonal principal son 1. Por su parte, M_S no es simétrica, puesto que hacen falta algunos pares simétricos, por ejemplo se encuentra $\langle 4,1\rangle$ pero no se encuentra $\langle 1,4\rangle$.

```
>>> R = [[3, 1], [3, 5], [1, 2], [4, 5], [5, 3], [3, 3], [4, 4], [5, 4], [1, 4], [1, 3],
     [2, 1], [4, 1]]
>>> esSim(R)
True
>>> S = [[4, 4], [4, 5], [5, 4], [3, 1], [1, 4], [3, 5], [5, 2], [5, 3], [1, 5], [3, 3],
     [3, 4], [2, 5]]
>>> esSim(S)
False
>>>
```

7.1.4 Asimetría

Definición 7.1.4 -- Asimetría. Una relación $R : A \to A$ es **asimétrica** cuando

$$\forall \langle a,b \rangle \in R : \langle b,a \rangle \notin R$$

En una relación asimétrica, si un elemento a está relacionado con un elemento b, entonces tal elemento b no está relacionado con el elemento a.

Ejemplo 7.9

Las siguientes relaciones son asimétricas:

1. En las personas, una persona a está relacionada con una persona b si a «es progenitor de» b. Claramente si a es progenitor de b, b no puede ser progenitor de a.
2. La relación $R : \mathbb{Z} \to \mathbb{Z}$ definida como $\{\langle x,y \rangle \in \mathbb{Z}^2 \mid x > y\}$.
3. La relación dada por los pares $\{\langle 2,1 \rangle, \langle 3,1 \rangle, \langle 3,2 \rangle, \langle 4,1 \rangle, \langle 4,2 \rangle, \langle 4,3 \rangle, \langle 5,1 \rangle, \langle 5,2 \rangle, \langle 5,3 \rangle, \langle 5,4 \rangle, \langle 6,1 \rangle, \langle 6,2 \rangle, \langle 6,3 \rangle, \langle 6,4 \rangle, \langle 6,5 \rangle\}$, que tiene como matriz de pertenencia

$$M_S \leftarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & \cdot & \cdot \\ 1 & 1 & 1 & 1 & 1 & \cdot \end{pmatrix} \end{array}$$

La máxima cantidad de pares en una relación asimétrica está dada por la expresión

$$\frac{|R| * (|R| - 1)}{2},$$

la matriz de pertenencia del ejemplo 7.9 muestra una relación con la máxima cantidad de pares; al agregar un solo par, ocasiona que la relación deje de ser asimétrica.

De la matriz de pertenencia, observamos que la cantidad de pares obedecen a la suma

$$\sum_{i=1}^{|R|-1} i = 1 + 2 + \cdots + (|R| - 1) \mapsto \frac{|R| * (|R| - 1)}{2},$$

Código 7.4: *Verifica que una relación sea asimétrica*

```
1  def esAsim(R:list)-> bool:
2      """
3      Determina si una relación es asimétrica
4      R: list
5      """
6      return paraTodo(lambda t:neg(en(tInversa(t), R)), R)
```

Ejemplo 7.10

Utiliza la función `esAsim` para verificar que la relación $R \leftarrow \{\langle 5,1 \rangle, \langle 3,5 \rangle, \langle 3,1 \rangle, \langle 6,4 \rangle, \langle 2,6 \rangle, \langle 6,1 \rangle, \langle 5,4 \rangle, \langle 2,4 \rangle, \langle 3,6 \rangle\}$ sea una relación asimétrica.

```
>>> R = conj([5, 1], [3, 5], [3, 1], [6, 4], [2, 6], [6, 1], [5, 4], [2, 4], [3, 6])
>>> esAsim(R)
True
>>>
```

7.1.5 Antisimetría

Definición 7.1.5 -- Antisimetría. Una relación $R : A \to A$ es **antisimétrica** cuando se cumple $\forall \langle a,b \rangle \in R : a \neq b \to \langle b,a \rangle \notin R$.

La antisimetría no es lo contrario de la simetría, la antisimetría es una relación binaria en la que se verifica que un par $\langle a,b \rangle$ de elementos pueden estar relacionados cuando son iguales o bien cuando el par inverso $\langle b,a \rangle$ no es parte de la relación. Otro modo de entenderlo es que el único modo en que aRb y bRa, es que $a = b$.

Ejemplo 7.11

La matriz de pertenencia siguiente, corresponde a una relación antisimétrica; observa que los valores en la diagonal principal pueden ser 1 o 0, mientras que los valores 1 fuera de la diagonal, corresponden a posiciones $\langle i,j \rangle$ en donde el par inverso $\langle j,i \rangle$ tiene un valor 0. En aquellas posiciones donde el valor es 0, el la posición inversa también puede haber un 0.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 \\ 1 & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array}$$

Por claridad, analizaremos dos elementos de la relación:

1. El par $\langle 5,1 \rangle \in R$, porque $5 \neq 1$ es `True` y también lo es la proposición $\langle 1,5 \rangle \notin R$, por lo que $(5 \neq 1) \to (\langle 1,5 \rangle \notin R) \mapsto$ `True`.
2. El par $\langle 2,2 \rangle \in R$, porque $2 \neq 2$ es `False` y la proposición $\langle 2,1 \rangle \notin R$ también es `False`, por lo que $(2 \neq 2) \to (\langle 2,1 \rangle \notin R) \mapsto$ `True`.
3. Un par fuera de la relación, por ejemplo $\langle 4,2 \rangle$, el antecedente de la implicación es `False` puesto que $\langle 4,2 \rangle \notin R$, lo que permita que la implicación sea `True`.

Una matriz de pertenencia que corresponde a una relación antisimétrica, puede tener 1's o 0's en la diagonal principal, en aquellas posiciones fuera de la diagonal, una posición $\langle i,j \rangle$ tendrá valor 1, siempre que en la posición $\langle j,i \rangle$ se encuentre un valor 0.

La tabla de verdad de $a \neq b \to \langle b,a \rangle \notin R$ permite ver que $a = b \to \langle a,b \rangle \in R$. es una expresión lógica equivalente, lo que es una buena noticia porque simplifica un poco la programación.

$a \neq b$	$\langle b,a \rangle \notin R$	$a \neq b \to \langle b,a \rangle \notin R$	$\overset{\neg(a\neq b)}{a = b}$	$\overset{\neg(\langle b,a\rangle \notin R)}{\langle b,a \rangle \in R}$	$\langle b,a \rangle \in R \to a = b$
True	True	True	False	False	True
True	False	False	False	True	False
False	True	True	True	False	True
False	False	True	True	True	True

Código 7.5: *Verifica que una relación sea antisimétrica*

```
def esAntisim(R:list)-> bool:
    """
    Determina si una relación es antisimétrica
    """
    return paraTodo(lambda t:
                    impl(en(tInversa(t),R),
                         car(t)==car(cdr(t))), R)
```

Ejemplo 7.12

Alicia, Roberto, Susana, Brenda y Alfredo son amigos, ellos han hecho una campaña para recaudar fondos para una actividad de beneficencia. Alicia recaudó \$1200.0; Roberto recaudó \$1145.0; Susana, \$563.0; Brenda, \$1723.0 y Alfredo recaudó \$945.0. Esta información ayuda a formar el siguiente conjunto $A \leftarrow \{\langle ali, 1200\rangle, \langle rob, 1145\rangle, \langle sus, 563\rangle, \langle bren, 1723\rangle, \langle alf, 945\rangle\}$, con lo que se forma la siguiente relación.

Se ha diseñado la siguiente relación, «la persona a está relacionada con la persona b si la cantidad recaudada por a es menor o igual a la cantidad recaudada por b».

Así, se tiene la siguiente matriz de pertenencia:

$$M \leftarrow \begin{array}{c} \\ Alicia \\ Roberto \\ Susana \\ Brenda \\ Alfredo \end{array} \begin{array}{c} \begin{array}{ccccc} Alicia & Roberto & Susana & Brenda & Alfredo \end{array} \\ \begin{pmatrix} 1 & \cdot & \cdot & 1 & \cdot \\ 1 & 1 & \cdot & 1 & \cdot \\ 1 & 1 & 1 & 1 & 1 \\ \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & 1 & 1 \end{pmatrix} \end{array}$$

```
>>> R = [[['ali', 1200], ['ali', 1200]], [['ali', 1200], ['bre', 1723]],
         [['rob', 1145], ['ali', 1200]], [['rob', 1145], ['rob', 1145]],
         [['rob', 1145], ['bre', 1723]], [['sus', 563], ['ali', 1200]],
         [['sus', 563], ['rob', 1145]], [['sus', 563], ['sus', 563]],
         [['sus', 563], ['bre', 1723]], [['sus', 563], ['alf', 945]],
         [['bre', 1723], ['bre', 1723]], [['alf', 945], ['ali', 1200]],
         [['alf', 945], ['rob', 1145]], [['alf', 945], ['bre', 1723]],
         [['alf', 945], ['alf', 945]]]
>>> esAntisim(R)
True
>>>
```

7.1.6 Transitividad

Esta propiedad toma su nombre del propio concepto de «transitivo», que se refiere a lo que pasa de uno a otro, formalmente

Definición 7.1.6 -- Transitividad. Una relación $R : A \to A$ es transitiva si cumple

$$\forall a \in \mathrm{dom}(R) : (\forall c \in \mathtt{Im}(R, \mathtt{im}(R, a)) : \langle a, c\rangle \in R)$$

En una relación transitiva, el predicado o propiedad que relaciona un elemento con otro se verifica en terceros, si aRb y bRc entonces aRc, donde $a, b, c \in A$, como se muestra en la siguiente figura.

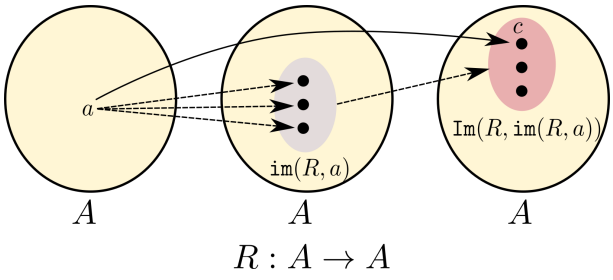


Figura 7.1: *Comportamiento de la verificación de una relación transitiva*

Supongamos ahora la relación $\{\langle 1,2\rangle,\langle 1,3\rangle,\langle 1,4\rangle,\langle 2,3\rangle,\langle 2,4\rangle,\langle 3,4\rangle\}$, la relación es reflexiva, veamos:

$$\mathtt{dom}(R) \mapsto \{1,2,3\}.$$

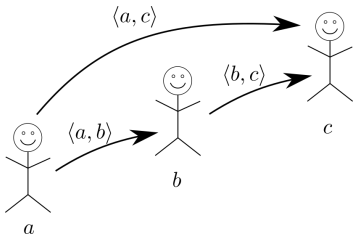
$$\forall a \in \{1,2,3\} : (\forall c \in \mathtt{Im}(R, \mathtt{im}(R,a)) : \langle a,c\rangle \in R)$$

$$\begin{aligned}
(a \leftarrow 1) &: \forall c \in \mathtt{Im}(R, \mathtt{im}(R,1)) : \langle 1,c\rangle \in R \\
&: \forall c \in \mathtt{Im}(R, \{2,3,4\}) : \langle 1,c\rangle \in R \\
&: \forall c \in \{3,4\} : \langle 1,c\rangle \in R \\
&: \langle 1,3\rangle \in R \wedge \langle 1,4\rangle \in R \mapsto \mathtt{True} \\
(a \leftarrow 2) &: \forall c \in \mathtt{Im}(R, \mathtt{im}(R,2)) : \langle 2,c\rangle \in R \\
&: \forall c \in \mathtt{Im}(R, \{3,4\}) : \langle 2,c\rangle \in R \\
&: \forall c \in \{4\} : \langle 2,c\rangle \in R \\
&: \langle 2,4\rangle \in R \mapsto \mathtt{True} \\
(a \leftarrow 3) &: \forall c \in \mathtt{Im}(R, \mathtt{im}(R,3)) : \langle 3,c\rangle \in R \\
&: \forall c \in \mathtt{Im}(R, \{4\}) : \langle 3,c\rangle \in R \\
&: \forall c \in \{\} : \langle 3,c\rangle \in R \mapsto \mathtt{True}
\end{aligned}$$

Ejemplo 7.13

Considera las siguientes relaciones:

1. Imaginate la siguiente relación sobre las personas. «Dos personas a y b están relacionados si la persona a es descendiente de la persona b». La siguiente figura muestra la relación transitiva que involucra a tres personas a, b y c, cuando a es descendiente de b y b es descendiente de c, se debe cumplir que a es descendiente de c.



2. Considera $\mathbb{Z}$ el conjunto de los números enteros y la relación $\geq$. Dos números a y b están relacionados cuando es cierto que $a \geq b$.
3. La relación $\{\langle 6,8\rangle,\langle 4,8\rangle,\langle 6,1\rangle,\langle 1,3\rangle,\langle 5,8\rangle,\langle 8,8\rangle,\langle 7,6\rangle,\langle 8,4\rangle,\langle 3,3\rangle,\langle 1,6\rangle,\langle 6,4\rangle,\langle 4,4\rangle,\langle 6,3\rangle,\langle 6,6\rangle,\langle 5,4\rangle,\langle 7,8\rangle,\langle 7,1\rangle,\langle 1,8\rangle,\langle 1,1\rangle,\langle 7,4\rangle,\langle 7,3\rangle,\langle 1,4\rangle\}$ sobre $\{1,2,3,4,5,6,7,8\}$ que tiene una matriz de pertenencia:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\left(\begin{array}{cccccccc}
1 & \cdot & 1 & 1 & \cdot & 1 & \cdot & 1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 \\
1 & \cdot & 1 & 1 & \cdot & 1 & \cdot & 1 \\
1 & \cdot & 1 & 1 & \cdot & 1 & \cdot & 1 \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1
\end{array}\right)
\end{array}$$

En una matriz de pertenencia se puede determinar si una relación es transitiva, cuando los valores en las entradas de la matriz cumplen

$$\langle i,j\rangle = 1 \wedge \langle j,k\rangle = 1 \rightarrow \langle i,k\rangle = 1.$$

Cuando una relación no vacía tiene elementos relacionados en los que no se cumple el antecedente de la implicación, la relación es transitiva, por ejemplo siendo $A = \{1,2,3,4\}$ y $R : A \to A$ donde $R = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 4,2\rangle\}$, la relación R si es transitiva porque al no haber pares de la forma $\langle a,b\rangle$ y $\langle b,c\rangle$, el antecedente de la implicación es `False`, lo que ocasiona que la implicación sea `True` cumpliéndose la transitividad.

7.2 Relaciones especiales

7.2.1 Cerradura reflexiva, simétrica y transitiva

Definición 7.2.1 -- Cerradura reflexiva. La cerradura reflexiva de una relación $R : A \to A$ denotada $R^{(\mathbb{R})}$ es la relación sobre el mismo conjunto A, que es es la relación de cardinalidad mínima que contiene a R y es reflexiva.

Para obtener la cerradura reflexiva $R^{(\mathbb{R})}$ de una relación R, es necesario calcular el conjunto de pares ordenados que son reflexivos y agregarlos a la relación R, de este modo:

$$R^{(\mathbb{R})} \leftarrow\!\!\shortmid R \cup \{\langle a,a\rangle \mid a \in A\}$$

Ejemplo 7.14

Calcular la cerradura reflexiva de la relación $R \leftarrow\!\!\shortmid \{\langle 1,3\rangle, \langle 2,5\rangle, \langle 3,3\rangle, \langle 4,2\rangle, \langle 5,5\rangle\}$, cuya matriz de pertenencia se muestra enseguida.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} \odot & \cdot & 1 & \cdot & \cdot \\ \cdot & \odot & \cdot & \cdot & 1 \\ \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & 1 & \cdot & \odot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 \end{pmatrix} \end{array}$$

Los círculos muestran los lugares en donde faltan elementos para la cerradura reflexiva, de modo que se deben agregar los pares $\langle 1,1\rangle, \langle 2,2\rangle$ y $\langle 4,4\rangle$.

La cerradura reflexiva se obtiene calculando $\{\langle a,a\rangle \in A \times A \mid \langle a,a\rangle \notin R\}$, que son los pares reflexivos que no están presentes en la relación, así

$$\begin{aligned} R^{(\mathbb{R})} &\leftarrow\!\!\shortmid \{\langle 1,3\rangle, \langle 2,5\rangle, \langle 3,3\rangle, \langle 4,2\rangle, \langle 5,5\rangle\} \cup \{\langle 1,1\rangle, \langle 2,2\rangle, \langle 4,4\rangle\} \\ &\leftarrow\!\!\shortmid \{\langle 1,3\rangle, \langle 2,5\rangle, \langle 3,3\rangle, \langle 4,2\rangle, \langle 5,5\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \langle 4,4\rangle\} \end{aligned}$$

Código 7.6: *Cerradura reflexiva*

```
1  def cerrRef(R):
2      """
3      Obtiene la cerradura reflexiva de una relación
4      """
5      A = union(dom(R), ran(R))
6      pRef = enCada(lambda a:tupla(a,a), A)
7      R = union(R, pRef)
8      return R
```

```
>>> R = [[1,3], [2,5], [3,3], [4,2], [5,5]]
>>> cerrRef(R)
[[1, 3], [2, 5], [3, 3], [4, 2], [5, 5], [1, 1], [2, 2], [4, 4]]
>>>
```

Definición 7.2.2 -- Cerradura simétrica. La cerradura simétrica de una relación $R : A \to A$ denotada $R^{(S)}$ es la relación sobre el mismo conjunto A, que es es la relación de cardinalidad mínima que contiene a R y es simétrica.

Para obtener $R^{(S)}$ se deben agregar, los pares simétricos de los pares en R, de este modo:

$$R^{(S)} \leftarrowtail R \cup \{\langle b,a\rangle | \langle a,b\rangle \in R\}.$$

Ejemplo 7.15

Calcular la cerradura simétrica de la relación $R \leftarrowtail \{\langle 1,3\rangle,\langle 2,5\rangle,\langle 3,3\rangle,\langle 4,2\rangle,\langle 5,5\rangle\}$, cuya matriz de pertenencia se muestra enseguida.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \textcircled{\cdot} & 1 \\ \textcircled{\cdot} & \cdot & 1 & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \textcircled{\cdot} & \cdot & \cdot & 1 \end{pmatrix} \end{array}$$

Los círculos muestran los lugares en donde faltan elementos para la cerradura simétrica.

La cerradura simétrica se obtiene calculando los pares simétricos que no están presentes en la relación, así los pares que faltan son $\langle 3,1\rangle,\langle 2,4\rangle$ y $\langle 5,2\rangle$:

$$\begin{aligned} R^{(S)} &\leftarrowtail \{\langle 1,3\rangle,\langle 2,5\rangle,\langle 3,3\rangle,\langle 4,2\rangle,\langle 5,5\rangle\} \cup \{\langle 3,1\rangle,\langle 2,4\rangle,\langle 5,2\rangle\} \\ &\leftarrowtail \{\langle 1,3\rangle,\langle 2,5\rangle,\langle 3,3\rangle,\langle 4,2\rangle,\langle 5,5\rangle,\langle 3,1\rangle,\langle 2,4\rangle,\langle 5,2\rangle\} \end{aligned}$$

Código 7.7: *Cerradura simétrica*

```
1  def cerrSim(R):
2      """
3      Obtiene la cerradura simétrica de una relación
4      """
5      CS = enCada(lambda p:tInversa(p), R)
6      R = union(R,CS)
7      return R
```

```
>>> R = [[1,3], [2,5], [3,3], [4,2], [5,5]]
>>> cerrSim(R)
[[1, 3], [2, 5], [3, 3], [4, 2], [5, 5], [3, 1], [5, 2], [2, 4]]
>>>
```

Definición 7.2.3 -- Cerradura transitiva. La cerradura transitiva de una relación $R : A \to A$ denotada $R^{(T)}$ es la relación sobre el mismo conjunto A, que es es la relación de cardinalidad mínima que contiene a R y es transitiva.

Para calcular la cerradura transitiva de una relación R, se deben revisar todos los pares de la relación para detectar aquellos pares $\langle a,b\rangle$ y $\langle b,c\rangle$ para verificar que el par $\langle a,c\rangle$ pertenezca a la relación; en caso de no pertenecer, entonces agregarlo.

Por ejemplo, considera la relación $\{\langle 1,2\rangle,\langle 2,3\rangle,\langle 3,4\rangle\}$, tal como está, la relación no es transitiva y los únicos elementos con el patrón requerido son $\langle 1,2\rangle$ con $\langle 2,3\rangle$; y $\langle 2,3\rangle$ con $\langle 3,4\rangle$, con lo que se deben agregar los pares $\langle 1,3\rangle$ y $\langle 2,4\rangle$ respectivamente, quedando ahora la relación

$$\{\langle 1,2\rangle,\langle 2,3\rangle,\langle 3,4\rangle,\langle 1,3\rangle,\langle 2,4\rangle\}.$$

En una segunda revisión, se detectan ahora los elementos $\langle 1,2\rangle$ con $\langle 2,4\rangle$ para agregar el nuevo par $\langle 1,4\rangle$. Luego ya no hay más elementos nuevos que cumplan el patrón transitivo y la cerradura transitiva de la relación es:

$$\{\langle 1,2\rangle,\langle 2,3\rangle,\langle 3,4\rangle,\langle 1,3\rangle,\langle 2,4\rangle,\langle 1,4\rangle\}.$$

Ejemplo 7.16

Calcular la cerradura transitiva de la relación $R \leftarrow \{\langle 1,2\rangle,\langle 1,4\rangle,\langle 4,2\rangle,\langle 4,3\rangle,\langle 5,1\rangle,\langle 3,1\rangle\}$, cuya matriz de pertenencia se muestra enseguida.

	1	2	3	4	5
1	·	1	·	1	·
2	·	·	·	·	·
3	1	·	·	·	·
4	·	1	1	·	·
5	1	·	·	·	·

Para calcular la cerradura transitiva, se deben localizar los pares $\langle a,b\rangle$ y $\langle b,c\rangle$ de la relación, luego agregar aquellos pares $\langle a,c\rangle$ que aún no pertenezcan a la relación. Por ejemplo, dado que se encuentran los pares $\langle 1,4\rangle$ y $\langle 4,3\rangle$, se debe agregar el par $\langle 1,3\rangle$.

Se ha determinado que se deben agregar los pares $\{\langle 1,3\rangle,\langle 4,1\rangle,\langle 5,2\rangle,\langle 5,4\rangle,\langle 5,3\rangle,\langle 3,2\rangle,\langle 3,4\rangle,\langle 3,3\rangle\}$, que se muestran en la siguiente matriz en un círculo.

	1	2	3	4	5
1	·	1	⊙	1	·
2	·	·	·	·	·
3	1	⊙	⊙	⊙	·
4	⊙	1	1	·	·
5	1	⊙	⊙	⊙	·

Después de haber agregado los pares recomendados, la relación es $\{\langle 1,2\rangle,\langle 1,4\rangle,\langle 4,2\rangle,\langle 4,3\rangle,\langle 5,1\rangle,\langle 3,1\rangle,\langle 1,3\rangle,\langle 4,1\rangle,\langle 5,2\rangle,\langle 5,4\rangle,\langle 5,3\rangle,\langle 3,2\rangle,\langle 3,4\rangle,\langle 3,3\rangle\}$, sin embargo aún es posible que la relación resultante aún no sea transitiva, pues al haber agregado nuevos pares es posible que falten otros. Por ejemplo al haber agregado el par $\langle 1,3\rangle$ tenemos ahora los pares $\langle 1,3\rangle$ y $\langle 3,1\rangle$ por lo que falta agregar el par $\langle 1,1\rangle$. Detectamos que se deben agregar los pares $\langle 1,1\rangle$ y $\langle 4,4\rangle$, que se muestran mediante un círculo en la siguiente matriz.

	1	2	3	4	5
1	⊙	1	1	1	·
2	·	·	·	·	·
3	1	1	1	1	·
4	1	1	1	⊙	·
5	1	1	1	1	·

Así $R^{(\mathrm{T})} \leftarrow \{\langle 1,2\rangle,\langle 1,4\rangle,\langle 4,2\rangle,\langle 4,3\rangle,\langle 5,1\rangle,\langle 3,1\rangle,\langle 1,3\rangle,\langle 4,1\rangle,\langle 5,2\rangle,\langle 5,4\rangle,\langle 5,3\rangle,\langle 3,2\rangle,\langle 3,4\rangle,\langle 3,3\rangle,\langle 1,1\rangle,\langle 4,4\rangle\}$.

7.2.2 Relación de equivalencia

Las relaciones de equivalencia son muy importantes en algunas aplicaciones, particularmente en aquellas en donde es difícil trabajar con un elemento en particular, porque ese elemento es de difícil acceso; pero menos complicado que utilizar otro elemento con las mismas características y más de fácil acceso; solamente se debe asegurar que ambos elementos sean equivalentes.

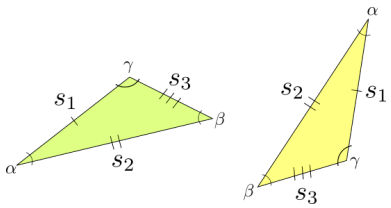
Definición 7.2.4 -- Relación de equivalencia. Una relación $R : A \to A$ es una relación de equivalencia si R es reflexiva, simétrica y transitiva al mismo tiempo.

Ejemplo 7.17

Si A es el conjunto de triángulos del plano y R la relación sobre A

$$R = \{\langle a, b\rangle \in A^2 | a \text{ es congruente con } b\}$$

Por su definición, dos triángulos son congruentes entre sí, si sus lados correspondientes tienen la misma longitud y sus ángulos correspondientes tienen la misma medida.



R es una relación de equivalencia porque:

1. Es reflexiva, porque un triángulo es congruente consigo mismo.
2. Es simétrica, ya que si t_1 es congruente con un triángulo t_2, ambos tienen lados correspondientes de la misma longitud y ángulos correspondientes iguales, por lo que t_2 es congruente con t_1.
3. Es transitiva. Esto sucede porque si un triángulo t_1 es congruente con otro triángulo t_2 significa que tienen lados correspondientes de la misma longitud y ángulos correspondientes iguales. Luego, si t_2 es congruente con un tercer triángulo t_3, estos triángulos tendrán las mismas medidas, luego t_1 es congruente con t_3, cumpliendo la propiedad transitiva.

Ejemplo 7.18

Consideremos la relación $R = \{\langle 1,3\rangle, \langle 1,5\rangle, \langle 2,2\rangle, \langle 4,6\rangle, \langle 1,1\rangle, \langle 4,4\rangle, \langle 3,3\rangle, \langle 5,5\rangle, \langle 6,6\rangle, \langle 3,1\rangle, \langle 5,1\rangle, \langle 6,4\rangle, \langle 3,5\rangle, \langle 5,3\rangle\}$. Suponiendo $A \leftarrow \{1,2,3,4,5,6\}$, $R : A \to A$ es una relación de equivalencia porque:

1. Es reflexiva ya que $\forall a \in A : \langle a, a\rangle \in R$, esto es $1R1 \wedge 2R2 \wedge 3R3 \wedge 4R4 \wedge 5R5 \wedge 6R6 \mapsto \texttt{True}$.
2. Es simétrica ya que $\forall \langle a, b\rangle \in R : \langle b, a\rangle \in R$, que se verifica revisando el par simétrico de cada par en R: si $1R3 \wedge 3R1 \mapsto \texttt{True}, \ldots, 5R3 \wedge 3R5 \mapsto \texttt{True}$.
3. Es transitiva. Esto se verifica revisando cada elemento $a \in A$.

$$\begin{aligned}
(a \leftarrow 1) &: \forall c \in \texttt{Im}(R, \texttt{im}(R,1)) : \langle 1, c\rangle \in R \\
&: \forall c \in \{3,5,1\} : \langle 1, c\rangle \in R \\
&: \langle 1,3\rangle \in R \wedge \langle 1,5\rangle \in R \wedge \langle 1,1\rangle \in R \mapsto \texttt{True} \\
(a \leftarrow 2) &: \forall c \in \texttt{Im}(R, \texttt{im}(R,2)) : \langle 2, c\rangle \in R \\
&: \forall c \in \{2\} : \langle 2, c\rangle \in R \\
&: \langle 2,2\rangle \in R \mapsto \texttt{True} \\
(a \leftarrow 3) &: \forall c \in \texttt{Im}(R, \texttt{im}(R,3)) : \langle 3, c\rangle \in R \\
&: \forall c \in \{5,1,3\} : \langle 3, c\rangle \in R \\
&: \langle 3,5\rangle \in R \wedge \langle 3,1\rangle \in R \wedge \langle 3,3\rangle \in R \mapsto \texttt{True} \\
(a \leftarrow 4) &: \forall c \in \texttt{Im}(R, \texttt{im}(R,4)) : \langle 4, c\rangle \in R \\
&: \forall c \in \{6,4\} : \langle 4, c\rangle \in R \\
&: \langle 4,6\rangle \in R \wedge \langle 4,4\rangle \in R \mapsto \texttt{True} \\
(a \leftarrow 5) &: \forall c \in \texttt{Im}(R, \texttt{im}(R,5)) : \langle 5, c\rangle \in R \\
&: \forall c \in \{3,1,5\} : \langle 5, c\rangle \in R \\
&: \langle 5,3\rangle \in R \wedge \langle 5,1\rangle \in R \wedge \langle 5,5\rangle \in R \mapsto \texttt{True} \\
(a \leftarrow 6) &: \forall c \in \texttt{Im}(R, \texttt{im}(R,6)) : \langle 6, c\rangle \in R \\
&: \forall c \in \{6,4\} : \langle 6, c\rangle \in R \\
&: \langle 6,6\rangle \in R \wedge \langle 6,4\rangle \in R \mapsto \texttt{True}
\end{aligned}$$

La relación R tiene una matriz de pertenencia

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & \cdot & 1 & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & 1 & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & 1 \\ 1 & \cdot & 1 & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & 1 \end{pmatrix} \end{array}$$

Definición 7.2.5 -- Clases de equivalencia. Sea R una relación de equivalencia en un conjunto A y $a \in A$. La **clase de equivalencia** de a es el conjunto de elementos relacionados con a y se denota $[a]_R$. Cuando se considera solamente una relación o no hay confusión acerca de qué relación es la referida, se puede omitir el subíndice R y escribir solamente $[a]$.

Al considerar la relación del ejemplo 7.18 se tienen las siguientes clases de equivalencia:

$$\begin{array}{lll} [1]_R \mapsto \{3,5,1\} & [2]_R \mapsto \{2\} & [4]_R \mapsto \{6,4\} \\ [3]_R \mapsto \{3,1,5\} & & [6]_R \mapsto \{6,4\} \\ [5]_R \mapsto \{5,1,3\} & & \end{array}$$

Ejemplo 7.19

Muestre todas las clases de equivalencia de la relación:

$$\begin{aligned} R \leftarrow\ & \{\langle a,a\rangle,\langle a,d\rangle,\langle a,f\rangle,\langle a,b\rangle,\langle d,a\rangle,\langle d,d\rangle,\langle d,f\rangle,\langle d,b\rangle,\langle f,a\rangle,\langle f,d\rangle, \\ & \langle f,f\rangle,\langle f,b\rangle,\langle b,a\rangle,\langle b,d\rangle,\langle b,f\rangle,\langle b,b\rangle,\langle c,e\rangle,\langle e,c\rangle,\langle e,e\rangle,\langle c,c\rangle\} \end{aligned}$$

El $\mathrm{Dom}(R) = \{a,b,c,d,e,f\}$, por lo que puede tener hasta 6 clases de equivalencia, pero en este caso tiene solamente dos:

1. La clase de equivalencia de a, b, d y f es el conjunto $\{b,f,d,a\}$.
2. La clase de equivalencia de c y e es el conjunto $\{c,e\}$.

Formalmente decimos que si $R : A \to A$ es una relación de equivalencia en un conjunto A, la clase de equivalencia de un elemento $a \in A$ es

$$[a]_R = \{x \in A | \langle a,x\rangle \in R\}$$

Cualquier elemento $x \in [a]$ es un **representante** de la clase, y debe ser considerado de igual manera que los demás representantes de la misma clase.

Si R es una relación de equivalencia sobre un conjunto A, se tiene que:

- $aRb \to [a] = [b]$
- $[a] = [b] \to [a] \cap [b] \neq \varnothing$

Lo anterior es cierto porque, si suponemos que $c \in [a]$, entonces aRc, como también aRb y R es simétrica, sabemos que bRa, pero R también es transitiva, por lo que si $bRa \wedge aRc$ se tiene que bRc, por lo tanto $c \in [b]$. Esto demuestra que $[a] \subseteq [b]$. Para determinar que $[a] = [b]$, hace falta demostrar que $[b] \subseteq [a]$, pero esto es parte del ejercicio 8.

Ahora, $[a] = [b] \to [b] \cap [a] \neq \varnothing$. Supongamos que $[a] = [b]$. $[a] \cap \varnothing \neq \varnothing$ porque $[a]$ no es vacía,ya que tiene al menos al elemento a, y $a \in [a]$ porque R es reflexiva.

Finalmente supongamos que $[a] \cap [b] \neq \varnothing$. Entonces $\exists c \in \mathrm{dom}(R) : c \in [a] \wedge c \in [b]$. Como R es simétrica, cRb y por transitividad, como $aRc \wedge cRb$, se tiene aRb.

Por lo que aRb, $[a] = [b]$ y $[a] \cap [b] \neq \varnothing$ son expresiones equivalentes en una relación de equivalencia R.

7.2.3 Relación de orden parcial

Una de los problemas clásicos de computación es el ordenamiento de elementos. El ordenamiento es importante por varias razones:

1. Establece un criterio de turnos para ejecutar un proceso sobre los elementos de un conjunto.
2. Se tiene conocimiento sobre la totalidad de los elementos, cosa que es particularmente útil para determinar si un proceso ya ha agotado a todos los posibles candidatos.
3. Es posible delimitar búsquedas y así ahorrar tiempo de ejecución.

Definición 7.2.6 -- Relación de orden parcial. Sea $\preceq: A \to A$ una relación. $\preceq$ es un **orden parcial** si $\preceq$ es reflexiva, antisimétrica y transitiva [KB84]. Si $\preceq$ es un orden parcial sobre A, decimos que A es un **conjunto parcialmente ordenado** [poset] y lo denotamos por $(A;\preceq)$.

Ejemplo 7.20

El conjunto de los números enteros es un conjunto parcialmente ordenado con la relación $\leq$, porque:

1. $\leq$ es una relación reflexiva. $\forall a \in \mathbb{Z} : a \leq a$.
2. $\leq$ es una relación antisimétrica. $\forall a, b \in \mathbb{Z} : (a \leq b) \wedge (b \leq a) \to a = b$
3. $\leq$ es una relación transitiva. $\forall a, b, c \in \mathbb{Z} : (a \leq b) \wedge (b \leq c) \to a \leq c$

Es común denotar por $\preceq$ a cualquier relación de orden parcial. Este símbolo se utiliza generalmente cuando se quiere mencionar que la relación tiene las propiedades reflexiva, antisimétrica y transitiva. El símbolo $\preceq$ se parece mucho a $\leq$ que es el menor o igual, pero este último se reserva para los números.

Definición 7.2.7 -- Elementos comparables. Si $(A;\preceq)$ es un conjunto parcialmente ordenado y $a, b \in A$, decimos que a es **comparable** con b si $a \preceq b$.

Ejemplo 7.21

Sea $A = \{1,2,3,4\}$ y $\preceq \leftrightarrow \{\langle 1,4\rangle, \langle 3,1\rangle, \langle 6,4\rangle, \langle 2,4\rangle, \langle 2,6\rangle, \langle 5,2\rangle, \langle 1,6\rangle, \langle 3,3\rangle, \langle 6,6\rangle, \langle 2,2\rangle, \langle 5,5\rangle, \langle 1,1\rangle, \langle 4,4\rangle, \langle 3,4\rangle, \langle 3,6\rangle, \langle 5,4\rangle, \langle 5,6\rangle\}$ un poset.

4 es comparable a 4.
6 es comparable a 4 y 6.
1 es comparable a 1, 4 y 6.
2 es comparable a 2, 4 y 6.
5 es comparable a 2, 5, 4 y 6.

Para visualizar un poset se utiliza un **diagrama de Hasse** [Ros04, KB84], que resume la información acerca de la relación. El diagrama de Hasse de la relación del ejemplo 7.21 se muestra en la figura 7.2.

Un diagrama Hasse para una relación $R : A \to A$ se construye del siguiente modo:

1. Los elementos de A se colocan en círculos. También se puede utilizar puntos grandes en lugar de círculos, colocando el nombre como una etiqueta.
2. Borrar de R todos los pares reflexivos. Esta información es redundante pues debe ser obvio que un elemento se relaciona consigo mismo.
3. Quitar de R todos los pares transitivos, es decir, si $\langle a,b\rangle \in \preceq$ y también $\langle b,c\rangle \in \preceq$, entonces se puede remover $\langle a,c\rangle$.
4. Si $a \preceq b$, se debe dibujar una línea desde el círculo etiquetado con a hasta el círculo etiquetado con b, teniendo cuidado de dibujar a un poco más abajo que b.

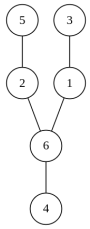


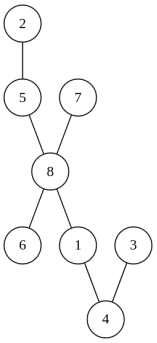
Figura 7.2: *Diagrama Hasse del poset $(A, \preceq)$ del ejemplo 7.21*

En un diagrama de Hasse, si hay una línea que une un elemento con otro, en dirección de abajo hacia arriba, entonces son elementos comparables. Un diagrama Hasse es útil para ver alguna información que es difícil ver en la lista de pares, por ejemplo que 5 no es comparable ni con 3, ni con 1 ya que no hay un camino que los conecte desde abajo hacia arriba.

Ejemplo 7.22

Sea $(A, \preceq)$ un conjunto parcialmente ordenado sobre el conjunto $A \leftarrow \{1,2,3,4,5,6,7,8\}$ y la relación $\preceq \leftarrow \{\langle 4,8\rangle, \langle 4,1\rangle, \langle 1,2\rangle, \langle 5,2\rangle, \langle 8,7\rangle, \langle 4,3\rangle, \langle 1,8\rangle, \langle 6,8\rangle, \langle 1,5\rangle, \langle 8,5\rangle, \langle 5,5\rangle, \langle 4,4\rangle, \langle 6,6\rangle, \langle 1,1\rangle, \langle 8,8\rangle, \langle 2,2\rangle, \langle 7,7\rangle, \langle 3,3\rangle, \langle 4,7\rangle, \langle 4,5\rangle, \langle 4,2\rangle, \langle 1,7\rangle, \langle 6,7\rangle, \langle 6,5\rangle, \langle 8,2\rangle, \langle 6,2\rangle\}$

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



En la relación $7 \not\preceq 3$ y también $3 \not\preceq 7$, por lo que 3 y 7 son elementos no comparables. Los elementos 8 y 6 son comparables, y también lo son 2 con 6.

Un programa que verifica que una relación dada de forma extensional sea una relación de orden parcial es el siguiente:

Código 7.8: *Verifica un conjunto parcialmente ordenado*

```
1 def esROP(R:list, D = None)-> bool:
2     """
3     Determina si una relación es parcialmente ordenada
4     """
5     if D is None: D = union(dom(R), ran(R))
6     return paraTodo(lambda P: P(R), [esRef, esAntisim, esTran])
```

Antes de ejecutar este código se deben tener definidas las funciones `esRef` [código 7.1, página 148], `esAntisim` [código 7.5, página 152] y `esTran` [ver ejercicio 3, en la página 166].

7.3 Operaciones con relaciones

Como las relaciones son esencialmente conjuntos, es posible utilizar operadores de conjuntos para combinar relaciones, pero hay que tener cuidado con la interpretación del resultado.

Consideremos como ejemplo $R_1 : Personas \to Libros$ la relación «una persona $a \in Personas$ está relacionada con un libro $b \in Libros$ si a **ya ha leído** el libro b». Por otro lado, consideremos $R_2 : Personas \to Libros$ como la relación «una persona a está relacionada con un libro b si la persona a **quiere leer** el libro y». ¿Cómo puede interpretarse el resultado de la operación $R_1 \cup R_2$?

Como $R_1 \cup R_2$ es de hecho una lista de pares en $Personas \times Libros$, es también una relación, que con pares $\langle a, b \rangle \in Personas \times Libros$ que puede ser interpretada como la persona a ha leído o quiere leer el libro b.

Los operadores de unión, intersección y diferencia, son operadores que comúnmente se utilizan al combinar relaciones.

7.3.1 Relación inversa

Definición 7.3.1 -- Relación inversa. Si $R : A \to B$ es una relación, la relación inversa de R se escribe R^{-1} y es una relación $R^{-1} : B \to A$ donde

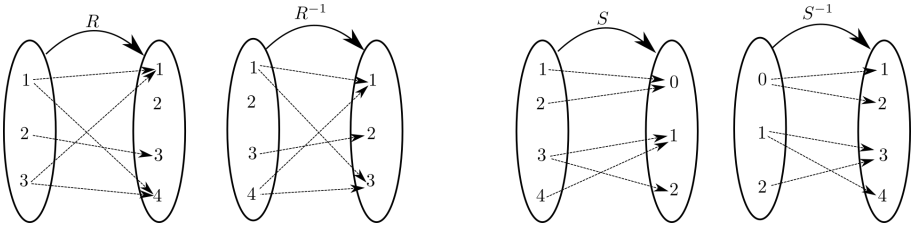
$$R^{-1} = \{\langle b, a \rangle \in B \times A \mid \langle a, b \rangle \in R\}$$

La inversa de una relación colecciona los pares inversos de una relación. Es decir, si $\langle a, b \rangle \in R$, entonces en la relación inversa de R debería estar $\langle b, a \rangle$. De este modo $|R| = |R^{-1}|$. Si $\langle a, b \rangle \in R$ y también $\langle a, b \rangle \in R^{-1}$, significa que $a = b$.

Ejemplo 7.23

Si $R = \{\langle 1,1 \rangle, \langle 3,1 \rangle, \langle 1,4 \rangle, \langle 2,3 \rangle, \langle 3,4 \rangle\}$ y $S = \{\langle 1,0 \rangle, \langle 2,0 \rangle, \langle 3,1 \rangle, \langle 3,2 \rangle, \langle 4,1 \rangle\}$, la relación inversa de R y la de S es respectivamente:

$R^{-1} = \{\langle 1,1 \rangle, \langle 1,3 \rangle, \langle 4,1 \rangle, \langle 3,2 \rangle, \langle 4,3 \rangle\}$, $S^{-1} = \{\langle 0,1 \rangle, \langle 0,2 \rangle, \langle 1,3 \rangle, \langle 2,3 \rangle, \langle 1,4 \rangle\}$



La relación inversa es una tarea sencilla, pues hay que obtener la tupla inversa de cada tupla en la relación.

Código 7.9: *Relación inversa*

```
1 def rInv(R:list)-> list:
2     """
3     Obtiene la relación inversa de una relación
4     """
5     return enCada(lambda t: tInversa(t), R)
```

7.3.2 Composición de relaciones

Definición 7.3.2 -- Composición de relaciones. Si $R : A \to B$ y $S : B \to C$ son dos relaciones. La **composición** o producto de R con S se escribe $(R \circ S)$ y es

$$(R \circ S) = \{\langle a,c\rangle \in A \times C \mid \exists b \in B : \langle a,b\rangle \in R \wedge \langle b,c\rangle \in S\}$$

En la relación $(R \circ S)$ se tiene que:

1. $\texttt{dom}(R \circ S) = \texttt{dom}(R)$.
2. $\texttt{ran}(R \circ S) = \texttt{Im}(S, \texttt{ran}(R))$.
3. Si $a \in \texttt{dom}(R)$, entonces $\texttt{im}((R \circ S), a) = \texttt{Im}(R, \texttt{im}(S, a))$. Esto significa que habrá un par $\langle a,c\rangle$ para cada $c \in \texttt{Im}(R, \texttt{im}(S, a))$.

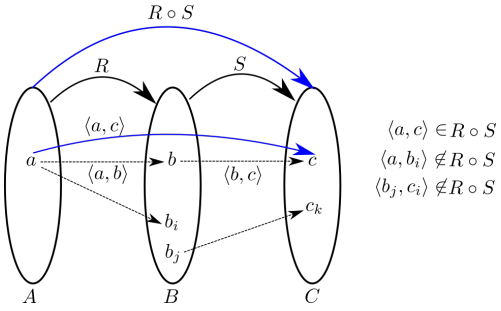


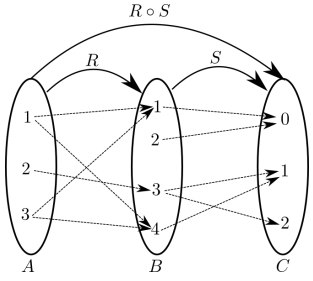
Figura 7.3: *Composición de relaciones*

Observa que, para que un par $\langle a,c\rangle$ pertenezca a la composición $R \circ S$, es necesario que un elemento en el rango de R, sirva de «puente» para alcanzar al elemento c en el rango de S.

En la figura establece un caso en que hay un par como el $\langle a, b_i\rangle$, que pertenece a la relación R, pero no hay elemento en el dominio de S que sirva de puente para alcanzar a algún elemento en C. Por otro lado, también puede haber algún par, como $\langle b_j, c_k\rangle$ en S, pero como no hay elemento en $\texttt{dom}(R)$ que tenga una imagen en el dominio de S, por lo que tampoco sirve para la composición.

Ejemplo 7.24

Si $R = \{\langle 1,1\rangle, \langle 3,1\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 3,4\rangle\}$ y $S = \{\langle 1,0\rangle, \langle 2,0\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle\}$, la composición $(R \circ S) = \{\langle 3,1\rangle, \langle 3,0\rangle, \langle 2,2\rangle, \langle 2,1\rangle, \langle 1,1\rangle, \langle 1,0\rangle\}$



Nota que el par $\langle 3,1\rangle \in (R \circ S)$ porque $4 \in \texttt{ran}(R)$ y $4 \in \texttt{dom}(S)$ sirve de puente para alcanzar a $1 \in \texttt{ran}(S)$. Por otro lado, $\texttt{im}(R,3) \mapsto \{1,4\}$, luego $\texttt{Im}(S,\{1,4\}) \mapsto \{0,1\}$, por lo que $\langle 3,0\rangle$ y $\langle 3,1\rangle$ deben pertenecer a la composición R con S.

Código 7.10: *Composición de relaciones*

```
def rComp(R:list, S:list)-> list:
    """
    La composición de la función R con S
    """
    RS = enCada(lambda a:enCada(lambda c: tupla(a,c),
                                Im(S,im(R,a))),
                dom(R))
    return Union(*RS)
```

Ejemplo 7.25

Utiliza la herramienta `rComp` para calcular la composición $(R \circ S)$ del ejemplo 7.24 En el ejemplo 7.24, las relaciones son $R = \{\langle 1,1\rangle, \langle 3,1\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 3,4\rangle\}$ y $S = \{\langle 1,0\rangle, \langle 2,0\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle\}$, por lo que podemos hacer:

```
>>> R = conj(tupla(1,1), tupla(3,1), tupla(1,4), tupla(2,3), tupla(3,4))
>>> print(R)
[[1, 1], [3, 1], [1, 4], [2, 3], [3, 4]]
>>> S = conj(tupla(1,0), tupla(2,0), tupla(3,1), tupla(3,2), tupla(4,1))
>>> print(S)
[[1, 0], [2, 0], [3, 1], [3, 2], [4, 1]]
>>> RoS = rComp(R,S)
>>> print(RoS)
[[3, 0], [3, 1], [1, 0], [1, 1], [2, 1], [2, 2]]
>>>
```

7.3.3 Potencia de relaciones

La potencia de una relación se puede definir de manera recursiva en base a la composición de funciones.

Definición 7.3.3 -- Potencia de una relación. Digamos que $R : A \to A$ es una relación. R^n con $n = 1, 2, 3, \ldots$ se define recursivamente como

$$R^n = \begin{cases} \text{Si } n = 1 & , R \\ \text{Si } n > 1 & , R \circ R^{n-1}. \end{cases}$$

La n-ésima potencia de una relación R, es la aplicación de la operación de composición de R con sí misma, n veces, se tiene entonces que:

R^1 es R.

R^2 es aplicar la composición de R con sí misma, es decir $(R \circ R)$.

R^3 es aplicar la composición de R con R^2, esto es, $(R \circ (R \circ R))$.

Si M_R es la matriz de pertenencia de la relación R, la n-ésima potencia de una relación se obtiene al calcular el producto de la matriz por sí misma, n veces.

Por ejemplo, si $R \leftarrowtail \{\langle 1,1\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 3,1\rangle, \langle 4,2\rangle\}$, R^2 se obtiene al multiplicar la matriz de pertenencia de R por sí misma, con lo que se obtiene $R^2 = \{\langle 1,1\rangle, \langle 1,4\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 3,1\rangle, \langle 3,4\rangle, \langle 4,3\rangle\}$.

$$\begin{array}{c|cccc} R & 1 & 2 & 3 & 4 \\ 1 & 1 & \cdot & \cdot & 1 \\ 2 & \cdot & \cdot & 1 & \cdot \\ 3 & 1 & \cdot & \cdot & \cdot \\ 4 & \cdot & 1 & \cdot & \cdot \end{array} \cdot \begin{array}{c|cccc} R & 1 & 2 & 3 & 4 \\ 1 & 1 & \cdot & \cdot & 1 \\ 2 & \cdot & \cdot & 1 & \cdot \\ 3 & 1 & \cdot & \cdot & \cdot \\ 4 & \cdot & 1 & \cdot & \cdot \end{array} \mapsto \begin{array}{c|cccc} R^2 & 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & \cdot & 1 \\ 2 & 1 & \cdot & \cdot & \cdot \\ 3 & 1 & \cdot & \cdot & 1 \\ 4 & \cdot & \cdot & 1 & \cdot \end{array}$$

Ejemplo 7.26

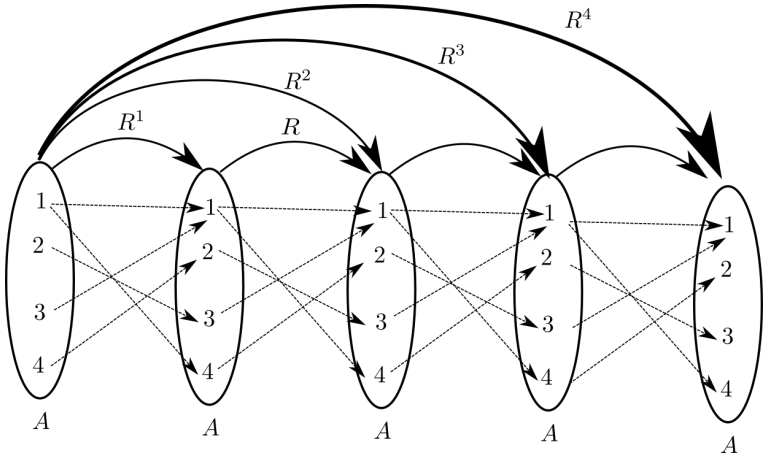
Si $R = \{\langle 1,1\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 3,1\rangle, \langle 4,2\rangle\}$, calcula R^2, R^3, R^4:

$R^2 = (R \circ R)$ ya fue calculado. $R^3 = (R \circ R^2)$ que es la composición de R con $\{\langle 1,1\rangle, \langle 1,4\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 3,1\rangle, \langle 3,4\rangle, \langle 4,3\rangle\}$,

$R^2 = \{\langle 1,1\rangle, \langle 1,4\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 3,1\rangle, \langle 3,4\rangle, \langle 4,3\rangle\}$

$R^3 = \{\langle 1,1\rangle, \langle 1,4\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,1\rangle, \langle 2,4\rangle, \langle 3,1\rangle, \langle 3,4\rangle, \langle 3,2\rangle, \langle 4,1\rangle\}$

$R^4 = \{\langle 1,1\rangle, \langle 1,4\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,1\rangle, \langle 2,4\rangle, \langle 2,2\rangle, \langle 3,1\rangle, \langle 3,4\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 4,1\rangle, \langle 4,4\rangle\}$



Las leyes de las potencias en los números, también tienen una interpretación en las relaciones:

$R^{m+n} = (R^m \circ R^n)$

$R^{mn} = (R^m)^n$

$(R^n)^{-1} = (R^{-1})^n$

Así por ejemplo, si $m \leftarrowtail 1$ y $n \leftarrowtail 1$, se tiene que $R^{m+n} = R^{1+1} = R^2 = (R^1 \circ R^1) \mapsto (R \circ R)$. Cuando $n \leftarrowtail 1$ y $m \leftarrowtail 2$ se tiene $R^{1+2} = R^3 = (R \circ R^2) \mapsto (R \circ (R \circ R))$ y así en adelante.

Ejercicios

1. Utiliza la matriz de pertenencia para verificar que las siguientes relaciones sean reflexivas. En estos ejercicios el dominio es $\{1,2,3,4,5,6,7,8\}$:
   a) $R_1 \leftarrow \{\langle 6,5\rangle, \langle 5,5\rangle, \langle 3,8\rangle, \langle 8,1\rangle, \langle 8,4\rangle, \langle 4,2\rangle, \langle 7,6\rangle, \langle 6,8\rangle, \langle 7,2\rangle, \langle 1,2\rangle\}$.
   b) $R_2 \leftarrow \{\langle 3,1\rangle, \langle 4,4\rangle, \langle 6,3\rangle, \langle 8,8\rangle, \langle 5,6\rangle, \langle 1,1\rangle, \langle 6,6\rangle, \langle 1,6\rangle, \langle 8,1\rangle, \langle 6,5\rangle, \langle 5,5\rangle, \langle 3,8\rangle, \langle 4,6\rangle, \langle 3,3\rangle\}$.
2. Utiliza el programa `esSim` [código 7.3, página 150] para verificar que las siguientes relaciones sean simétricas.
   a)

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

   b)

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

3. Escribe una función en `Python` llamada `esTran` que reciba como entrada de manera obligatoria una relación `R` en forma de una lista de pares y de manera opcional un conjunto `D` que será el conjunto dominio. El programa debe determinar si la relación es transitiva o no lo es, considerando el dominio `D`. El programa debe devolver `True` en caso de que `R` sea transitiva en el dominio `D` y `False` si no es transitiva. Si el dominio `D` no se proporciona, el dominio debe ser calculado como la unión del dominio de la relación con el rango, mismos que son tomados de la lista de pares dada en `R`.

```
>>> esTran([[3,3],[4,4],[5,5],[1,5],[1,2],[2,1],[2,5],[2,4],[2,3],[2,2],[1,4],
[1,3],[1,1],[3,5],[5,3],[4,1],[4,5],[4,3],[4,2]])
True
>>> esTran([[3,3],[4,4],[5,5],[1,5],[1,2],[2,1],[2,5],[2,4],[2,3],[2,2],[1,4],
[1,3],[1,1],[3,5],[5,3],[4,1],[4,5],[4,3],[4,2]], [1,2,3,4,5])
True
>>>
```

4. Escribe el programa llamado `propiedadesDe` que recibe una relación `R` en forma de lista de pares. El programa debe devolver un diccionario de `Python`, teniendo como claves las palabras `"Reflexiva"`, `"Irreflexiva"`, `"Simétrica"`, `"Asimétrica"`, `"Antisimétrica"` y `"Transitiva"`; como valor de cada clave un booleano `True` o `False` en dependencia si tiene o no tal propiedad.

```
>>> propiedadesDe([[1,2],[3,2],[4,2],[3,1]])
{'Reflexiva': False, 'Irreflexiva': True, 'Simétrica': False,
'Asimétrica': True, 'Antisimétrica': True, 'Transitiva': True}
>>>
```

Ayuda: Si no sabes cómo crear diccionarios en `Python`, puedes revisar en web `https://www.w3schools.com/python/python_dictionaries.asp`

5. Determina si las siguientes relaciones tienen algunas de las propiedades reflexiva, irreflexiva, simétrica, asimétrica, antisimétrica o transitiva. Marca la casilla con una ✓o ✗ si una relación tiene tal propiedad. Para este ejercicio considera el dominio $A = \{1,2,3,4,5,6,7,8\}$
   a) $\{\langle 6,5\rangle,\langle 6,4\rangle,\langle 8,6\rangle,\langle 1,7\rangle,\langle 8,3\rangle,\langle 8,5\rangle,\langle 1,6\rangle,\langle 7,6\rangle,\langle 4,8\rangle,\langle 7,2\rangle,\langle 1,8\rangle,\langle 6,3\rangle,\langle 7,5\rangle,\langle 3,4\rangle,\langle 5,3\rangle,\langle 2,5\rangle,\langle 2,8\rangle,\langle 2,3\rangle,\langle 7,8\rangle,\langle 7,4\rangle,\langle 2,4\rangle,\langle 4,5\rangle,\langle 1,5\rangle\}$.
   b) $\{\langle 5,7\rangle,\langle 4,2\rangle,\langle 3,7\rangle,\langle 7,3\rangle,\langle 6,5\rangle,\langle 8,7\rangle,\langle 5,6\rangle,\langle 3,8\rangle,\langle 4,4\rangle,\langle 6,2\rangle,\langle 8,6\rangle,\langle 6,8\rangle,\langle 6,6\rangle,\langle 8,1\rangle,\langle 2,6\rangle,\langle 5,4\rangle,\langle 1,3\rangle,\langle 4,3\rangle,\langle 5,5\rangle,\langle 3,3\rangle,\langle 2,4\rangle,\langle 1,4\rangle,\langle 2,5\rangle,\langle 4,8\rangle,\langle 8,4\rangle,\langle 5,2\rangle,\langle 3,1\rangle,\langle 4,1\rangle,\langle 7,5\rangle,\langle 8,8\rangle,\langle 7,8\rangle,\langle 4,5\rangle,\langle 3,4\rangle,\langle 1,8\rangle,\langle 8,3\rangle\}$.
   c) $\{\langle 6,4\rangle,\langle 1,6\rangle,\langle 3,3\rangle,\langle 8,3\rangle,\langle 5,2\rangle,\langle 3,8\rangle,\langle 6,5\rangle,\langle 3,6\rangle,\langle 1,7\rangle,\langle 4,5\rangle,\langle 1,4\rangle,\langle 1,5\rangle,\langle 8,8\rangle,\langle 8,6\rangle,\langle 6,2\rangle,\langle 3,4\rangle,\langle 3,5\rangle,\langle 3,2\rangle,\langle 4,2\rangle,\langle 1,2\rangle,\langle 8,4\rangle,\langle 8,5\rangle,\langle 8,2\rangle\}$.
   d) $\{\langle 3,3\rangle,\langle 4,1\rangle,\langle 4,6\rangle,\langle 7,5\rangle,\langle 5,7\rangle,\langle 1,7\rangle,\langle 5,1\rangle,\langle 6,4\rangle,\langle 5,4\rangle,\langle 7,1\rangle,\langle 1,2\rangle,\langle 5,6\rangle,\langle 6,2\rangle,\langle 3,4\rangle,\langle 7,2\rangle,\langle 7,4\rangle,\langle 7,7\rangle,\langle 6,5\rangle,\langle 7,6\rangle,\langle 3,6\rangle,\langle 3,7\rangle,\langle 6,6\rangle,\langle 1,4\rangle,\langle 4,4\rangle,\langle 3,2\rangle,\langle 1,6\rangle,\langle 3,5\rangle,\langle 5,2\rangle,\langle 4,7\rangle,\langle 5,5\rangle,\langle 1,5\rangle,\langle 3,1\rangle,\langle 4,5\rangle,\langle 1,1\rangle,\langle 2,2\rangle,\langle 6,1\rangle,\langle 4,2\rangle,\langle 6,7\rangle\}$

Rel	Reflexiva	Irreflexiva	Simétrica	Asimétrica	Antisimétrica	Transitiva
(a)						
(b)						
(c)						
(d)						
(e)						

6. Considera las relaciones $R = \{\langle 1,2\rangle,\langle 2,3\rangle,\langle 3,4\rangle\}$ y $S = \{\langle 1,1\rangle,\langle 1,2\rangle,\langle 2,1\rangle,\langle 2,2\rangle,\langle 2,3\rangle,\langle 3,1\rangle,\langle 3,2\rangle,\langle 3,3\rangle,\langle 3,4\rangle\}$ relaciones con dominio y rango en $\{1,2,3,4\}$. Calcula
   a) $R \cup S$
   b) $R \cap S$
   c) $S \backslash R$
7. Verifica que las siguientes relaciones sean relaciones de equivalencia:
   a)

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & \cdot & 1 & \cdot & 1 \\ 1 & 1 & 1 & \cdot & \cdot \\ 1 & \cdot & 1 & \cdot & 1 \\ \cdot & 1 & 1 & 1 & 1 \\ 1 & \cdot & \cdot & 1 & \cdot \end{pmatrix} \end{array}$$

   b)

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 1 & \cdot & 1 & 1 \\ 1 & 1 & \cdot & 1 & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & 1 & 1 \\ 1 & 1 & \cdot & 1 & 1 \end{pmatrix} \end{array}$$

8. Demuestra que si R es una relación de equivalencia, $aRb \rightarrow [a] = [b]$. Revisa nuevamente la página 159.
9. Escribe una función recursiva de cola en `Python` llamada `rPot`, que reciba como entrada obligatoria una relación `R` como una lista de pares: y un número entero positivo `n`. Opcionalmente puedes trabajar con un parámetro `RES` para llevar el resultado en cada recursión. El programa debe devolver la lista de pares que corresponde a la `n`-ésima potencia de `R`.

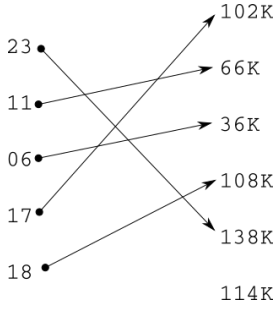
```
>>> rPot([[1,1], [1,4], [2,3], [3,1], [3,4]], 2)
[[3, 1], [3, 4], [1, 1], [1, 4], [2, 1], [2, 4]]
>>> rPot([[1,1], [1,4], [2,3], [3,1], [3,4]], 3)
[[3, 1], [3, 4], [1, 1], [1, 4], [2, 1], [2, 4]]
>>>
```

Funciones 8

8.1 Concepto fundamental

Supongamos que se tiene una colección de personas que trabajan en cierto proyecto, y que ya tienen algún tiempo trabajando allí. A cada persona se le debe pagar una cantidad en miles de pesos, que depende solamente del tiempo en que han estado laborando. La asignación es como se muestra:

Id	Nombre	Tiempo-meses
01	Arturo	23
02	Amelia	11
03	Carmen	06
04	Gustavo	17
05	Jackeline	18



La tabla de la izquierda muestra la información de cada empleado, se podrían agregar más columnas para agregar la información que sea necesaria, pero la información que determina la cantidad que se debe pagar, es la que se encuentra en la columna `Tiempo-meses`. La asignación de la derecha asocia una cantidad de meses con una y sólo una cantidad de dinero, lo que se muestra en la relación de la derecha.

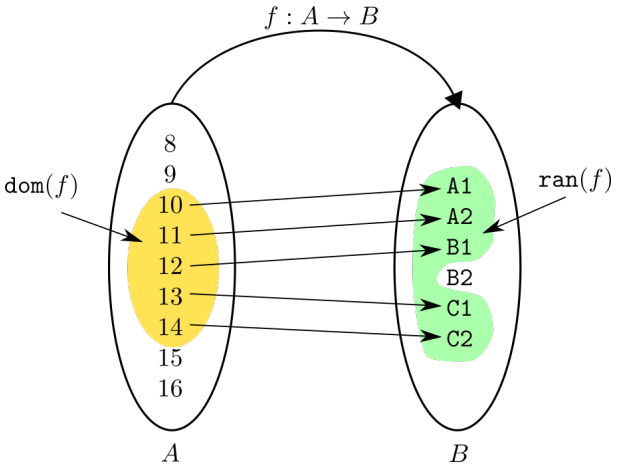
La situación anterior es una situación que se puede modelar mediante una función. El concepto de función es muy importante para matemáticas discretas, y especialmente en programación de computadoras. Las funciones están presentes en todos los programas de computadoras y son la base para el desarrollo teórico de las matemáticas y del desarrollo de software.

Definición 8.1.1 -- Función. Si A y B son conjuntos, con $f : A \to B$ una relación de A en B. Decimos que f es una función, si cada elemento del dominio se asocia con exactamente un elemento del rango.

Aunque una función $f : A \to B$ toma elementos del conjunto A y los asocia con elementos del conjunto B, es posible que no todos los elementos de A o no todos los elementos de B participen en la definición de la función.

Ejemplo 8.1

Se tiene un conjunto de niños con las siguientes edades $A \leftarrow \{8,9,10,11,12,13,14,15,16\}$. Por otro lado se tienen 5 clases $B \leftarrow \{\texttt{A1},\texttt{A2},\texttt{B1},\texttt{B2},\texttt{C1},\texttt{C2}\}$, con lo que crearemos la siguiente relación $f : A \to B$ definida $f \leftarrow \{\langle 10,\texttt{A1}\rangle,\langle 11,\texttt{A2}\rangle,\langle 12,\texttt{B1}\rangle,\langle 13,\texttt{C1}\rangle,\langle 14,\texttt{C2}\rangle\}$.



$\texttt{dom}(f) \mapsto \{10,11,12,13,14\}$ y $\texttt{ran}(f) \mapsto \{\texttt{A1},\texttt{A2},\texttt{B1},\texttt{C1},\texttt{C2}\}$. Cada elemento del $\texttt{dom}(f)$ se asocia con exactamente un elemento de su rango, entonces f es una función.

8.1.1 La firma de una función

Al definir una función, es necesario expresar los conjuntos de referencia, esto se hace mediante la firma de la función, como en $f : A \to B$. Esta firma ofrece la siguiente información:

1. El nombre de la función. En este caso es f.
2. El conjunto de referencia A, donde $\texttt{dom}(f) \subseteq A$.
3. El conjunto de referencia B, donde $\texttt{ran}(f) \subseteq A$.

Ejemplo 8.2

En los lenguajes de programación, especialmente en aquellos que son fuertemente tipificados, es común crear funciones mencionando su identificador y los conjuntos de referencia. Así la firma $suelo : \mathbb{R} \to \mathbb{Z}$, que se refiere a una función con nombre `suelo`, que recibe como entrada un valor del conjunto de reales y devuelve un elemento del conjunto de los enteros, puede ser traducida a diferentes lenguajes de programación como:

- En `Java` se escribe `int suelo(float x)`
- En `Pascal` se escribe `function suelo(x: real): integer`
- En `Python` se escribe `def suelo(x:float)->int:`

8.1.2 Verificación de función

Para verificar que una relación sea una función, basta verificar que la imagen de todos los elementos del dominio, sean unitarias. Esto significa que cada elemento del dominio está asociado con exactamente un elemento en el rango.

Código 8.1: *Verifica que una relación sea una función*

```
def esFun(f:list, A:list=None, B:list=None)-> bool:
    """
    Verifica que f sea una función
    """
    if A is None: A = dom(f)
    if B is None: B = ran(f)
    return paraTodo(lambda a: esSoliton(im(f,a)), A)
```

Por supuesto que es necesario que f sea en principio una relación de A en B, esto es un subconjunto del producto cartesiano de A con B.

Ejemplo 8.3

Revisa que la relación del ejemplo 8.1 sea una función.

```
>>> f = [[10,'A1'], [11,'A2'], [12,'B1'], [13,'C1'], [14,'C2']]
>>> esFun(f)
True
>>>
```

Revisa ahora que la relación $f \hookleftarrow \{\langle 10,\texttt{A1}\rangle,\langle 10,\texttt{C1}\rangle,\langle 11,\texttt{A2}\rangle,\langle 12,\texttt{B1}\rangle,\langle 13,\texttt{C1}\rangle,\langle 14,\texttt{C2}\rangle\}$ sea una función.

```
>>> f = [[10,'A1'], [10,'C1'], [11,'A2'], [12,'B1'], [13,'C1'], [14,'C2']]
>>> esFun(f)
False
>>>
```

El segundo ejemplo no es una función puesto que $\texttt{im}(f,10)$ no es unitaria. Observa que 10 está relacionado con `A1` y con `C1`.

8.1.3 Definición intencional o extensional

Como una función es en esencia un conjunto, por lo que puede ser definida de manera intencional o de forma extensional. Al definir la función de manera intencional, es necesario proporcionar una regla que asocie cada elemento del dominio de definición a su correspondiente elemento del rango.

Ejemplo 8.4

Sea f la función definida para todos los números enteros en el rango de 1 a 10 inclusive, que transforma cada elemento en el número que resulta de multiplicar el número por sí mismo.

$$f \hookleftarrow \{\langle a,b\rangle | a \in [1,\ldots,10], b = a^2\}$$

Aquí el dominio de definición es el conjunto de números $\{1,2,3,4,5,6,7,8,9,10\}$, el rango es $\{1,4,9,16,25,36,49,64,81,100\}$.

Una función definida de manera extensional debe hacer explícito cada par $\langle a,b\rangle$ que asocia cada elemento a del dominio a su correspondiente elemento b del codominio. El domino de definición se debe obtener de la primera entrada de cada par en el conjunto, y el codominio de la última entrada de cada par.

Ejemplo 8.5

Sea f la función definida para los números enteros 1,2,3,4,5,6,7,8,9 y 10, donde:

$$f \leftarrow \{\langle 1,1\rangle, \langle 2,4\rangle, \langle 3,9\rangle, \langle 4,16\rangle, \langle 5,25\rangle, \langle 6,36\rangle, \langle 7,49\rangle, \langle 8,64\rangle, \langle 9,81\rangle, \langle 10,100\rangle\}.$$

Aquí el dominio de definición es el conjunto de números $\{1,2,3,4,5,6,7,8,9,10\}$, el rango es $\{1,4,9,16,25,36,49,64,81,100\}$.

8.2 Evaluación y preimagen

Las funciones son como máquinas que transforman entradas en salidas, podemos decir que la función f **transforma** o **mapea** elementos del dominio de la función, en elementos de su rango. Cada entrada de la función es un elemento del dominio, la única salida permitida para tal entrada es un elemento del rango de la función. Otros nombres que son sinónimos de la palabra «función» son «transformación» o «mapeo».

8.2.1 Evaluación de un elemento del dominio

Definición 8.2.1 Si $f : A \to B$ es una función, con $\langle a,b\rangle \in f$, escribimos $f(a) = b$ y decimos que b es la **evaluación** de a bajo la función f, al mismo tiempo a es una **preimagen** de b.

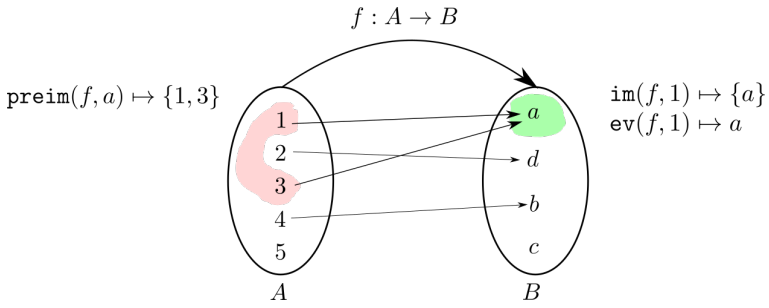


Figura 8.1: *Imagen y evaluación de un elemento del dominio de una función.*

Observa en la figura 8.1 que la imagen de un elemento del dominio de una función, siempre es un subconjunto unitario y la evaluación de ese elemento en la función, es un elemento del rango.

Código 8.2: *Evaluación de un elemento del dominio de una función*

```
def ev(f:list, x):
    """
    Calcula la evaluación de la función en un elemento del dominio.
    """
    y = im(f,x)
    if esSoliton(y):
        return car(y)
    elif esVacio(y):
        return "NoDef"
    else:
        return "NoFun"
```

Por comodidad, si f es una función, podemos «tolerar» que el concepto de imagen de un elemento sea un sinónimo de la evaluación de ese elemento. Esto es así porque la imagen de un elemento en la función, siempre es un conjunto unitario, cuyo único elemento es la evaluación de tal elemento en la función.

Ejemplo 8.6

Digamos que $f(x) = 2x^3 + 3x + 1$. La evaluación de la función f en 7, significa saber qué elemento del rango de la función está asociado con 7. Para esto simplemente sustituimos el valor 7 en cada ocurrencia de x de la definición de la función:

$$f(7) = 2(7)^2 + 3(7) + 1 = 2(49) + 2(7) + 1 = 98 + 14 + 1 = 113$$

Por lo que el par $\langle 7, 113 \rangle$ pertenece a la función f.

Ejemplo 8.7

Ahora supongamos que $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ es una función con dominio en $\mathbb{Z} \times \mathbb{Z}$ y rango en $\mathbb{Z}$, definida como $f(x, y) = x^2y + xy^2$. Debemos calcular la evaluación de $\langle 5, 4 \rangle$. También debemos sustituir en la expresión que define f los valores 5 para x y 4 para y:

$$f(5, 4) = (5)^2 \cdot 4 + 5 \cdot (4)^2 = 25 \cdot 4 + 5 \cdot 16 = 180$$

Por lo que la tupla $\langle \langle 5, 4 \rangle, 180 \rangle$ está en f. Con el fin de simplificar la notación se ha convenido escribir $\langle 5, 4, 180 \rangle$ en lugar de $\langle \langle 5, 4 \rangle, 180 \rangle$. Esta simplificación obliga a convenir que la evaluación de la función se encuentra en el último lugar de la tupla.

8.2.2 Preimagen de un elemento del rango

La preimagen de un elemento del rango de la función, siempre es un conjunto. En la preimagen se coleccionan todos los elementos del dominio que mapean al mismo elemento en el rango.

Definición 8.2.2 -- Preimagen. Sea $f : A \to B$. La preimagen de $b \in B$ se denota $\texttt{preim}(f, b)$ y es el conjunto $\{a \in A \mid f(a) = b\}$.

Código 8.3: Preimagen de un elemento del rango de una función

```
1 def preim(f:list, b)-> list:
2     """
3     Calcula la preimagen de un elemento del rango
4     """
5     return subc(lambda a: en(tupla(a,b),f))
```

La preimagen de un elemento del rango de una función se emplea para determinar qué valores del dominio se relacionan con el elemento del rango en cuestión.

Ejemplo 8.8

Sea $g \leftarrowtail \{\langle pedro, 22 \rangle, \langle arturo, 29 \rangle, \langle marco, 29 \rangle, \langle jaime, 28 \rangle, \langle karen, 27 \rangle, \langle lupita, 23 \rangle, \langle esteban, 30 \rangle, \langle fabiola, 26 \rangle, \langle ana, 30 \rangle\}$ una función que colecciona información acerca de personas y la edad de las personas, de modo que si $\langle a, n \rangle \in g$ es un par en la función, la persona a tiene $g(a) \mapsto n$ años de edad.

La preimagen de 29 es $\texttt{preim}(g, 29) \mapsto \{arturo, marco\}$. Este resultado se interpreta como el conjunto de personas que tienen 29 años de edad.

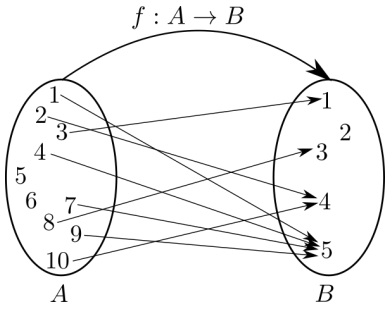
8.3 Clasificación de funciones

8.3.1 Funciones parciales y totales

Cuando en una función $f : A \to B$, el $\mathrm{dom}(f)$ es un subconjunto propio del conjunto A, decimos que la función es **parcialmente definida** o simplemente *parcial*; de otro modo, cuando $\mathrm{dom}(f) = A$, decimos que la función es **totalmente definida** o simplemente *total*.

Ejemplo 8.9

La función $f : \{1,\ldots,10\} \to \{1,2,3,4,5\}$, definida explícitamente como $f \leftarrowtail \{\langle 4,5\rangle, \langle 9,5\rangle, \langle 3,1\rangle, \langle 8,3\rangle, \langle 7,5\rangle, \langle 10,4\rangle, \langle 1,5\rangle, \langle 2,4\rangle\}$ es una función parcial.



Observa como no todos los elementos del conjunto de referencia A son mapeados a algún elemento del rango de la función.

En programación de computadoras, es frecuente el uso de funciones parciales, esto es porque a la hora de definir funciones en el lenguaje de programación, se utilizan tipos de datos predefinidos, cada tipo definido es un conjunto de valores, de los que solamente se utiliza un subconjunto de ellos como dominio de la función.

Ejemplo 8.10

La función $\mathtt{inc} : \mathbb{Z} \to \mathbb{Z}$ que está definida como $\mathtt{inc}(x) \mapsto x+1$ es la función llamada «incremento», que toma un valor entero y devuelve un valor entero que es mayor en una unidad respecto del valor de entrada.

Esta es una función total, ya que cualquier valor de entrada, tiene relacionado un elemento del rango.

Ejemplo 8.11

Se está programando una función en un lenguaje de programación que es fuertemente tipificado [los tipos de datos se resuelven en tiempo de compilación]. La función acepta como dato de entrada la edad [en términos enteros] de un niño en edad de educación básica [6-12 años] y el programa devuelve otro número entero que corresponde con el grado escolar, que es un valor entre 1 y 6.

Imagina que la función `grado` tiene el siguiente encabezado: `intgrado(int edad)`. Esto significa que el valor de entrada tendrá el nombre `edad` que es un elemento de `int` y debe devolver un valor del mismo tipo. El conjunto de referencia `int`, se refiere al conjunto de números enteros, que es mucho más grande de lo que se requiere, que son los números 6 al 12. Así que el dominio de la función es un subconjunto propio del conjunto de referencia, esto es

$$\mathtt{dom(grado)} \subset \mathtt{int}.$$

Por lo que `grado` es una función parcialmente definida.

El problema con las funciones parciales es que un valor en el conjunto de referencia, puede con todo derecho ser elegido para ser evaluado por la función, sin embargo, si tal valor no está relacionado entonces la evaluación será nula.

Definición 8.3.1 -- Nulo. El nulo es representado por `None`, y es un símbolo que significa simplemente que no hay valor asociado.

Ejemplo 8.12

Considera nuevamente la función f del ejemplo 8.9. Aquí $5 \in A$, por lo que en principio se podría intentar la evaluación, pero $\texttt{ev}(f,5) \mapsto \texttt{None}$.

En las funciones totales el dominio de la función coincide por completo con el conjunto de referencia, esto significa que cualquier valor del conjunto de referencia pertenece al dominio de la función, por lo que tendrá un valor asociado en el rango de la función.

Ejemplo 8.13

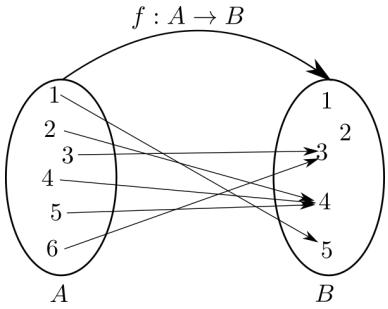
Considera ahora la función $\texttt{abs} : \mathbb{R} \to \mathbb{R}$, que asocia cada valor real con su valor absoluto, que también es un valor real, definida como:

$$\texttt{abs}(x) \mapsto \begin{cases} -x & \text{si } x < 0; \\ 0 & \text{si } x = 0: \\ x & \text{si } x > 0; \end{cases}$$

Como todos los elementos del conjunto de referencia $\mathbb{R}$ tienen algún valor asociado, la función es total.

Ejemplo 8.14

La función $f \leftarrow \{\langle 6,3\rangle, \langle 1,5\rangle, \langle 5,4\rangle, \langle 2,4\rangle, \langle 4,4\rangle, \langle 3,3\rangle\}$ es una función total.



En ocasiones es posible convertir una función parcial en total; esto se logra dedicando un valor en el rango de la función para que sea asociado con todos aquellos valores en el conjunto de referencia que inicialmente no pertenezcan al dominio.

Si suponemos que $v_{\texttt{None}} \in B$ es un valor en B que será dedicado para crear asociaciones con los valores que antes no habían sido asociados, entonces se modificará la firma de la función.

Sea $f : A \to B$ una función parcial, donde por supuesto $\texttt{dom}(f) \subset A$. supongamos que $v_{\texttt{None}} \in B \backslash \texttt{ran}(f)$ es el nuevo valor que será asociado con aquellos valores de A sin relación.

Si hacemos $f \leftarrow \{\langle a, v\rangle \in A \times B \mid a \in (A \backslash \texttt{dom}(f)\} \cup f$, entonces tendremos una función total.

8.3.2 Funciones inyectivas y sobreyectivas

Otra clasificación de las funciones tiene que ver con el rango de la función. Las funciones son etiquetadas como funciones inyectivas o funciones sobreyectivas [incluso ambas]. La manera en que el rango de la función es ocupado, es lo que determina la característica de la función.

Definición 8.3.2 -- Función inyectiva. Si $f : A \to B$ es una función, decimos que f es **inyectiva** o 1-1, si se cumple que

$$\forall b \in \texttt{ran}(f) : |\texttt{preim}(f, b)| = 1$$

La definición 8.3.2 establece que la cantidad de elementos del dominio que mapean al mismo elemento en el rango, debe ser exactamente uno.

Observa que si la preimagen de un elemento del rango es mayor que uno, entonces más de un elemento del dominio mapea al mismo elemento en el rango, lo que no es permitido en las funciones inyectivas.

Por otra parte, si la preimagen de un elemento en el conjunto de referencia B, significa simplemente que no pertenece al rango de la función. Observa la siguiente figura.

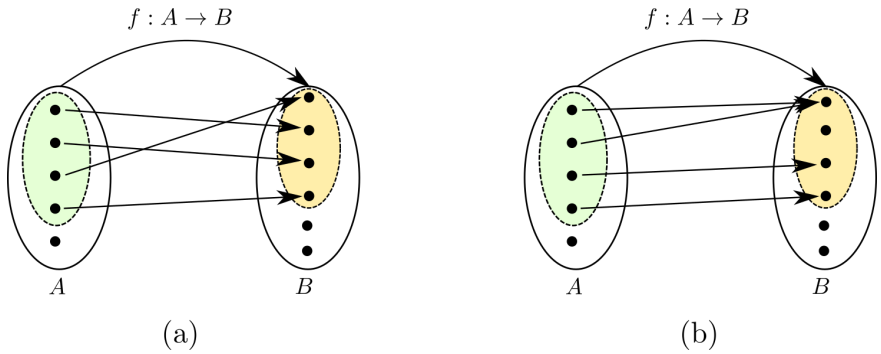
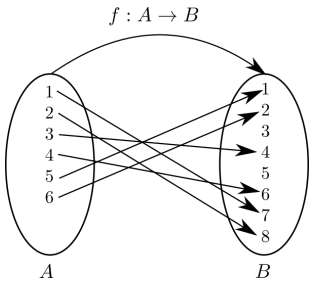


Figura 8.2: (a) *Función inyectiva.* (b) *Función no inyectiva, hay al menos un elemento en* $\texttt{ran}(f)$ *que no tiene preimagen unitaria.*

Ejemplo 8.15

Digamos que $f : A \to B$ es una función de $A \leftarrowtail \{1, 2, 3, 4, 5, 6\}$ a $B \leftarrowtail \{1, 2, 3, 4, 5, 6, 7, 8\}$, donde $f \leftarrowtail \{\langle 1, 7\rangle, \langle 2, 8\rangle, \langle 3, 4\rangle, \langle 4, 6\rangle, \langle 5, 1\rangle, \langle 6, 2\rangle\}$.



Aquí $\texttt{ran}(f) \leftarrowtail \{1, 2, 4, 6, 7, 8\}$ y cada uno de ellos tiene una preimagen unitaria, esto hace que la función sea inyectiva.

¿Es posible que para algún par $b_1, b_2 \in \texttt{ran}(f)$ diferentes, se cumpla que $|\texttt{preim}(b_1)| = 1$, $|\texttt{preim}(b_2)| = 1$ y $\texttt{preim}(b_1) = \texttt{preim}(b_2)$? Esto es que dos elementos diferentes del rango de la función, tengan preimágenes unitarias y éstas sean iguales. Esta situación es posible en las relaciones que no son funciones, por lo que en las funciones no es posible.

Una manera de verificar que una función es inyectiva, es asegurando que todos los elementos del rango tienen preimagen unitaria. Si un elemento del rango tuviera una preimagen que no es unitaria, entonces significa que hay más de un elemento en el dominio que está relacionado con el elemento del rango que está siendo probado.

Código 8.4: *Verifica funciones inyectivas*

```
def esIny(f:list, A:list=None, B:list=None)-> bool:
    """
    Verifica que una función sea inyectiva
    """
    if A is None: A = dom(f)
    if B is None: B = ran(f)
    return paraTodo(lambda b: esUnit(preim(f,b)), B)
```

Ejemplo 8.16

Utiliza la herramienta `esIny` para verificar que la función $f : \{1,\ldots,10\} \to \{1,\ldots,10\}$, definida como $f \leftarrow \{\langle 1,2\rangle, \langle 2,1\rangle, \langle 3,7\rangle, \langle 4,9\rangle, \langle 5,10\rangle, \langle 6,4\rangle, \langle 7,6\rangle, \langle 8,3\rangle, \langle 9,8\rangle, \langle 10,5\rangle\}$ es inyectiva.

```
>>> f = [[1, 2], [2, 1], [3, 7], [4, 9], [5, 10], [6, 4], [7, 6], [8, 3], [9, 8], [10, 5]]
>>> esFun(f)
True
>>> esIny(f)
True
>>>
```

Definición 8.3.3 -- Función sobreyectiva. Una función $f : A \to B$ es **sobreyectiva** o **sobre**, si se cumple que $B = \texttt{ran}(f)$.

Esta definición indica que todos los elementos en el conjunto de referencia B, deben ser utilizados en los pares de la función. Otro modo de ver esta definición, es que cada elemento de B, debe haber un elemento en el dominio de la función que lo mapee. La figura 8.3 muestra que en las funciones sobreyectivas, $\texttt{ran}(f) = B$ y en las funciones no sobreyectivas, $\texttt{ran}(f) \subset B$.

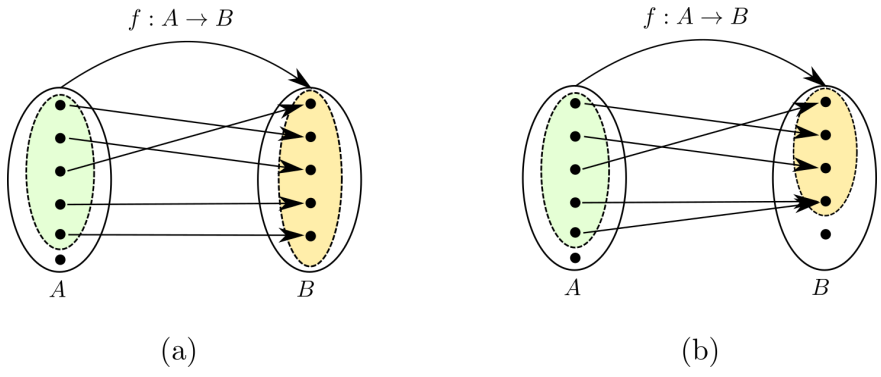
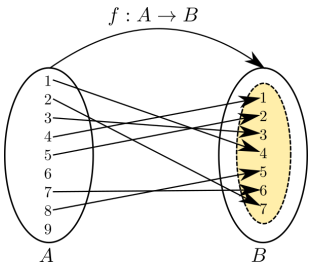


Figura 8.3: (a) *Función sobreyectiva.* (b) *Función no sobreyectiva.* $\texttt{ran}(f) \subset B$.

Ejemplo 8.17

Considera la función $f : A \to B$ con $A \leftarrow \{1,2,3,4,5,6,7,8,9\}$, $B \leftarrow \{1,2,3,4,5,6,7\}$ definida como $f \leftarrow \{\langle 4,1\rangle, \langle 5,2\rangle, \langle 3,3\rangle, \langle 1,4\rangle, \langle 8,5\rangle, \langle 7,6\rangle, \langle 2,7\rangle\}$.



En una función sobreyectiva, el rango de la función es el mismo que el conjunto de referencia B.

Para verificar que una función $f : A \to B$ es sobreyectiva hay varias opciones. Una de ellas es revisar cada elemento $b \in B$ y verificar que exista un elemento en $\texttt{dom}(f)$ que mapee a tal elemento b. Otra opción es verificar la igualdad entre el conjunto de referencia B y el rango de la función.

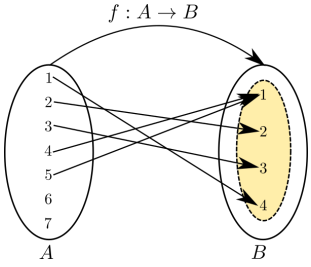
Código 8.5: *Verifica que una función sea sobreyectiva*

```
1 def esSobre(f:list, A:list=None, B:list=None)-> bool:
2     """
3     Verifica que una función sea sobreyectiva
4     """
5     if A is None: A = dom(f)
6     if B is None: B = ran(f)
7     return B == ran(f)
```

Ejemplo 8.18

Los siguientes ejemplos son funciones sobreyectivas:

1. La función identidad. $I : \mathbb{R} \to \mathbb{R}$ que toma un número real y devuelve ese mismo número.
2. La función módulo k. Dados dos números enteros $n \geq 0, k > 0$, $\texttt{Mod}(n,k)$ devuelve el residuo de la división entera $\frac{n}{k}$.
   a) $\texttt{Mod}(5,2) \mapsto 1$, esto es porque $\frac{5}{2}$ tiene cociente 2 y residuo 1.
   b) $\texttt{Mod}(14,8) \mapsto 5$, esto es porque $\frac{14}{8}$ tiene cociente 1 y residuo 6.
   c) $\texttt{Mod}(3,2) \mapsto 1$, esto es porque $\frac{3}{2}$ tiene cociente 1 y residuo 1.
3. La función $f : \{1,2,3,4,5,6,7\} \to \{1,2,3,4\}$ definida $\{\langle 1,4\rangle, \langle 2,2\rangle, \langle 3,3\rangle, \langle 4,1\rangle, \langle 5,1\rangle\}$



Observa en el ejemplo 3 de esta lista, que $\texttt{preim}(f,1) \mapsto \{4,5\}$, que evidentemente no es unitaria, pero esto no es impedimento alguno, pues la condición se restringe a verificar que el codominio B sea igual al $\texttt{ran}(f)$.

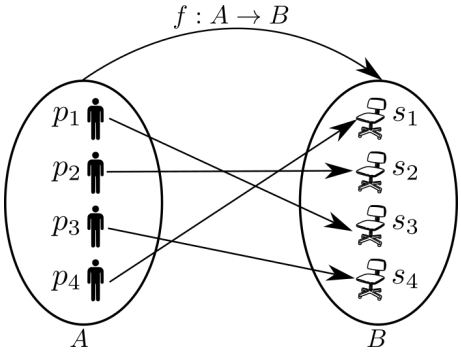
8.3.3 Funciones biyectivas

Definición 8.3.4 -- Función biyectiva. Una función $f : A \to B$ es una **biyección** o es una **función biyectiva** si f es tanto inyectiva como sobreyectiva.

La definición de función biyectiva, involucra las dos condiciones previas. Por un lado la función debe ser inyectiva, esto significa que para una función $f : A \to B$, cada elemento del conjunto de referencia A tiene exactamente un elemento de $\texttt{ran}(f)$ asociado; y por otro lado, todos los elementos del conjunto de referencia B se ocupan en la asociación.

Ejemplo 8.19

En un panel de discusión se han invitado a cuatro personas $A \leftarrow \{p_1, p_2, p_3, p_4\}$ y se han colocado cuatro asientos $B \leftarrow \{s_1, s_2, s_3, s_4\}$. La asociación de sillas a personas se dio de acuerdo a como llegaron al evento, el orden de llegada fue p_4, p_2, p_1 y al final la p_3, por lo que a cada persona se le asignó una silla de acuerdo a la siguiente regla: $f \leftarrow \{\langle p_4, s_1\rangle, \langle p_2, s_2\rangle, \langle p_1, s_3\rangle, \langle p_3, s_4\rangle\}$.



La relación $f : A \to B$ de la figura, es una función biyectiva.

```
def esBiy(f:list, A:list=None, B:list=None)-> bool:
    """
    Verifica que una función sea biyectiva
    """
    if A is None: A = dom(f)
    if B is None: B = ran(f)
    return y(esIny(f,A,B), esSobre(f,A,B))
```

Ejemplo 8.20

Verifica que la función $f : \{1,\ldots,6\} \to \{1,\ldots,6\}$ definida $f \leftarrow \{\langle 2,2\rangle, \langle 1,3\rangle, \langle 3,4\rangle, \langle 5,1\rangle, \langle 4,5\rangle, \langle 6,6\rangle\}$ es biyectiva.

```
>>> f = conj(tupla(2,2), tupla(1,3), tupla(3,4), tupla(5,1), tupla(4,5), tupla(6,6))
>>> esIny(f)
True
>>> esSobre(f)
True
>>> esBiy(f)
True
>>>
```

En primer lugar f es una función porque cada elemento del dominio está relacionado con exactamente un elemento del rango. En segundo lugar, la función es inyectiva, porque todos los elementos del conjunto de referencia $B \leftarrow \{1,\ldots,6\}$ tienen preimagen unitaria. En tercer lugar, la función es sobreyectiva, porque el rango es igual al conjunto B. Finalmente es biyectiva porque es inyectiva y sobreyectiva.

Una función biyectiva de un conjunto sobre sí mismo, es decir de la forma $f : A \to A$, también recibe el nombre de **función de permutación**. Podemos decir de una manera informal, que el orden de los elementos en un conjunto es una lista de los elementos que pertenecen al conjunto. Así los elementos del conjunto están ordenados de acuerdo a la posición que ocupan en esa lista. Una función permutación es un reordenamiento de los elementos del conjunto de referencia.

Ejemplo 8.21

En un juego de preguntas y respuestas por equipos, cada equipo consta de 5 integrantes que responden a las preguntas en cada turno del equipo. Para responder la pregunta que le corresponde al equipo, los cinco integrantes mantienen un orden y el que está al frente del equipo le corresponde contestar la pregunta. Una vez contestada la pregunta, el integrante pasa al final, de modo que la siguiente pregunta a contestar, será responsabilidad del siguiente miembro del equipo.

Esta dinámica se realiza un número k de veces. Al final se contabiliza el número re respuestas acertadas.

Supongamos que los integrantes de un equipo son $E \leftarrow \{e_1, e_2, e_3, e_4, e_5\}$, que han decidido un orden inicial $\langle e_1, e_5, e_2, e_3, e_4 \rangle$. Así, la primera pregunta la responderá e_1 y tendrá que pasar al final, la segunda le corresponde a e_5 y luego pasará al final. En ese momento el orden es $\langle e_2, e_3, e_4, e_1, e_5 \rangle$.

Cada vez que una persona contesta y pasa al final, es un reordenamiento.

En resumen, las funciones de la forma $f : A \to B$, es decir que relacionan elementos de un conjunto de referencia A a elementos en el otro conjunto de referencia B, se han clasificado como lo muestra el siguiente diagrama, aunque sin duda, puede haber más clasificaciones.

- Funciones
  - Dominio
    - Parciales: No todo el conjunto de referencia A se considera en el dominio
    - Totales: Todo el conjunto de referencia A se considera en el dominio
  - Rango
    - Inyectivas: Una asignación uno a uno. El rango no necesariamente es igual al codominio.
    - Sobreyectivas: El rango y el codominio son iguales.
    - Biyectivas: Son tanto inyectivas como sobreyectivas.

8.4 Operaciones con funciones

8.4.1 Función inversa

Definición 8.4.1 -- Función inversa. Sea $f : A \to B$ una biyección de A a B. La **función inversa** de f se denota $f^{-1} : B \to A$ y es tal que si $f(a) = b$, entonces $f^{-1}(b) = a$, con elementos $a \in A$ y $b \in B$.

La definición 8.4.1 asegura que si $\langle a, b \rangle \in f$, entonces $\langle b, a \rangle \in f^{-1}$. Esto es cierto porque f es biyectiva, pero si f no fuera biyectiva, pueden suceder dos cosas:

1. f^{-1} puede resultar se una función parcial, porque puede haber un elemento en el dominio de f^{-1} para el que no está definida la función.
2. f^{-1} resulta no ser una función, porque puede haber un elemento en el dominio de f^{-1} para el que existan más de un elemento en el rango que estén relacionados.

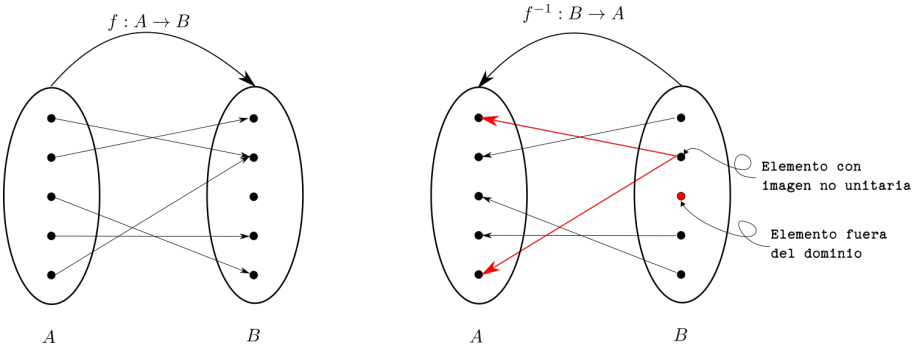


Figura 8.4: *La inversa de una función no siempre resulta ser una función.*

Cuando la inversa de una función resulta ser una función, decimos que la función es **invertible**, y esto sucede cuando la función original es biyectiva. Cuando la inversa de la función no es una función, decimos que la función es **no invertible**.

```
def esInv(f:list)-> bool:
    """
    Verifica que una función sea invertible.
    """
    return esBiy(f)

def esNoInv(f:list)-> bool:
    """
    Verifica que una función sea no invertible.
    """
    return neg(esBiy(f))
```

Ejemplo 8.22

Sea $f : \{a,b,c,d\} \to \{1,2,3,4\}$ una función definida por lo pares $\{\langle a,3\rangle, \langle b,1\rangle, \langle c,4\rangle, \langle d,2\rangle\}$. La función f es invertible porque es biyectiva.

```
>>> f = conj(tupla('a',3), tupla('b',1), tupla('c',4), tupla('d',2))
>>> esInv(f)
True
>>> esBiy(f)
True
>>> finv = rInv(f)
>>> print(finv)
[[3, 'a'], [1, 'b'], [4, 'c'], [2, 'd']]
>>> esInv(finv)
True
>>>
```

La inversa $f^{-1} \leftarrow \{\langle 3,a\rangle, \langle 1,b\rangle, \langle 4,c\rangle, \langle 2,d\rangle\}$ es también una función invertible.

Observa que si f es una función invertible, entonces la función inversa de la función inversa de f es f.

$$f = \texttt{rInv}(\texttt{rInv}(f))$$

8.4.2 Composición de funciones

Definición 8.4.2 -- Composición de funciones. Sean $f : A \to B$ y $g : B \to C$ dos funciones. La composición de f con g, es denotada por $(f \circ g)$, y se define por

$$(f \circ g) : A \to C$$
$$(f \circ g)(a) \mapsto g(f(a)).$$

Observa que en la composición de las funciones f con g, se crea una relación entre un elemento a en el dominio de f, con un elemento c en el rango de g, siempre y cuando exista un elemento b en el rango de f, que también sea elemento del dominio de g, que pueda ser evaluado en $f(a) \mapsto b$ y $g(b) \mapsto c$.

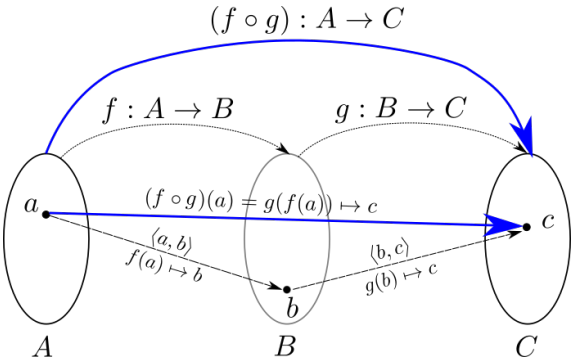


Figura 8.5: *Composición de funciones*

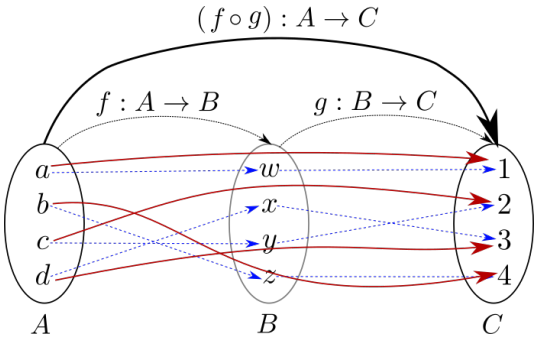
Si $(f \circ g)$ también se puede escribir simplemente fg.

Ejemplo 8.23

Digamos que la función f está dada por los pares $\{\langle a, \mathtt{w} \rangle, \langle b, \mathtt{z} \rangle, \langle c, \mathtt{y} \rangle, \langle d, \mathtt{x} \rangle\}$ y la función g está dada por los pares $\{\langle \mathtt{w}, 1 \rangle, \langle \mathtt{x}, 3 \rangle, \langle \mathtt{y}, 2 \rangle, \langle \mathtt{z}, 4 \rangle\}$.

Respecto de la función f, se tiene que $\mathtt{dom}(f) \mapsto \{a, b, c, d\}$ y $\mathtt{ran}(f) \mapsto \{w, x, y, z\}$; por su parte, la función g tiene $\mathtt{dom}(g) \mapsto \{w, x, y, z\}$ y $\mathtt{ran}(g) \mapsto \{1, 2, 3, 4\}$.

Como el $\mathtt{ran}(f) \subseteq \mathtt{dom}(g)$, es posible intentar la composición, de hecho $\mathtt{ran}(f) = \mathtt{dom}(g)$, así la composición $(f \circ g)$ es una nueva función con dominio en $\{a, b, c, d\}$ y rango en $\{1, 2, 3, 4\}$, con los siguientes pares:



Así $(f \circ g) \mapsto \{\langle a, 1 \rangle, \langle b, 4 \rangle, \langle c, 2 \rangle, \langle d, 3 \rangle\}$, porque:

$$\begin{array}{lll}(f\circ g)(a) & = g(f(a)) & = g(w)\mapsto 1\\ (f\circ g)(b) & = g(f(b)) & = g(z)\mapsto 4\\ (f\circ g)(c) & = g(f(c)) & = g(y)\mapsto 2\\ (f\circ g)(d) & = g(f(d)) & = g(x)\mapsto 3\end{array}$$

Consideremos ahora la función $I : C \to C$ la función identidad. Las siguientes igualdades:

Si $f : C \to B$, entonces $(I \circ f) = f$
Si $f : A \to C$, entonces $(f \circ I) = f$

8.5 Funciones de permutación

Debido a la importancia de este tipo de funciones [ver página 180] tienen utilidad en la creación de otras estructuras algebraicas, en el modelado matemático de juegos como el cubo de Rubik [Joy02, p. 221], entre muchas otras.

Definición 8.5.1 Una función de permutación, o simplemente permutación de A, es una función $\pi : A \to A$ que es biyectiva.

Ejemplo 8.24

Un cable de datos tipo USB se ha construido con 9 conductores eléctricos enumerados $A \leftarrow \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9\}$. Al conectar el cable al receptor, se hace un reordenamiento para que sea más fácil conectar los componentes. La reordenación es como se muestra en la siguiente tabla:

Fuente	Pista de uso	Descripción
c_1	c_3	GND
c_2	c_4	Vdd
c_3	c_5	Clock
c_4	c_2	DI
c_5	c_7	DO
c_6	c_1	SS
c_7	c_6	GND
c_8	c_9	RSVD
c_9	c_8	RSVD

La función $\pi \leftarrow \{\langle c_1, c_3\rangle, \langle c_2, c_4\rangle, \langle c_3, c_5\rangle, \langle c_4, c_2\rangle, \langle c_5, c_7\rangle, \langle c_6, c_1\rangle, \langle c_7, c_6\rangle, \langle c_8, c_9\rangle, \langle c_9, c_8\rangle\}$ es una biyección sobre un mismo conjunto, por lo que es una función de permutación.

8.5.1 Notación

Las permutaciones sobre un conjunto tienen una notación particular. Digamos que $A \leftarrow \{a_1, \ldots, a_n\}$ es un conjunto finito. Una permutación $\pi : A \to A$ se puede escribir en la siguiente forma

$$\pi \leftarrow \begin{pmatrix} a_1 & a_2 & \ldots & a_n \\ \pi(a_1) & \pi(a_2) & \ldots & \pi(a_n) \end{pmatrix}$$

La secuencia $\langle \pi(a_1), \ldots, \pi(a_n)\rangle$ es un reordenamiento de la secuencia $\langle a_1 \cdots a_n\rangle$ después de haber aplicado la permutación π, lo que podemos escribir como:

$$\langle a_1, \ldots, a_n\rangle \xrightarrow{\pi} \langle \pi(a_1), \ldots, \pi(a_n)\rangle$$

Como π es una biyección, la permutación inversa π^{-1} es:

$$\pi^{-1} \leftarrow\!\!\shortmid \begin{pmatrix} \pi(a_1) & \pi(a_2) & \dots & \pi(a_n) \\ a_1 & a_2 & \dots & a_n \end{pmatrix}$$

Por supuesto la composición de funciones también se puede aplicar en las permutaciones. De modo que $(\pi \circ \pi) = \pi^2$ es una nueva permutación:

$$\begin{pmatrix} a_1 & \dots & a_n \\ \pi(a_1) & \dots & \pi(a_n) \end{pmatrix} \circ \begin{pmatrix} a_1 & \dots & a_n \\ \pi(a_1) & \dots & \pi(a_n) \end{pmatrix} \mapsto \begin{pmatrix} a_1 & \dots & a_n \\ \pi(\pi(a_1)) & \dots & \pi(\pi(a_n)) \end{pmatrix}$$

Ejemplo 8.25

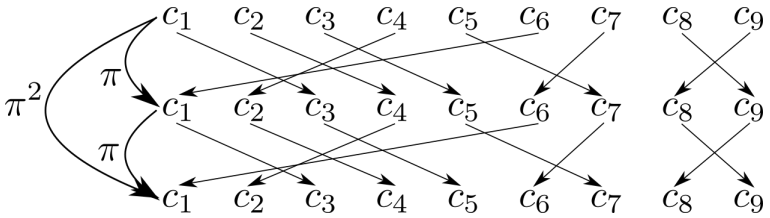
Digamos que $\pi \leftarrow\!\!\shortmid \{\langle c_1, c_3\rangle, \langle c_2, c_4\rangle, \langle c_3, c_5\rangle, \langle c_4, c_2\rangle, \langle c_5, c_7\rangle, \langle c_6, c_1\rangle, \langle c_7, c_6\rangle, \langle c_8, c_9\rangle, \langle c_9, c_8\rangle\}$, que es la permutación del ejemplo 8.24. La composición $\pi^2 = \pi \circ \pi$ genera la nueva permutación:

$$\pi^2 \leftarrow\!\!\shortmid \{\langle c_1, c_5\rangle, \langle c_2, c_2\rangle, \langle c_3, c_7\rangle, \langle c_4, c_4\rangle, \langle c_5, c_6\rangle, \langle c_6, c_3\rangle, \langle c_7, c_1\rangle, \langle c_8, c_8\rangle, \langle c_9, c_9\rangle\}$$

Porque:

$$\langle c1, c2, c3, c4, c5, c6, c7, c8, c9\rangle \xrightarrow{\pi} \langle c3, c4, c5, c2, c7, c1, c6, c9, c8\rangle \xrightarrow{\pi} \langle c5, c2, c7, c4, c6, c3, c1, c8, c9\rangle$$

Que también se puede obtener al seguir un camino de flechas de arriba hacia abajo en el siguiente diagrama:



8.5.2 Composición de permutaciones

La composición de dos permutaciones genera una nueva permutación. Para una colección de n elementos, hay $n!$ permutaciones diferentes. La permutación que deja el orden actual, se suele denotar como 1_A, donde A es el conjunto de referencia. Así $(\pi \circ I_A) = \pi$, también $(1_A \circ \pi) = \pi$.

Ejemplo 8.26

Sea $A \leftarrow\!\!\shortmid \{1, 2, 3\}$. Todas las $3! = 6$ permutaciones de A son:

$$1_A \leftarrow\!\!\shortmid \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} \quad \pi_2 \leftarrow\!\!\shortmid \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$$

$$\pi_3 \leftarrow\!\!\shortmid \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} \quad \pi_4 \leftarrow\!\!\shortmid \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$

$$\pi_5 \leftarrow\!\!\shortmid \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \quad \pi_6 \leftarrow\!\!\shortmid \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

La enumeración de las permutaciones es arbitraria. Observa que $\pi_2^2 = (\pi_2 \circ \pi_2) \mapsto 1_A$, también $\pi_2^2 = \pi_3^2 = \pi_6^2 = 1_A$, pero también $\pi_4^3 = \pi_5^3 = 1_A$.

8.5.3 Permutaciones cíclicas

Supongamos que $\pi : A \to A$ es una permutación sobre un conjunto $A \leftarrow\!\shortmid \{a_1, \ldots, a_n\}$, además supongamos que hay k elementos de A, digamos $a_1, a_2, \ldots, a_k$, que mapean como se muestra en la figura; el resto de los elementos de A, es decir, los elementos $a_{k+1}, \ldots, a_n$ mapean a sí mismos.

$$\pi \leftarrow\!\shortmid \left(\begin{array}{ccccc|cc} a_1 & a_2 & \ldots & a_{k-1} & a_k & a_{k+1} & a_n \\ \\ a_2 & a_3 & \ldots & a_k & a_1 & a_{k+1} & a_n \end{array} \right)$$

Observa que el primer elemento del renglón superior, mapea al segundo, el segundo al tercero; y así sucesivamente hasta el k-ésimo elemento, que mapea al primero. Un mapeo de esta forma es una **permutación cíclica** de longitud k, o simplemente ciclo de longitud k. Los demás elementos carecen de interés pues mapean a sí mismos. En particular, un ciclo de longitud 2 se llama **transposición**.

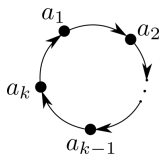


Figura 8.6: *Permutación cíclica* $a_1 \to a_2 \to \cdots \to a_k$

Una notación simplificada para una permutación cíclica consiste en escribir solamente los elementos involucrados en el ciclo y omitiendo los demás elementos. Por ejemplo, si en una permutación π sobre un conjunto $A \leftarrow\!\shortmid \{a_1, a_2, a_3, a_4, a_5\}$ y se tiene que $\pi(a_1) \mapsto a_2$, $\pi(a_2) \mapsto a_3$, $\pi(a_4) \mapsto a_4$ y $\pi(a_5) \mapsto a_5$, podemos escribir $\pi = \langle a_1, a_2, a_3 \rangle$, para denotar la permutación que toma a_1 y lo envía a a_2, este a a_3 y este lo regresa a a_1.

Observa que en esta notación se requiere saber el conjunto A, puesto que los elementos que no aparecen en la tupla, se supone que mapean a sí mismos.

Ejemplo 8.27

La permutación $\pi \leftarrow\!\shortmid \langle 4, 2, 3, 1 \rangle$ del conjunto $A \leftarrow\!\shortmid \{1, 2, 3, 4, 5, 6\}$ se refiere a la permutación:

$$\pi \leftarrow\!\shortmid \left(\begin{array}{cccccc} 4 & 2 & 3 & 1 & 5 & 6 \\ 2 & 3 & 1 & 4 & 5 & 6 \end{array} \right)$$

Observa que los elementos que no aparecen en la tupla del ciclo permutación producen pares reflexivos. En el caso de este ejemplo, ni el 5 ni el 6 aparecen en la tupla de la permutación cíclica $\langle 4, 2, 3, 1 \rangle$, por lo que se generan los pares $\langle 5, 5 \rangle$ y $\langle 6, 6 \rangle$ que también debe pertenecer a la función.

Código 8.6: *Notación de ciclo permutación*

```
1  def fPerm(p:list, A:list)-> list:
2      """
3      Genera una permutación de un conjunto A
4      dando un ciclo p.
5      """
6      B = tEscribe(cdr(p), car(p))
7      C = list(map(lambda a,b:tupla(a,b), p, B))
8      D = difc(A, p)
9      E = enCada(lambda d:tupla(d,d), D)
10     return tConcat(C, E)
```

Ejemplo 8.28

Utiliza la herramienta `fPerm` para generar la función permutación del ciclo $\langle 4,2,3,1\rangle$ en el conjunto $A \leftarrow \{1,2,3,4,5,6\}$.

```
>>> p = tupla(4,2,3,1)
>>> A = conj(1,2,3,4,5)
>>> fPerm(p, A)
[[4, 2], [2, 3], [3, 1], [1, 4], [5, 5], [6,6]]
>>>
```

8.5.4 Composición de permutaciones cíclicas

Debido a que una una permutación cíclica genera una función de permutación, es posible hacer la composición de dos funciones de permutación al dar las dos tuplas de ciclo de permutación y el conjunto de referencia.

Si $\pi_1 \leftarrow \langle a_1,\ldots,a_j\rangle$ y $\pi_2 \leftarrow \langle b_1,\ldots,b_k\rangle$ son dos permutaciones cíclicas sobre un mismo conjunto A. El producto $\pi_1\pi_2$ se refiere a la composición de la función de permutación generadas por π_1 con la función de permutación generada por π_2 [KB84, p. 182], esto es

$$\begin{pmatrix} a_1 & \ldots & a_j & a_{j+1} & \ldots & a_n \\ \pi_1(a_1) & \ldots & \pi_1(a_j) & a_{j+1} & \ldots & a_n \end{pmatrix} \circ \begin{pmatrix} b_1 & \ldots & b_k & b_{k+1} & \ldots & b_n \\ \pi_2(b_1) & \ldots & \pi_2(b_k) & b_{k+1} & \ldots & b_n \end{pmatrix}$$

Ejemplo 8.29

Sea $A \leftarrow \{1,2,3,4,5,6,7,8\}$ sobre el que se han definido las permutaciones cíclicas $c_1 \leftarrow \langle 2,4,5,7,1\rangle$ y $c_2 \leftarrow \langle 3,5,4,8,7\rangle$. El producto c_1c_2 es como sigue:

$$c_1 \leftarrow \begin{pmatrix} 2 & 4 & 5 & 7 & 1 & 3 & 6 & 8 \\ 4 & 5 & 7 & 1 & 2 & 3 & 6 & 8 \end{pmatrix}, \quad c_2 \leftarrow \begin{pmatrix} 3 & 5 & 4 & 8 & 7 & 1 & 2 & 6 \\ 5 & 4 & 8 & 7 & 3 & 1 & 2 & 6 \end{pmatrix}$$

Al hacer la composición

$$\begin{matrix} & & & & & c_1 & & & & & & & & c_2 & & \\ \Big(& 2 & 4 & 5 & 7 & 1 & 3 & 6 & 8 & \Big) & \Big(& 3 & 5 & 4 & 8 & 7 & 1 & 2 & 6 & \Big) \\ & \downarrow & & & & & & & & & & & & \downarrow & & \\ & 4 & 5 & 7 & 1 & 2 & 3 & 6 & 8 & & & 5 & 4 & 8 & 7 & 3 & 1 & 2 & 6 & \end{matrix}$$

Después de calcular todos los pares: $(c_1 \circ c_2) \leftarrow \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 8 & 5 & 4 & 3 & 6 & 1 & 7 \end{pmatrix}$.

Código 8.7: *Producto de permutaciones cíclicas*

```
1  def pProd(p1:list, p2:list, A:list)-> list:
2      """
3      Producto de permutaciones cíclicas sobre
4      un conjunto A
5      """
6      return rComp(fPerm(p1,A), fPerm(p2,A))
```

```
>>> c1 = tupla(2,4,5,7,1)
>>> c2 = tupla(3,5,4,8,7)
>>> print(pProd(c1,c2,[1,2,3,4,5,6,7,8]))
[[2, 8], [4, 4], [5, 3], [7, 1], [1, 2], [3, 5], [6, 6], [8, 7]]
>>>
```

8.5.5 Permutaciones cíclicas disjuntas

Consideremos ahora el resultado del ejemplo 8.29, donde se obtuvo la siguiente permutación:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 8 & 5 & 4 & 3 & 6 & 1 & 7 \end{pmatrix}$$

Como las columnas representan pares en la función, es posible reacomodar los términos con el fin de intentar construir una permutación cíclica:

$$\begin{pmatrix} 1 & 2 & 8 & 7 & 3 & 5 & 4 & 6 \\ 2 & 8 & 7 & 1 & 5 & 3 & 4 & 6 \end{pmatrix}$$

Observamos un patrón similar a una permutación cíclica, pero no es tal. Aunque parecen dos ciclos, uno de longitud 3 que es $1 \to 2 \to 8 \to 7$ y la transposición $3 \to 5$ como se muestra:

$$\begin{pmatrix} 1 & 2 & 8 & 7 & 3 & 5 & 4 & 6 \\ 2 & 8 & 7 & 1 & 5 & 3 & 4 & 6 \end{pmatrix}$$

Definición 8.5.2 Decimos que dos permutaciones cíclicas π_1 y π_2 de un mismo conjunto A son **disjuntas**, si los elementos que aparecen en π_1 no aparecen en π_2 y ningún elemento de π_2 aparece en π_1.

Ejemplo 8.30

Las permutaciones cíclicas $\langle 1,2,8,7 \rangle$ y $\langle 3,5 \rangle$ sobre el conjunto $A \leftarrowtail \{1,2,3,4,5,6,7,8\}$ son disjuntas, porque ningún elemento de A aparece en ambas permutaciones cíclicas.

Ahora, si π_1 y π_2 son permutaciones disjuntas sobre el mismo conjunto, se tiene que $(\pi_1 \circ \pi_2) = (\pi_2 \circ \pi_1)$. Esto es así porque los elementos de una permutación no intervienen en los elementos de la otra.

De modo que cualquier composición de permutaciones se puede representar como composición de permutaciones cíclicas.

Ejemplo 8.31

La permutación $\begin{pmatrix} 1 & 2 & 8 & 7 & 3 & 5 & 4 & 6 \\ 2 & 8 & 7 & 1 & 5 & 3 & 4 & 6 \end{pmatrix}$ se puede representar por el producto de las permutaciones cíclicas disjuntas $\langle 1,2,8,7 \rangle \langle 3,5 \rangle$ sobre el conjunto $\{1,2,3,4,5,6,7,8\}$.

Ahora consideremos cualquier permutación cíclica sobre un conjunto A con $n \leftarrowtail |A|$. La permutación cíclica $\langle a_1, \ldots, a_\ell \rangle$ con $2 \leq \ell \leq n$, puede ser expresada como la composición de las transposiciones

$$\langle a_1, a_2 \rangle \circ \langle a_1, a_3 \rangle \circ \cdots \circ \langle a_1, a_\ell \rangle$$

Para asegurarnos de que esto es cierto seguiremos el método inductivo. Verificamos en primer lugar que se cumpla para $\ell = 2$. Con este valor, la permutación cíclica es $\langle a_1, a_2 \rangle$, que ya está en la forma de una transposición, así que $\langle a_1, a_2 \rangle = \langle a_1, a_2 \rangle$.

Ahora, supongamos que es cierto que

$$\langle a_1, a_2 \rangle \circ \langle a_1, a_3 \rangle \circ \cdots \circ \langle a_1, a_\ell \rangle \mapsto \langle a_1, \ldots, a_\ell \rangle, \tag{8.1}$$

hay que asegurar que

$$\langle a_1, a_2 \rangle \circ \langle a_1, a_3 \rangle \circ \cdots \circ \langle a_1, a_\ell \rangle \circ \langle a_1, a_{\ell+1} \rangle \mapsto \langle a_1, \ldots, a_{\ell+1} \rangle. \tag{8.2}$$

A partir de 8.1 podemos escribir

$$\langle a_1, a_2 \rangle \circ \langle a_1, a_3 \rangle \circ \cdots \circ \langle a_1, a_\ell \rangle \circ \langle a_1, a_{\ell+1} \rangle = \langle a_1, \ldots, a_\ell \rangle \circ \langle a_1, a_{\ell+1} \rangle \tag{8.3}$$

Esto significa en realidad que

$$\left(\begin{array}{cccc|ccc} a_1 & a_2 & \ldots & a_\ell & a_{\ell+1} & \ldots & a_n \\ a_2 & a_3 & \ldots & a_1 & a_{\ell+1} & \ldots & a_n \end{array} \right) \circ \left(\begin{array}{cc|ccccc} a_1 & a_{\ell+1} & a_2 & \ldots & a_\ell & \ldots & a_n \\ a_{\ell+1} & a_1 & a_2 & \ldots & a_\ell & \ldots & a_n \end{array} \right).$$

La línea vertical divide a los elementos del conjunto de referencia en dos partes, a la izquierda los que intervienen en el ciclo y a la derecha los elementos reflexivos. Al hacer la composición se tiene

$$\left(\begin{array}{ccccc|cc} a_1 & a_2 & \ldots & a_\ell & a_{\ell+1} & \ldots & a_n \\ a_2 & a_3 & \ldots & a_{\ell+1} & a_1 & \ldots & a_n \end{array} \right) \mapsto \langle a_1, \ldots, a_{\ell+1} \rangle. \tag{8.4}$$

Entonces, podemos expresar cualquier permutación como una composición de transposiciones, primero expresándola como productos de permutaciones disjuntas y luego como producto de transposiciones.

Ejemplo 8.32

Expresar la permutación $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 4 & 5 & 6 & 1 & 2 & 7 \end{pmatrix}$ como producto de transposiciones, considerando $\{1, 2, 3, 4, 5, 6, 7\}$ en conjunto de referencia.

Primero reordenamos los elementos para identificar los ciclos.

$$\begin{pmatrix} 1 & 3 & 5 & 2 & 4 & 6 & 7 \\ 3 & 5 & 1 & 4 & 6 & 2 & 7 \end{pmatrix}$$

Ahora expresamos la permutación como producto de permutaciones cíclicas.

$$\langle 1, 3, 5 \rangle \langle 2, 4, 6 \rangle$$

Finalmente como producto de transposiciones

$$\langle 1, 3 \rangle \langle 1, 5 \rangle \langle 2, 4 \rangle \langle 2, 6 \rangle.$$

Las permutaciones pueden ser pares o impares. Si una permutación se puede expresar como un producto de un número par de transposiciones, entonces es una **permutación par**. En cambio, si la permutación se puede expresar con un número impar de transposiciones, es una **permutación impar**.

Ejercicios

1. Determina si los siguientes conjuntos de pares son funciones de A en A, cuando $A \leftarrow \{n \in \mathbb{Z} | 1 \leq n \leq 10\}$.
   a) $\{\langle 2,6\rangle, \langle 4,4\rangle, \langle 6,7\rangle, \langle 2,5\rangle, \langle 5,8\rangle, \langle 2,7\rangle, \langle 9,7\rangle, \langle 7,4\rangle, \langle 4,6\rangle, \langle 5,7\rangle\}$
   b) $\{\langle 1,5\rangle, \langle 7,2\rangle, \langle 3,10\rangle, \langle 8,8\rangle, \langle 10,2\rangle, \langle 6,3\rangle, \langle 2,2\rangle, \langle 5,6\rangle, \langle 4,9\rangle, \langle 9,7\rangle\}$
   c) $\{\langle 1,9\rangle, \langle 7,7\rangle, \langle 4,2\rangle, \langle 6,9\rangle, \langle 6,2\rangle, \langle 3,3\rangle, \langle 6,4\rangle, \langle 6,8\rangle, \langle 6,10\rangle, \langle 2,6\rangle\}$
   d) $\{\langle 4,2\rangle, \langle 10,3\rangle, \langle 3,8\rangle, \langle 5,6\rangle, \langle 8,3\rangle, \langle 9,1\rangle, \langle 2,5\rangle, \langle 1,1\rangle, \langle 6,6\rangle, \langle 7,5\rangle\}$
2. Considera el conjunto $A \leftarrow \{a,b,c,d,e,x,g,h\}$ y la función $f : A \to A$ definida por el conjunto $\{\langle h,b\rangle, \langle c,h\rangle, \langle b,c\rangle, \langle d,d\rangle, \langle a,h\rangle, \langle g,h\rangle, \langle e,c\rangle, \langle x,d\rangle\}$. Calcula la evaluación y preimagen de los siguientes valores.
   a) $a \in A$
   b) $x \in A$
   c) $d \in A$
   d) $b \in A$
3. Considera los conjuntos $A \leftarrow \{1,2,3,4,5,6,7,8\}$ y $B \leftarrow \{1,2,3,4,5\}$. De las siguientes funciones de A en B, determina si es una función total o es una función parcial.
   a) $\{\langle 8,7\rangle, \langle 3,1\rangle, \langle 5,4\rangle, \langle 6,2\rangle, \langle 4,7\rangle, \langle 7,7\rangle, \langle 1,1\rangle, \langle 2,2\rangle\}$.
   b) $\{\langle 1,6\rangle, \langle 7,8\rangle, \langle 8,7\rangle, \langle 4,4\rangle, \langle 5,6\rangle\}$.
   c) $\{\langle 7,3\rangle, \langle 6,2\rangle, \langle 1,8\rangle, \langle 3,6\rangle, \langle 5,7\rangle, \langle 2,6\rangle, \langle 4,4\rangle, \langle 8,6\rangle\}$.
   d) $\{\langle 6,8\rangle, \langle 2,8\rangle, \langle 5,5\rangle, \langle 8,3\rangle, \langle 7,2\rangle, \langle 3,5\rangle, \langle 1,8\rangle\}$.
4. Considera los conjuntos $A \leftarrow \{1,2,3,4,5\}$ y $B \leftarrow \{1,2,3,4,5\}$. De las siguientes funciones de A en B, determina si es una función inyectiva, sobreyectiva, es ambas o ninguna.
   a) $\{\langle 1,4\rangle, \langle 2,3\rangle, \langle 3,1\rangle, \langle 4,4\rangle, \langle 5,2\rangle\}$.
   b) $\{\langle 1,3\rangle, \langle 2,2\rangle, \langle 3,1\rangle, \langle 4,5\rangle, \langle 5,4\rangle\}$.
   c) $\{\langle 4,2\rangle, \langle 3,1\rangle, \langle 1,4\rangle, \langle 5,4\rangle, \langle 2,3\rangle\}$.
   d) $\{\langle 1,5\rangle, \langle 2,2\rangle, \langle 3,1\rangle, \langle 4,4\rangle, \langle 5,3\rangle\}$.
5. Considera los conjuntos $A \leftarrow \{1,2,3,4,5\}$ y $B \leftarrow \{1,2,3,4,5\}$. De las siguientes funciones de A en B, determina si es una función biyectiva.
   a) $\{\langle 1,5\rangle, \langle 2,4\rangle, \langle 3,2\rangle, \langle 4,3\rangle, \langle 5,1\rangle\}$.
   b) $\{\langle 3,5\rangle, \langle 4,5\rangle, \langle 1,5\rangle, \langle 5,1\rangle, \langle 2,4\rangle\}$.
   c) $\{\langle 4,2\rangle, \langle 2,5\rangle, \langle 1,5\rangle, \langle 5,2\rangle, \langle 3,4\rangle\}$.
   d) $\{\langle 3,3\rangle, \langle 1,4\rangle, \langle 5,5\rangle, \langle 2,2\rangle, \langle 4,1\rangle\}$.
6. Considera las siguientes funciones. $f : A \to A$ y $g : A \to A$, donde el conjunto de referencia es $A \leftarrow \{1,2,3,4,5,6,7,8,9,10\}$. $f \leftarrow \{\{\langle 8,3\rangle, \langle 10,5\rangle, \langle 3,2\rangle, \langle 6,9\rangle, \langle 9,6\rangle, \langle 5,8\rangle, \langle 7,7\rangle, \langle 2,4\rangle, \langle 1,1\rangle, \langle 4,10\rangle\}\}$ y $g \leftarrow \{\langle 7,2\rangle, \langle 8,4\rangle, \langle 2,10\rangle, \langle 6,9\rangle, \langle 1,8\rangle, \langle 10,1\rangle, \langle 4,6\rangle, \langle 3,7\rangle, \langle 9,5\rangle, \langle 5,3\rangle\}$. Calcula
   a) $(f \circ g)$
   b) $(g \circ f)$
   c) $(f \circ g)^{-1}$
   d) $(g \circ f)^{-1}$
7. Considera las permutaciones

$$\pi_1 = \leftarrow \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 7 & 7 & 6 & 3 & 4 & 5 \end{pmatrix} \quad \pi_2 = \leftarrow \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 1 & 5 & 2 & 3 & 7 & 6 \end{pmatrix}$$

   Calcula:
   a) π_1^3
   b) $(\pi_1 \circ {\pi_2}^2)$
   c) $({\pi_1}^2 \circ \pi_2)$

8. Con las permutaciones π_1 y π_2 sobre $\{1,2,3,4,5,6,7\}$ del ejercicio anterior, ¿cuántas veces hay que componer π_1 consigo misma para obtener 1_A? y para lograr 1_A, ¿cuántas veces se debe componer π_2 consigo misma?
9. Escribe π_1 y π_2 como producto de permutaciones cíclicas disjuntas.
10. Escribe π_1 y π_2 como producto de transposiciones.
11. Escribe una función en `Python` que calcule la preimagen de un conjunto de elementos en el rango de una relación. La función se debe llamar `Preim` y debe recibir dos argumentos: una relación R dada como una lista de pares; y un subconjunto Y de elementos en el rango de la relación. El resultado debe ser un conjunto que es calculado como la unión de las preimágenes de cada elemento en el subconjunto Y:

$$\texttt{Preim}(R,Y) \mapsto \bigcup_{y\in Y} \texttt{preim}(R,y).$$

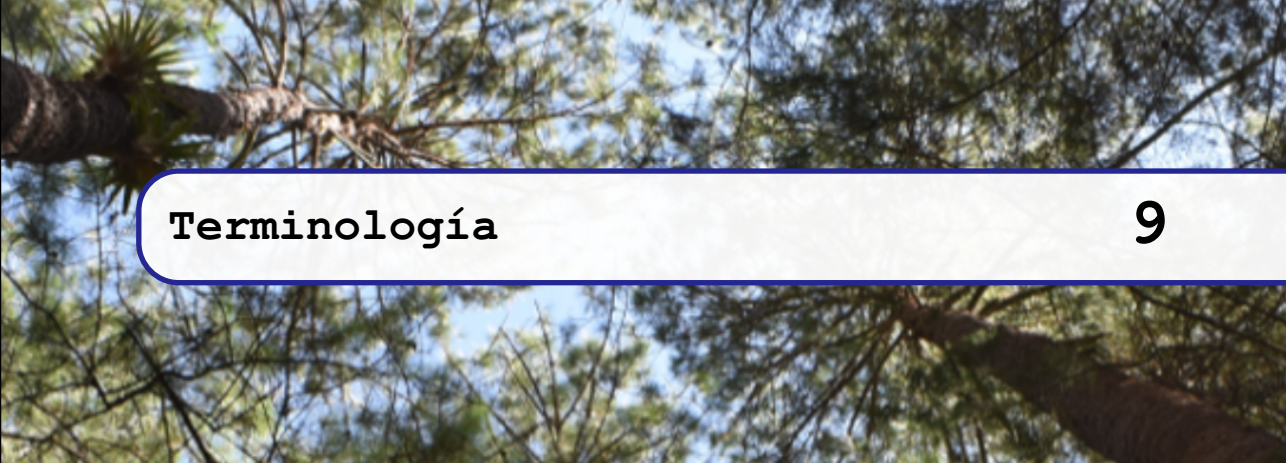
```
>>> R = conj(tupla(1, 4), tupla(2, 7), tupla(3, 6), tupla(7, 4), tupla(6, 3), tupla
    (2, 5), tupla(3, 4), tupla(5, 7), tupla(3, 7), tupla(7, 6))
>>> Preim(R,[1,3,7])
[2,5,3,6]
>>> Preim(R,[])
[]
>>>
```

12. Escribe una función en `Python` que se llame `esPpar` que tiene el propósito determinar si una permutación es par. El programa debe recibir una permutación en forma de lista de pares; como salida, el programa debe devolver `True` si la permutación es una permutación par y `False` si no lo es.

```
>>> pi1 = [[7, 5], [1, 1], [6, 4], [4, 6], [3, 7], [2, 2], [5, 3]]
>>> pPpar(pi1)
False
>>> pi2 = [[4, 2], [1, 4], [6, 7], [2, 1], [5, 3], [7, 6], [3, 5]]
>>> pPpar(pi2)
True
>>>
```

IV

Grafos dirigidos



Terminología 9

9.1 Conceptos generales

Los grafos dirigidos, también conocidos como «digrafos», son representaciones gráficas de las relaciones. Tienen una notable importancia en muchas áreas de conocimiento, en particular en Ciencias Computacionales, que son base de muchos modelos. En general, aquellas situaciones en donde intervenga un conjunto de elementos y una relación sobre ese conjunto es susceptible de ser representada con un grafo dirigido. Aunque hay grafos dirigidos y no dirigidos, como más adelante se detalla [página 199], en este libro solamente trataremos con grafos dirigidos, por lo que cuando se hace mención de un grafo, se refiere al caso dirigido, salvo en aquellas ocasiones donde se hace mención explícita del caso.

Los grafos se utilizan para representar objetos y la relación entre estos. Entre las disciplinas que utilizan frecuentemente los grafos son matemáticas, física, química, ciencias sociales y ecología entre muchas otras.

9.1.1 Definición estructural

Definición 9.1.1 Sea V un conjunto no vacío de elementos llamados «vértices» y $A \subseteq V \times V$ una relación entre vértices, cuyos elementos son conocidos como las «aristas». Un grafo dirigido es una estructura $G \leftarrowtail [\![V; A]\!]$ que encapsula los vértices y las aristas.

Ejemplo 9.1

En ecología se puede crear un grafo $[\![V; A]\!]$ donde V es un conjunto de seres vivos y A es el conjunto de pares $\langle u, v \rangle \in V^2$, que indica que u es predador de v. Así un grafo que involucre estos conjuntos, puede servir para modelar una cadena alimenticia.

Si $G \leftarrow [\![V; A]\!]$ es un grafo, $G.V$ se refiere al conjunto de vértices del grafo G y $G.A$ es el conjunto de aristas del mismo grafo. Si no hay confusión alguna sobre el grafo en uso, podemos omitir el nombre del grafo y denotar simplemente V como los vértices y A las aristas del grafo G.

Como la intención primordial al escribir este libro, ha sido proporcionar herramientas computacionales para las matemáticas discretas, en las siguientes secciones se detallan funciones en lenguaje de programación, para modelar los vértices, las aristas y los grafos; también serán útiles algunas herramientas para verificar, comparar y manipular tales objetos.

Se ha utilizado un enfoque orientado a objetos para la definición de los elementos que constituyen un grafo, la razón es en parte para introducir este paradigma de programación y en parte por la facilidad y versatilidad que proporciona este modelo para la implementación y uso de los grafos.

La definición de una clase se hace escribiendo la declaración de atributos y métodos con la siguiente estructura:

```
class ⟨Nombre_de_la_clase⟩:
    def __init__(self, ⟨param⟩...):
        ⟨expr⟩
    def id_metodo(self, ⟨param⟩...)
        ⟨expr⟩
    ...
```

El ⟨*Nombre_de_la_clase*⟩ es el identificador de la clase. `__init__` [conn dos líneas de subrayar en cada lado] establece la definición de los parámetros que dan valor inicial a los atributos de la clase. `id_metodo` representa el nombre que identifica un método, se pueden colocar de ese modo todos los métodos necesarios. Dentro de la lista de parámetros, se incluye `self`, para indicar que se deben incluir los atributos de la clase [CA16, p. 187][Pyt23, cap. 9. Classes].

9.1.2 Vértices

Definición 9.1.2 Un vértice es un elemento de un conjunto V sobre el que se define una relación binaria de la forma $G : V \to V$.

Los vértices que participan en la definición de la relación encapsulada en G, pueden ser de cualquier naturaleza, por ejemplo personas, números, animales, ciudades, etc. Gráficamente se representan como pequeños círculos etiquetados.

Código 9.1: *Creación de vértices*

```
1  class Vertice:
2      """
3      Unidad fundamental e información en Grafos.
4      Crea un vértice con la información dada.
5      """
6      def __init__(self, inf = None):
7          self.inf = inf
8          # otros atributos
9
10     def __str__(self):
11         return f"[{self.inf}]"
```

El método especial `__str__` [con dos líneas de subrayar en cada lado] ofrece la representación textual del objeto. Cuando se imprime el objeto, por ejemplo `print(Vertice)`, el texto impreso es lo que devuelve este método. En general este método se utiliza para beneficio de los programadores y las personas que utilizan esta clase [Lot19, p. 7][Lot19, p. 118].

Gracias a esta clase, es posible agregar atributos cuando sea necesario, también será posible agregar nuevos métodos. Por ejemplo puede ser necesario agregar un atributo `color`, que es útil en algunos tipos particulares de grafos.

Creación de vértices

Para crear un vértice haremos: ⟨*var*⟩ = `Vertice`(⟨*valor*⟩), donde `var` es el identificador de una variable y `valor` es de cualquier tipo. En este libro utilizaremos números para el valor de los vértices. Por convención no habrá información repetida en el conjunto de vértices.

Ejemplo 9.2

Cuando se tiene una expresión como: Sea $V \leftarrow \{1, 2, 3, 4, 5\}$ el conjunto de vértices. Para modelar en `Python` estos vértices creamos cada vértice y luego los reunimos en un conjunto:

```
>>> v1 = Vertice(1)
>>> v2 = Vertice(2)
>>> v3 = Vertice(3)
>>> v4 = Vertice(4)
>>> v5 = Vertice(5)
>>> V = conj(v1, v2, v3, v4, v5)
>>>
```

Observa que `V` es ahora un conjunto de estructuras de tipo `Vertice`:

```
>>> V
[<__main__.Vertice object at 0x7f8a99200ac0>, <__main__.Vertice object at 0x7f8a7c14dee0>,
<__main__.Vertice object at 0x7f8a7c14df10>, <__main__.Vertice object at 0x7f8a7c14dcd0>,
<__main__.Vertice object at 0x7f8a99829eb0>]
>>>
```

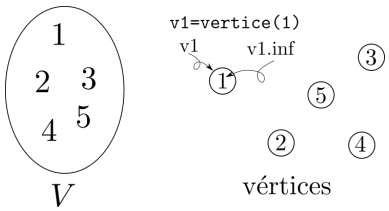


Figura 9.1: *Los vértices son elementos de un conjunto que encapsulan información.*

La información de un vértice puede ser de cualquier tipo, por ejemplo un vértice puede tener como información un número entero, que puede ser un índice en una tabla, con la información de una persona, ciudad o algún otro concepto.

Si v es un vértice, es decir, un objeto de la clase vértice, $v.inf$ se refiere a la información del vértice v, que es un atributo de la clase. Por comodidad, al referirse a un vértice v, significa en realidad $v.inf$.

```
>>> v1.inf
1
>>> v2.inf
2
>>> v3.inf
3
>>>
```

Una extensión natural sería obtener la información de una lista de vértices.

```
def VInf(LV:list):
    """
    Obtiene la información de una lista de vértices [objetos].
    """
    return enCada(lambda v: v.inf, LV)
```

Ejemplo 9.3

Del ejemplo 9.2, donde se tiene definido el conjunto $V \leftarrow \{1,2,3,4,5\}$, ya se tienen las estructuras `v1`, `v2`, `v3`, `v4` y `v5` que son de la clase `Vertice`. Podemos obtener la información de todos los vértices en `V` con la función `VInf`:

```
>>> VInf(V)
[1, 2, 3, 4, 5]
>>>
```

Definición 9.1.3 -- Vértices iguales. Si u, v son vértices, decimos que u es igual a v y lo denotamos $u = v$, si la información contenida en u es exactamente la misma que la información de v.

```
def vIguales(u:Vertice, v:Vertice)-> bool:
    """
    Determina si el vértice u es igual al vértice v.
    """
    return u.inf == v.inf
```

Al trabajar con vértices, usualmente se trabaja con la información contenida en los vértices y pocas veces con la propia estructura de datos. Cuando se tiene un conjunto de vértices [de estructuras de tipo `Vertice`] es necesario obtener el objeto cuya información sea la que se está buscando. De no encontrar el vértice buscado, el resultado de la función puede ser el valor `False`.

```
def vEnV(u, V:list)-> Vertice or bool:
    """
    Busca el vértice u en el conjunto de vértices V,
    dando la información del vértice
    y un conjunto de vértices.
    """
    U = subc(lambda v: u == v.inf, V)
    if esVacio(U):
        return False
    else:
        return car(U)
```

Ejemplo 9.4

Del ejemplo 9.2 se tiene el conjunto `V`. Se debe obtener el vértice con información igual a `4`

```
>>> vEnV(4, V)
<__main__.Vertice object at 0x7f07a5c15dc0>
>>> xx = vEnV(4, V)
>>> vInf(xx)
4
>>> vEnV(8, V)
False
>>>
```

En esta interacción, `xx` es una variable cuyo valor queda determinado al buscar en el conjunto V el vértice cuya información es 4; como el vértice sí existe, `xx` es ahora un objeto de la clase `Vertice`, y está alojado en una dirección de memoria.

9.1.3 Aristas

Definición 9.1.4 -- Arista. Una arista es un predicado que relaciona vértices. Si u y v son vértices, una arista de u a v es la tupla $\langle u, v \rangle$, o simplemente uv.

En un grafo, todas las aristas representan instancias del mismo predicado. Así un predicado $P(u, v)$, se utilizará para todos los vértices en el mismo grafo. Las aristas son representadas gráficamente por flechas. Si el vértice u está relacionado con el vértice v, entonces habrá una flecha que sale de u y llega a v. Para abreviar flechas, es posible dibujar aristas con doble flecha si en la relación se encuentran tanto $\langle u, v \rangle$ como $\langle v, u \rangle$.

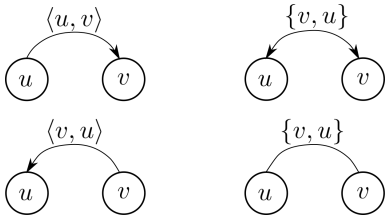


Figura 9.2: *Aristas dirigidas y no dirigidas*

Creación de aristas

El concepto de «arista» depende del concepto «vértice». Usaremos una clase llamada `Arista` con dos atributos `vi` y `vf` para nombrar el vértice inicial `vi` y el vértice final `vf`, que deben ser objetos de la clase `Vertice`, por comodidad permitiremos que los vértices sean dados por su información, que debe ser única en el conjunto de vértices.

Cuando se crea un objeto de la clase `Arista` dando la información de los vértices, primero se deben crear los objetos de tipo `Vertice` y luego crear la arista.

```
class Arista:
    """
    Se define una arista de la forma <vi, vf> donde
    vi es el vértice inicial y vf es el vértice final.
    """
    def __init__(self, vi = None, vf = None):
        self.vi = vi if isinstance(vi,Vertice) else Vertice(vi)
        self.vf = vf if isinstance(vf,Vertice) else Vertice(vf)
        self.inf = [self.vi.inf, self.vf.inf]
```

Ejemplo 9.5

Sea $A \leftarrow \{\langle 1,2\rangle, \langle 2,1\rangle, \langle 2,3\rangle, \langle 3,5\rangle, \langle 5,2\rangle, \langle 5,3\rangle\}$ un conjunto de aristas.

```
>>> a1 = Arista(1,2)
>>> a2 = Arista(2,1)
>>> a3 = Arista(2,3)
>>> a4 = Arista(3,5)
>>> a5 = Arista(5,2)
>>> a6 = Arista(5,3)
>>> A = conj(a1,a2,a3,a4,a5,a6)
>>>
```

Cuando se crea una arista mediante `Arista`, el contenido de la arista se encapsula. Para acceder a los atributos y métodos de la clase, es necesario hacerlo con la notación adecuada.

```
>>> a1
<__main__.Arista object at 0x7fcbbe2ccac0>
>>> type(a1)
<class '__main__.Arista'>
>>>
```

La información de una arista

La información de la arista se compone de la información de los vértices que la constituyen, necesitaremos un par de procedimientos para obtener los vértices [los objetos] definidos en las aristas.

Definición 9.1.5 Si $a \in A$ es una arista, donde $a = \langle u, v \rangle$, el **vértice inicial** de a se denota $a.vi$ y es el vértice u; el **vértice final** de a se denota $a.vf$ y es v.

Además de los vértices inicial y final de la arista, es posible adjuntar otro tipo de información, como la tupla conformada por la información de los vértices.

Si a es una arista, $a.inf$ es la información de la arista. Como $a.vi$ es un vértice, $a.vi.inf$ es la información del vértice inicial de la arista a, así como $a.vf.inf$ es la información del vértice final de la arista a.

Ejemplo 9.6

Para las siguientes interacciones se requieren las definiciones de los vértices hechas en el ejemplo 9.2 y las definiciones de las aristas del ejemplo 9.5.

```
>>> a1.vi
<__main__.Vertice object at 0x7f5b3051bc40>
>>> a1.vi.inf
1
>>> a1.vf
<__main__.Vertice object at 0x7f5b2fed3b80>
>>> a1.vf.inf
2
>>> a1.inf
[1,2]
>>>
```

Si $A = \{a_1, \ldots, a_n\}$ es una colección finita de aristas, `AInf`(A) genera una lista con la información de cada arista en A. El resultado es formado por el atributo `inf` de cada una de las aristas $a_i \in A$.

Código 9.2: *Información de una lista de aristas*

```
1 def AInf(A:list)-> list:
2     """
3     Obtiene la información de todas las aristas de la lista dada.
4     """
5     return enCada(lambda a: a.inf, A)
```

Ejemplo 9.7

Consideremos nuevamente las aristas definidas en el ejemplo 9.5.

```
>>> A
[<__main__.Arista at 0x7fa7cc50d2b0>,
 <__main__.Arista at 0x7fa7cc50d250>,
 <__main__.Arista at 0x7fa7cc50da00>,
 <__main__.Arista at 0x7fa7cc50d460>,
 <__main__.Arista at 0x7fa7cc50d340>,
 <__main__.Arista at 0x7fa7cc50d160>]
>>> car(A).inf
```

```
[1,2]
>>> AInf(A)
[[1, 2], [2, 1], [2, 3], [3, 5], [5, 2], [5, 3]]
>>>
```

Aristas iguales

La igualdad entre aristas está determinada en parte por la igualdad de los vértices que la constituyen, de acuerdo con la siguiente definición.

Definición 9.1.6 Si a y b son aristas, decimos que a y b son iguales si ambas aristas tienen el mismo vértice inicial y el mismo vértice final.

Código 9.3: *Aristas iguales*

```
1 def aIguales(a:Arista, b:Arista):
2     """
3     Determina si dos aristas son iguales.
4     """
5     return y(vIguales(aVi(a), aVi(b)), vIguales(aVf(a),aVf(b)))
```

A la hora de la programación, es importante saber cómo y con qué elementos se ha definido una arista, porque es muy posible que se generen objetos diferentes con la misma información, ocasionando un mal uso en la memoria o un mal comportamiento del programa.

Ejemplo 9.8

Definiremos un par de aristas $b_1 \leftrightarrow \langle 3,4\rangle$ y $b_2 = \langle 3,4\rangle$:

```
>>> b1 = Arista(3,4)
>>> b2 = Arista(3,4)
>>> aIguales(b1,b2)
True
>>> b1
<__main__.Arista at 0x7fa7cc093220>
>>> b2
<__main__.Arista at 0x7fa7cc090970>
>>>
```

El objeto `b1` es diferente al objeto `b2` puesto que están alojados en direcciones de memoria diferentes, pero como tienen la misma información, se consideran aristas iguales. De hecho, los vértices con los que se han creado las aristas, también son objetos diferentes.

9.1.4 Estructura de grafo

De acuerdo con la definición 9.1.1, un grafo es una estructura $[\![V;A]\!]$ donde $V \neq \varnothing$ es un conjunto de vértices y A es un conjunto de aristas.

En la literatura de la teoría de grafos, estos se clasifican en grafos dirigidos y grafos no dirigidos. La diferencia es que los grafos dirigidos se definen con un conjunto de aristas que son tuplas de dos vértices; mientras que en los grafos no dirigidos, las aristas son conjuntos de dos vértices.

Definición 9.1.7 -- Grafo no dirigido. Un grafo no dirigido es una clase $[\![V;A]\!]$ donde:
$V \neq \varnothing$**:** Es un conjunto denominado los **vértices** del grafo no dirigido.
$A \subseteq \{W \in \mathcal{P}(V) : |W| = 2\}$**:** Es un conjunto de **aristas-no-dirigidas**.

Ejemplo 9.9
Sea $G = [\![V;A]\!]$ un grafo no dirigido, con vértices $V = \{1,2,3,4,5\}$ y aristas $A = \{\{3,2\},\{2,5\},\{3,4\},\{5,4\},\{1,5\},\{1,4\},\{4,3\},\{3,1\}\}$

En los grafos no dirigidos, la arista $\{u,v\}$ compuesta por los vértices u y v es la misma que la arista $\{v,u\}$. Es decir, no hay necesidad de definir dos aristas diferentes que estén compuestas con el mismo par de vértices.

Definición 9.1.8 -- Grafo dirigido. Un grafo dirigido o digrafo es una clase $[\![V;A]\!]$ donde:
$V \neq \varnothing$**:** Es un conjunto denominado los **vértices** del digrafo.
$A \subseteq V \times V$**:** Es un conjunto de **aristas-dirigidas**.

Ejemplo 9.10
Sea $G = [\![V;A]\!]$ un grafo, donde $V = \{1,2,3,4,5\}$ y $A = \{\langle 3,2\rangle,\langle 2,5\rangle,\langle 3,4\rangle,\langle 5,4\rangle,\langle 1,5\rangle,\langle 1,4\rangle,\langle 4,3\rangle,\langle 3,1\rangle\}$

En un grafo dirigido, una arista de la forma $\langle u,v\rangle$ es diferente a una arista $\langle v,u\rangle$, ya que tienen vértices iniciales diferentes y vértices finales diferentes.

La gran mayoría de definiciones hechas para grafos no dirigidos, también sirven para los grafos dirigidos. Como en este libro solamente estudiaremos los grafos dirigidos, cuando se haga referencia a un «grafo», significará un digrafo.

Retomando la definición de grafo como una estructura $[\![V;A]\!]$, creamos una clase `Grafo`, que inicialmente tiene un par de atributos fundamentales, el conjunto de vértices V y el conjunto de aristas A.

La clase puede crecer con más atributos y métodos, como un atributo `Nombre` para identificar el grafo con un nombre; otro atributo puede ser `inf` para ver la información del grafo; incluso podemos agregar un método `Ladys`, para ver la relación $G.A$ como una lista de relaciones [página 140].

Código 9.4: *Clase base* `Grafo`

```
1  class Grafo:
2      """
3      Estructura de vértices relacionados mediante aristas.
4      """
5      def __init__(self, V=None, A=None, Nombre = "G"):
6          self.V = V
7          self.A = A
8          self.Nombre = Nombre
9          self.inf = None if V is None else [VInf(self.V), AInf(self.A)]
10         self.Lady = paresArels(AInf(self.A)) # Ver capítulo 6, ejercicio 6, pág. 145.
11
12     def __str__(self):
13         return f"[{VInf(self.V)}; {AInf(self.A)}]"
```

Creación de digrafos

Para crear un grafo es necesario tener un conjunto de objetos de la clase `Vertice` y un conjunto de objetos de la clase `Arista`.

Ejemplo 9.11
Digamos que $V \leftarrow \{1,2,3,4\}$ y $A \leftarrow \{\langle 1,1\rangle,\langle 1,2\rangle,\langle 1,3\rangle,\langle 2,1\rangle,\langle 2,5\rangle,\langle 3,4\rangle,\langle 3,5\rangle,\langle 4,3\rangle,\langle 4,4\rangle,\langle 4,5\rangle,\langle 5,2\rangle\}$. Construimos el grafo $G_1 = [\![V;A]\!]$ simplemente considerando a V y A, ya no como términos aislados, sino como parte de una estructura.

```
# los @vértices@
>>> v1 = Vertice(1)
>>> v2 = Vertice(2)
>>> v3 = Vertice(3)
>>> v4 = Vertice(4)
>>> v5 = Vertice(5)

>>> V = conj(v1, v2, v3, v4, v5)

# las aristas
>>> a11 = Arista(v1, v1)
>>> a12 = Arista(v1, v2)
>>> a13 = Arista(v1, v3)
>>> a21 = Arista(v2, v1)
>>> a25 = Arista(v2, v5)
>>> a34 = Arista(v3, v4)
>>> a35 = Arista(v3, v5)
>>> a43 = Arista(v4, v3)
>>> a44 = Arista(v4, v4)
>>> a45 = Arista(v4, v5)
>>> a52 = Arista(v5, v2)

>>> A = conj(a11, a12, a13, a21, a25, a34, a35, a43, a44, a45, a52)

# El grafo G1

>>> G1 = Grafo(V, A, "G1")
```

```
>>> VInf(G1.V)
[1, 2, 3, 4, 5]
>>> AInf(G1.A)
[[1, 1], [1, 2], [1, 3], [2, 1], [2, 5], [3, 4], [3, 5], [4, 3], [4, 4], [4, 5], [5, 2]]
>>> G1.Nombre
'G1'
>>> G1.Lady
[[1, 1, 2, 3], [2, 1, 5], [3, 4, 5], [4, 3, 4, 5], [5, 2]]
>>>
```

Es conveniente crear un grafo a partir de la información dada de los vértices y las aristas en lugar de las estructuras de datos, por lo que es necesario una manera de crear un grafo a partir de la información de sus componentes.

Sean $V_{inf} \leftarrow \{v_1, \ldots, v_n\}$ un conjunto de vértices dados por su información; y $A_{inf} \subseteq V_{inf} \times V_{inf}$ un subconjunto de pares de vértices. Un algoritmo para crear un grafo a partir de la información dada por un conjunto de vértices y un conjunto de aristas se puede llamar `genGrafo` y debe hacer lo siguiente:

1. Crear un conjunto `V` de objetos `Vertice` a partir del conjunto V_{inf}.
2. Crear un conjunto `A` de objetos `Arista` a partir del conjunto de vértices `V` recién creado, auxiliados de la información de cada arista que se encuentra en el conjunto A_{inf}. Esto es, tomar el primer par ordenado $\langle u.inf, v.inf \rangle$ en A_{inf} y buscar en `V` aquellos vértices que contengan la información $u.inf$ y $v.inf$, luego crear una arista con esos vértices. Hacer lo propio con el resto de las aristas en A_inf.
3. Crear el objeto `Grafo` con los conjuntos `V` y `A`.

Código 9.5: *Genera un grafo con la información dada*

```
1 def genGrafo(VInfo: list, AInfo:list)-> list:
2     """
3     Crea un grafo tomando la información de los vértices y aristas.
4     """
5     V = enCada(lambda vinf: Vertice(vinf), VInfo)
6     A = []
7     for a in AInfo:
```

```
8           vi = vEnV(a[0], V)
9           vf = vEnV(a[1], V)
10          A += [Arista(vi,vf)]
11      return Grafo(V,A)
```

Ejemplo 9.12

Vamos a crear nuevamente el grafo `G1` del ejemplo 9.11 pero ahora damos los vértices y aristas como información del grafo.

```
>>> G1 = genGrafo([1,2,3,4,5], [[1,1],[1,2],[1,3],[2,1],[2,5],[3,4],[3,5],[4,3],[4,4],
[4,5],[5,2]])
>>> G1
<__main__.Grafo object at 0x7f627137e7f0>
>>>
```

La información de un grafo

Para manipular convenientemente un grafo, es necesario tener control de los elementos que forman la estructura, por lo que requerimos obtener el conjunto de vértices y el de aristas que definen un grafo.

Si $G = [\![V;A]\!]$ es un objeto de la clase `Grafo`, $G.V$ se refiere al conjunto V de vértices del grafo G que es un conjunto de objetos de la clase `Vertice` y $G.A$ se refiere al conjunto A de aristas del grafo G, que es un conjunto de objetos de la clase `Arista`. La información del grafo se obtiene mediante el atributo `inf`, que contiene la información de sus componentes.

Ejemplo 9.13

Sea G_1 el mismo grafo del ejemplo 9.12.

```
>>> print(G1.inf)
[[1, 2, 3, 4, 5], [[1, 1], [1, 2], [1, 3], [2, 1], [2, 5], [3, 4], [3, 5], [4, 3], [4, 4],
 [4, 5], [5, 2]]]
>>>
```

9.1.5 La gráfica de un grafo

Los grafos se llaman así, porque se pueden dibujar y es precisamente por su representación gráfica que es más fácil entender sus propiedades. La representación gráfica más usual consiste en círculos etiquetados [los vértices], que se unen mediante flechas [las aristas]. Las aristas en los grafos dirigidos se represente con flechas, mientras que en lo no-dirigidos se represente con líneas.

Ejemplo 9.14

Sea $F_1 \leftarrow [\![V;A]\!]$ un grafo con vértices $F_1.V \leftarrow \{1,2,3,4,5\}$ y aristas $F_1.A \leftarrow \{\{1,2\},\{1,1\},\{2,3\},\{3,1\}\}$. También $F_2 \leftarrow [\![V;A]\!]$, con vértices $F_2.V \leftarrow \{1,2,3\}$ y con aristas $F_2.A \leftarrow \{\langle 1,2\rangle,\langle 2,1\rangle,\langle 1,1\rangle,\langle 2,3\rangle,\langle 1,3\rangle,\langle 3,1\rangle\}$.

No hay diferencia alguna en la distribución espacial de los vértices y aristas, por lo que la ubicación de los vértices y aristas no es relevante. Por supuesto, para mejorar la legibilidad del grafo, es recomendable que los vértices estén en lugares diferentes entre sí, aunque no tan separados unos de otros.

Es posible saber si un grafo es dirigido o es no-dirigido atendiendo al conjunto de aristas. Un grafo dirigido se puede representar como un grafo no-dirigido si cumple

Figura 9.3: *El grafo F_1 es un grafo no dirigido, mientras que el grafo F_2 es dirigido, porque no todas las aristas que hay son simétricas.*

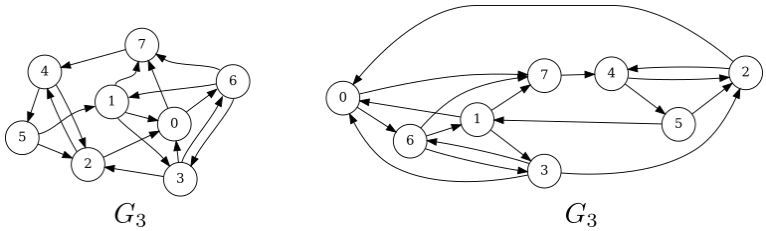


Figura 9.4: *Dos representaciones de $G_3 \hookleftarrow [\![V;A]\!]$, con vértices $V \hookleftarrow \{0,1,2,3,4,5,6,7\}$ y aristas $A \hookleftarrow \{\langle 0,6\rangle,\langle 0,7\rangle,\langle 1,0\rangle,\langle 1,3\rangle,\langle 1,7\rangle,\langle 2,0\rangle,\langle 2,4\rangle,\langle 3,0\rangle,\langle 3,2\rangle,\langle 3,6\rangle,\langle 4,2\rangle,\langle 4,5\rangle,\langle 5,1\rangle,\langle 5,2\rangle,\langle 6,1\rangle,\langle 6,3\rangle,\langle 6,7\rangle,\langle 7,4\rangle\}$.*

$\forall a \in A : a^{-1} \in A$. En la figura 9.3, el grafo F_2 es dirigido pues en la relación inducida por las aristas, no todas las aristas tienen su arista inversa.

Un grafo que no cumple la condición anterior es un grafo dirigido y todas sus aristas deben ser dibujadas con flechas. Un grafo dirigido simétrico no siempre es equivalente a un grafo no dirigido. Una diferencia es que al analizar una arista no dirigida, esta se marca como «ya analizada»; pero cuando se trata de grafos dirigidos, es necesario marcar la característica en ambas aristas simétricas; también es necesario conservar un registro de la dirección.

Aristas particulares

Podemos distinguir aristas que podrían aparecen en un grafo como consecuencia del mismo modelo de relación, por ejemplo en una relación reflexiva, para una relación R en el dominio de las personas, aRb significa que a «tiene el mismo apellido» que b, por lo que se observa que aRa para cualquier elemento a en el dominio de la relación.

Un grafo que represente la relación R anterior ocasiona que existan aristas dirigidas que salgan de un vértice y lleguen al mimo vértice. Este tipo de aristas se llama **autociclos**.

Otro tipo de aristas surge cuando la interpretación de la relación exige poner más de una arista que tenga los mismos vértices y en el mismo orden, por ejemplo en una relación con dominio en lugares, podemos decir que aRb significa que hay una vía que inicia en a y termina en b. Sin embargo en realidad puede haber más de una vía con esas características, lo que ocasionaría que en el grafo que modela la relación sea necesario dibujar aristas diferentes que tengan el mismo origen y el mismo destino. Este tipo de aristas se llaman **aristas paralelas**.

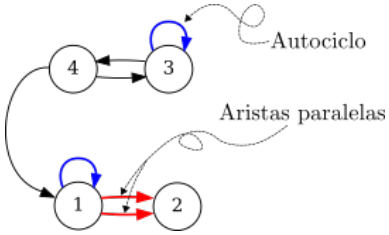


Figura 9.5: *Grafo* $[\![\{1,2,3,4\};\{\langle 1,1\rangle,\langle 3,3\rangle,\langle 1,2\rangle,\langle 1,2\rangle,\langle 3,4\rangle,\langle 4,1\rangle,\langle 4,3\rangle\}]\!]$. *Las aristas* $\langle 3,3\rangle$ *y* $\langle 1,1\rangle$ *son autociclos. Las aristas que inician en 1 y terminan en 2, son aristas paralelas. Las aristas que involucran a los vértices 4 y 3 no son paralelas, porque no tienen los mismos vértices al inicio ni al final.*

9.2 Adyacencia y conexión

Los conceptos de adyacencia y conexión están estrechamente relacionados y son de suma importancia en la teoría de grafos. Gracias a estos conceptos, se puede «viajar» a través del grafo, recorriendo sus vértices y aristas. La conectividad en los grafos que son interpretados como relaciones, tiene mucho que ver con el concepto de la potencia de una relación [definición 7.3.3, página 164].

9.2.1 Vértices adyacentes

Definición 9.2.1 Sea $G \leftarrow [\![V;A]\!]$ un grafo dirigido con $u,v \in G.V$. Decimos que v es adyacente a u si se cumple

$$\langle u,v\rangle \in a.$$

Ejemplo 9.15

Considera ahora el grafo dirigido G_3 de la figura 9.4.

1. El vértice 3 es adyacente al vértice 1, porque $\langle 1,3\rangle \in G.A$.
2. El vértice 7 es adyacente al vértice 1, porque $\langle 1,7\rangle \in G.A$.
3. Como $\langle 6,4\rangle \notin G.A$, sabemos que 4 no es adyacente a 6.
4. Como $\langle 7,6\rangle \notin G.A$, sabemos que 6 no es adyacente a 7, pero $\langle 6,7\rangle \in G.A$, por lo que 7 es adyacente a 6.

Para verificar que un vértice es adyacente desde otro vértice, se crea una arista [como un par de elementos] y se revisa que pertenezca al conjunto de aristas.

Código 9.6: *Verifica que un vértice sea adyacente desde otro*

```
1 def esAdy(v, u, G:Grafo)-> bool:
2     """
3     Determina si el vértice v es adyacente al vértice u en el grafo G.
4     """
5     A = AInf(G.A)
6     return en(tupla(u,v), A)
```

Considera nuevamente el grafo G_3 de la figura 9.4. El vértice 5 es adyacente al vértice 4, pero no en sentido opuesto. La adyacencia se puede ver como la posibilidad de alcanzar un vértices desde otro. Si hay aristas simétricas, la adyacencia está garantizada en ambos sentidos.

```
>>> esAdy(5,4,G3)
True
>>> esAdy(4,5,G3)
```

```
False
>>> esAdy(4,2,G3)
True
>>> esAdy(2,4,G3)
True
>>>
```

Como las aristas $G.A$ forman una relación entre los vértices, los vértices adyacentes se pueden obtener como la imagen de un vértice en la relación dada por $G.A$.

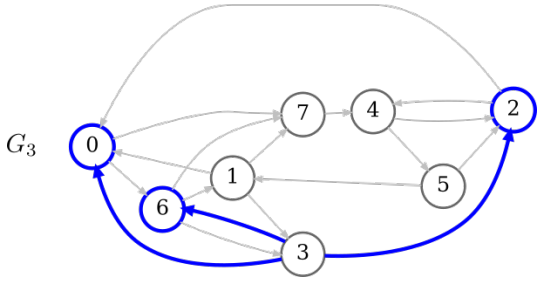


Figura 9.6: *Los vértices 0, 6 y 2 son adyacentes al vértice 3.*

Código 9.7: *Vértices adyacentes a un vértice*

```
1 def vAdy(v, G:Grafo)-> list:
2     """
3     Obtiene la lista de vértices adyacentes a un vértice en un grafo.
4     """
5     return im(AInf(G.A),v)
```

Ejemplo 9.16

En el grafo G_3 de la figura 9.4, los vértices adyacentes de 3 son $\{0, 2, 6\}$.

```
>>> G3 = genGrafo([0,1,2,3,4,5,6,7],
     [[0,6],[0,7],[1,0],[1,3],[1,7],[2,0],[2,4],[3,0],[3,2],
     [3,6],[4,2],[4,5],[5,1],[5,2],[6,1],[6,3],[6,7],[7,4]])
>>> vAdy(1,G3)
[0, 3, 7]
>>> vAdy(3,G3)
[0, 2, 6]
>>>
```

9.2.2 Vértices conectados

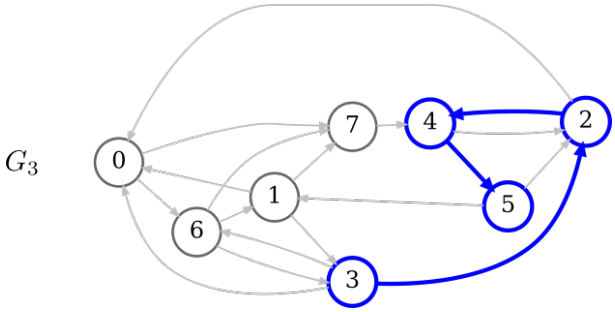
Definición 9.2.2 Dos vértices $u, v \in G.V$ son **vértices conectados** si a partir de u se puede alcanzar v siguiendo aristas que terminen en vértices adyacentes. Si u y v están conectados por ℓ aristas, se denota

$$u \overset{\ell}{\rightsquigarrow} v.$$

Cuando v es adyacente a u, significa que hay una arista $\langle u, v \rangle$, que inicia en u y termina en v, por lo tanto decimos que u y v están **conectados** mediante 1 arista, y lo escribimos $u \overset{1}{\rightsquigarrow} v$. El número indica la cantidad de aristas que deben seguirse para alcanzar el vértice v a partir de u. Cuando $\ell = 1$, generalmente se omite de la notación, pero debe escribirse cuando el número de aristas es mayor a 1.

Ejemplo 9.17

En el grafo G_3, el vértice 3 está conectado con el vértice 5 por 3 aristas, por lo que $3 \overset{3}{\rightsquigarrow} 5$ como se muestra en la figura. También $3 \overset{5}{\rightsquigarrow} 5$, siguiendo las aristas $\langle 3,6\rangle, \langle 6,1\rangle, \langle 1,7\rangle, \langle 7,4\rangle$ y $\langle 4,5\rangle$.



9.2.3 Grado de un vértice

Los vértices de un grafo tienen características en función del número de veces que participa en la relación dada por las aristas. Consideremos un grafo $G \leftarrow [\![V;A]\!]$.

El **grado de entrada** de un vértice $v \in V.G$ se refiere a la cantidad de veces que el vértice aparece como segunda entrada en un par, es decir, la cantidad de aristas que apuntan hacia v. Esta cantidad se puede calcular por

$$\texttt{vGsal}\,(v,G) \leftarrow |\{\langle x,y\rangle \in A.G \,|\, v = y\}|$$

El **grado de salida** de $v \in G.V$ es el número de veces que v aparece como primera entrada en las aristas, esto es, la cantidad de aristas que inician en v, sin importar el vértice hacia adonde apunte.

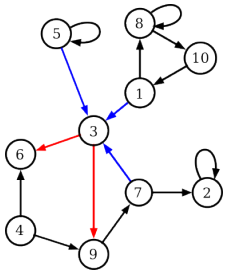
$$\texttt{vGent}\,(v,G) \leftarrow |\{\langle x,y\rangle \in A.G \,|\, v = x\}|$$

El grado de un vértice es simplemente la suma del grado de entrada y el grado de salida del vértice en el grafo.

$$\texttt{vGr}\,(v,G) \leftarrow \texttt{vGent}\,(v,G) + \texttt{vGsal}\,(v,G).$$

Ejemplo 9.18

Sea G el grafo con vértices $\{4,6,1,8,3,2,7,5,9,10\}$ y aristas $\{\langle 1,8\rangle, \langle 3,6\rangle, \langle 2,2\rangle, \langle 1,3\rangle, \langle 7,3\rangle, \langle 7,2\rangle, \langle 5,5\rangle, \langle 9,7\rangle, \langle 10,1\rangle, \langle 4,9\rangle, \langle 5,3\rangle, \langle 3,9\rangle, \langle 8,10\rangle, \langle 8,8\rangle, \langle 4,6\rangle\}$.



- $\texttt{vGent}\,(3,G) \mapsto 3$.
- $\texttt{vGsal}\,(3,G) \mapsto 2$.
- $\texttt{vGr}\,(3,G) \mapsto 5$.

9.3 Tipología en digrafos

9.3.1 Grafo nulo, trivial y simple

Si un grafo $G \leftarrow [\![V;A]\!]$ es tal que $G.V = \varnothing$ y en consecuencia también $G.A = \varnothing$, entonces G se denomina **grafo nulo**. Como el grafo nulo no tiene vértices ni aristas, tampoco tiene representación gráfica. El grafo nulo existe únicamente por conveniencia matemática y computacional.

Si un grafo G tiene un conjunto de vértices tal que $|G.V| = 1$, entonces lo podemos llamar **grafo trivial**. Todos los demás grafos son **no-triviales**.

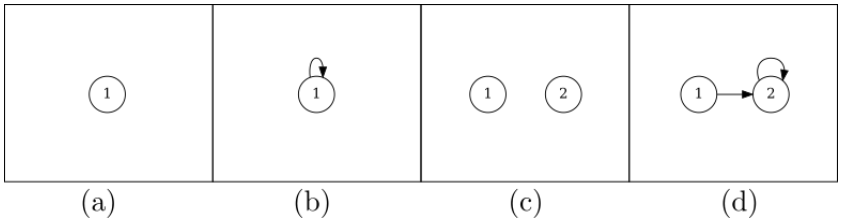


Figura 9.7: *(a) y (b) Grafos triviales. (c) y (d) Grafos no triviales. El caso particular de (c), es un grafo con dos vértices y ninguna arista.*

Decimos que un grafo es **simple**, si no tiene ni aristas paralelas ni autociclos [BM76, p. 4]. Muchos resultados en la teoría de grafos exigen que el grafo sea simple.

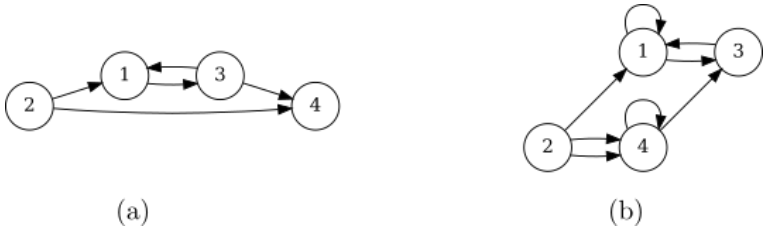


Figura 9.8: *(a) Grafo simple. (b) Grafo no simple, pues tiene autociclos y aristas paralelas.*

Un **multigrafo** es un grafo que permite aristas paralelas y autociclos. De modo que los grafos simples también son multigrafos. En este libro trataremos con multigrafos que no permiten las aristas paralelas, aunque algunos casos si permitiremos los autociclos, particularmente para modelar relaciones de equivalencia.

9.3.2 Subgrafos

Definición 9.3.1 -- Subgrafo. Sean $G \leftarrow [\![V;A]\!]$ y $H \leftarrow [\![V;A]\!]$ dos grafos. Decimos que G es un subgrafo de H y lo denotamos como $G \subseteq H$ si se cumple:

$$G.V \subseteq H.V \wedge G.A \subseteq H.A$$

Los subgrafos son importantes porque expresan situaciones que simplifican una realidad que ha sido modelada mediante un grafo, por ejemplo un grafo con menos aristas pero que garantizan las mismas propiedades sobre los vértices; o bien grafos

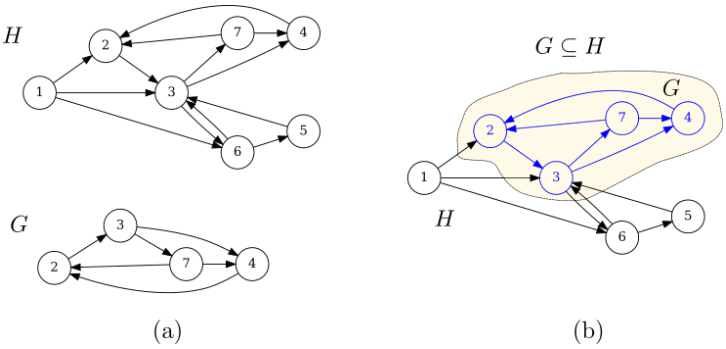
con menos vértices y menos aristas como una subred de computadoras o de tendido eléctrico.

Código 9.8: *Subgrafo de un grafo*

```
def esSubgrafo(G:Grafo, H:Grafo):
    """
    Determina si un grafo es subgrafo de otro.
    """
    return y(esSubc(VInf(G.V), VInf(H.V)), esSubc(AInf(G.A), AInf(H.A)))
```

Ejemplo 9.19

Considera los grafos G y H que se muestra abajo a la izquierda. El grafo G es subgrafo de H porque sus vértices aparecen en H y sus aristas también aparecen en H. La posición de los vértices y aristas en la imagen es irrelevante.




```
>>> H = genGrafo(genGrafo([1,2,3,4,5,6,7], [[1,2],[1,3],[1,6],[2,3],[3,4],[3,6],[3,7],
[4,2],[5,3],[6,3],[6,5],[7,2],[7,4]]))
>>> G = genGrafo([2,3,4,7],[[2, 3], [3, 4], [3, 7], [4, 2], [7, 2], [7, 4]])
>>> esSubgrafo(G,H)
True
>>> esSubc(AInf(G.A), AInf(H.A))
True
>>> esSubc(VInf(G.V), VInf(H.V))
True
>>>
```

Subgrafos inducidos

Con la consideración anterior se pueden obtener subgrafos dando o bien el subconjunto de vértices o el subconjunto de aristas. Al primer método le podemos llamar **subgrafo inducido por vértices**, y al segundo método le podemos llamar **subgrafo inducido por aristas**.

De este modo si $H \leftarrow [\![V;A]\!]$ es un grafo, podemos crear un subgrafo $G \leftarrow [\![V';A']\!]$ inducido por los vértices seleccionados $V' \subseteq H.V$ y seleccionando de H, solamente las aristas que tienen ambos vértices en V'.

$$A' \leftarrow \{\langle u,v\rangle \in H.A | u \in V' \wedge v \in V'\}$$

También podemos crear un subgrafo inducido por un conjunto seleccionado de aristas $A' \subseteq H.A$ y seleccionando los vértices que intervienen en las aristas.

$$V' \leftarrow \{v \in H.V | \exists \langle a,b\rangle \in H.A : v = a \vee v = b\}$$

Como consecuencia de que G sea un subgrafo de H, se tiene que H es un **supergrafo** de G. Este concepto es análogo a los superconjuntos en la teoría general de conjuntos.

9.3.3 Familias de grafos

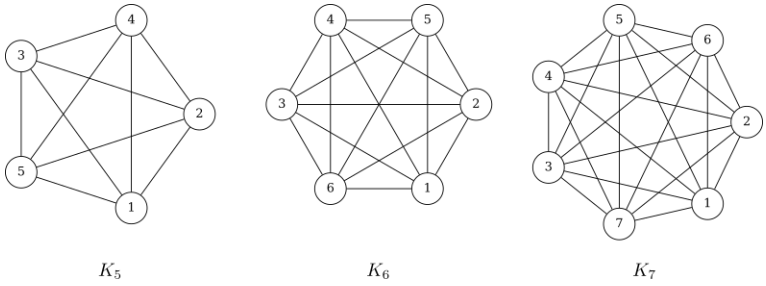
Grafos completos

Definición 9.3.2 Sea $G \leftarrow [\![V;A]\!]$ un grafo simple. Decimos que G es un grafo completo de orden $n \geq 2$, y lo escribimos K_n, si cuando A representa una relación simétrica e irreflexiva:

$$|V| = n \wedge \forall u \in V : \mathtt{im}(u,A) = V \backslash \{u\}.$$

Ejemplo 9.20

Se muestra los grafos completos K_5, K_6 y K_7.



K_5 K_6 K_7

Observa que las aristas representan una relación simétrica, es decir cada arista en la figura significa en realidad un par de aristas en el digrafo, una de ida y otra de regreso.

Cada vértice en un grafo completo tiene conexión con cada uno de los $n-1$ vértices restantes, por lo que el número de aristas individuales en el grafo es $n(n-1)$. Al tratarse de una relación simétrica, generalmente se dibujan las aristas como líneas, sin considerar su dirección. De modo que en un grafo completo de orden n, el número de aristas no dirigidas es

$$\frac{n(n-1)}{2}.$$

Grafos tipo ciclo

Definición 9.3.3 Un grafo ciclo $C_n \leftarrow [\![V;A]\!]$ de tamaño n, es un grafo simple con $n \geq 2$ vértices, donde un vértice está conectado con otro, ese otro con un siguiente y así sucesivamente hasta conectar el último con el primero.

Los grafos ciclos pueden ser simple o doblemente dirigidos, como se muestra en la figura 9.9. Hay aplicaciones en las que se requiere un modelo cíclico, particularmente en Ciencias Computacionales este tipo de grafos se conocen como estructuras de datos de tipo lista circular y lista circular doblemente ligada.

Hay muchas aplicaciones en las que se utilizan grafos de tipo ciclo, ya sea simple o doblemente ligados, por ejemplo al diseñar una aplicación de reproductor de música en la que se ponga un botón de «avance» y «retroceso» en el caso del grafo bidireccional .

Otra situación común es en redes de computadoras, donde las computadoras se colocan en una topología conocida como «anillo» [Ren13]. En esta configuración, cada vértice representa una computadora conectada a la red en la forma que se muestra en la figura 9.9.

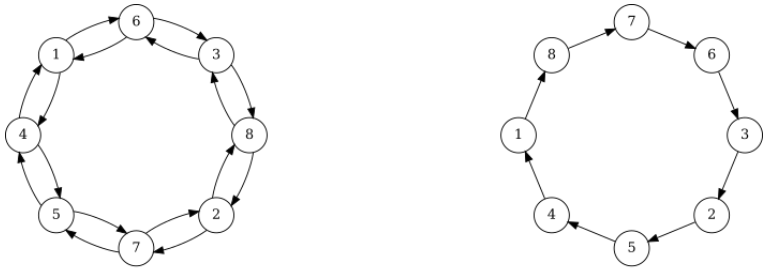


Figura 9.9: *Diferentes configuraciones de un grafo ciclo de tamaño 7. (a) completo, (b) simplemente dirigido.*

Grafos tipo rueda

Definición 9.3.4 Un grafo rueda $R_n \hookleftarrow [\![V;A]\!]$ de tamaño n, es un grafo simple con $n \geq 4$ vértices, construido de tal forma que todos los vértices, excepto uno llamado «el centro», forman un ciclo; y el centro de la rueda se conecta a cada uno de los otros $n-1$ vértices.

Los grafos de tipo rueda pueden servir de modelo para ciertas aplicaciones, en particular en redes de computadoras. Usualmente en una red de computadoras de tipo anillo, hay un servicio que el resto de computadoras requiere, así que las solicitudes se pueden modelar como en la figura 9.10 (b), mientras que la respuesta a tales solicitudes se puede modelar como en la figura 9.10 (c). Ambas situaciones, tanto la solicitud como el servicio prestado se pueden modelar como en (a).

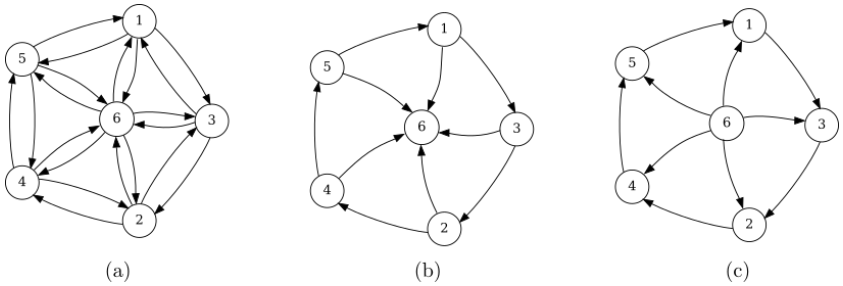


Figura 9.10: *Diferentes configuraciones de un grafo rueda de tamaño 6. (a) completa, (b) convergente, (c) divergente.*

9.3.4 Grafo bipartita

Definición 9.3.5 -- Grafo bipartita. Un grafo $G \hookleftarrow [\![V;A]\!]$ es bipartita cuando existen subconjuntos $A \subset V$ y $B \subset V$ escogidos de tal modo que $(A \cup B) = V \wedge A \cap B = \varnothing$ y se debe cumplir que:

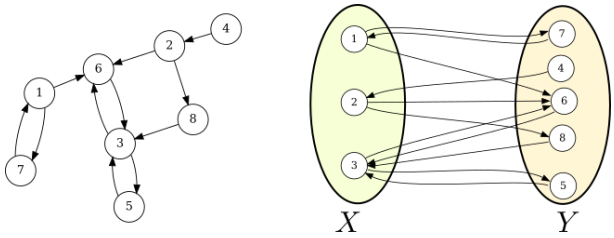
$$\forall \langle v_i, v_f \rangle \in A : (v_i \in A \wedge v_f \in B) \oplus (v_i \in B \wedge v_f \in A).$$

Los grafos bipartitas se llaman así porque tienen la característica de que sus vértices, en el modo en como están conectados, forman una partición $\{X, Y\}$ con X y Y dos subconjuntos de V.

Cada subconjunto de la partición $\{X, Y\}$ agrupa a vértices que están conectados con vértices que pertenecen al otro subconjunto. Así, si $u \in X$ entonces u está conectado con al menos un vértice $v \in Y$. Si, por el contrario, $u \in Y$ entonces está conectado con al menos un vértice $v \in X$.

Ejemplo 9.21

El siguiente grafo es un grafo bipartita, donde los vértices están distribuidos en $G.V = \{1,2,3\} \cup \{4,5,6,7,8\}$ y las aristas $G.A \leftarrow \{\langle 1,7\rangle, \langle 1,6\rangle, \langle 2,6\rangle, \langle 2,8\rangle, \langle 3,6\rangle, \langle 3,5\rangle, \langle 4,2\rangle, \langle 5,3\rangle, \langle 6,3\rangle, \langle 7,1\rangle, \langle 8,3\rangle\}$ como se muestra:



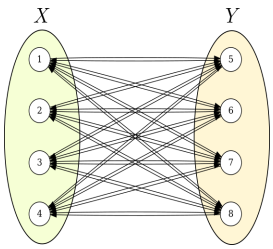
No existe vértice de un subconjunto que conecte con un vértice del mismo subconjunto.

Un grafo bipartita es **completo** si cada vértice de una partición está conectado con todos los vértices de la otra partición.

Ejemplo 9.22

Grafo bipartita completo.

$V = \{1,2,3,4\} \cup \{5,6,7,8\}$ y $A \leftarrow \{\langle 1,5\rangle, \langle 1,6\rangle, \langle 1,7\rangle, \langle 1,8\rangle, \langle 2,5\rangle, \langle 2,6\rangle, \langle 2,7\rangle, \langle 2,8\rangle, \langle 3,5\rangle, \langle 3,6\rangle, \langle 3,7\rangle, \langle 3,8\rangle, \langle 4,5\rangle, \langle 4,6\rangle, \langle 4,7\rangle, \langle 4,8\rangle, \langle 5,1\rangle, \langle 5,2\rangle, \langle 5,3\rangle, \langle 5,4\rangle, \langle 6,1\rangle, \langle 6,2\rangle, \langle 6,3\rangle, \langle 6,4\rangle, \langle 7,1\rangle, \langle 7,2\rangle, \langle 7,3\rangle, \langle 7,4\rangle, \langle 8,1\rangle, \langle 8,2\rangle, \langle 8,3\rangle, \langle 8,4\rangle\}$



Observa que si $G \leftarrow [\![V; A]\!]$ es un grafo bipartita con particiones $\{X, Y\}$, y $u \in X$; al ver el conjunto de aristas como una relación, $\mathtt{im}(A, u) \subseteq Y$. Lo que significa que todos los vértices adyacentes a $u \in X$ son elementos de la otra partición Y, a su vez, la imagen de cada vértice en Y es subconjunto de X.

```
Algoritmo esBipartita(G):
Requiere: G = (V;A) un grafo
Devuelve: booleano

 1. V ↤ VInf(G.V)
 2. A ↤ AInf(G.A)
 3. X ↤ {car(V)}
 4. Y ↤ ∅
 5. Mientras que V ≠ ∅:
 6.     Si ((Im(A,X) ∪ Preim(A,X)) ⊆ Y) ∧ ((Im(A,Y) ∪ Preim(A,Y)) ⊆ X):
 7.         X ↤ car(V) ▷ X
```

```
 8.          V ↤ cdr(V)
 9.      Y ↤ Union(Y,Im(A,X),Preim(A,X)) # Ver la página 190
10.      X ↤ Union(X,Im(A,Y),Preim(A,Y))
11.      V ↤ V\(X ∪ Y)
12. Si (X ∩ Y) = ∅: ↦ True, en otro caso ↦ False.
```

La idea general del algoritmo es construir una bipartición. Se eligen dos conjuntos, uno de ellos con un vértice y el otro vacío: $X \leftarrow \{\texttt{car}(V)\}$ y $Y \leftarrow \varnothing$. Mientras que aún haya vértices en V:

1. Se agregan a Y los elementos en la imagen de X sobre la relación inducida por A, también se agregan los vértices en la preimagen del conjunto X sobre A.
2. Se hace lo propio con el conjunto X. Se agregan los elementos en la imagen de Y sobre A, junto con los vértices de la preimagen de Y sobre A.
3. Se retiran de V los elementos que fueron agregados a X y a Y.

En ocasiones el grafo puede contener subgrafos desconectados, por lo que cabe la posibilidad de caer en un ciclo infinito. Para evitarlo [línea 6], se agrega cualquier vértice de los restantes a X, que ha sido elegido arbitrariamente y se retira de V [líneas 7 y 8].

El procedimiento devuelve `True` si la intersección de ambos conjuntos X y Y es vacía, de otro modo devuelve `False`.

Ejemplo 9.23

Determina si el grafo $G \leftarrow [\![V;A]\!]$ es bipartita. Con vértices $V \leftarrow \{1,2,3,4,5,6,7,8,9,10,11\}$ y aristas $A \leftarrow \{\langle 1,7\rangle,\langle 3,8\rangle,\langle 4,9\rangle,\langle 4,10\rangle,\langle 5,9\rangle,\langle 5,11\rangle,\langle 7,2\rangle,\langle 8,1\rangle,\langle 10,4\rangle,\langle 10,5\rangle,\langle 11,5\rangle\}$.

De acuerdo con el algoritmo `esBipartita`:

- Se inicia con $X \leftarrow \{1\}$, $Y \leftarrow \{\}$ y $V \leftarrow \{1,2,3,4,5,6,7,8,9,10,11\}$.
- Se agregan los nuevos elementos a los conjuntos Y y X:
  - Se agrega a Y la imagen y preimagen de X, esto es $Y \leftarrow \{\} \cup \{7,8\} = \{7,8\}$.
  - Se agrega a X la imagen y preimagen de Y, $X \leftarrow \{1\} \cup 2 = \{1,2,3\}$.
  - Se actualiza $V \leftarrow \{4,5,6,9,10,11\}$.
- Como la imagen o preimagen de ambos conjuntos está contenida en los conjuntos X y Y, se ha detectado un cluster, por lo que arbitrariamente agregamos un nuevo vértice a X.
  - $X \leftarrow X \cup \{\texttt{car}(V)\} = X \cup \{4\} = \{1,2,3,4\}$.
  - Retiramos de V el elemento agregado. $V \leftarrow \{5,6,9,10,11\}$.
- Se agregan los nuevos elementos a los conjuntos Y y X:
  - Se agrega a Y la imagen y preimagen de X, $Y \leftarrow \{8,7,9,10\}$.
  - Se agrega a X la imagen y preimagen de Y, $X \leftarrow \{3,1,4,5,2\}$.
  - Actualizamos $V \leftarrow \{6,11\}$.
- Se agregan los nuevos elementos a los conjuntos Y y X
  - $Y \leftarrow \{9,10,11,8,7\}$.
  - $X \leftarrow \{2,1,4,5,3\}$.
  - Actualizamos $V \leftarrow \{6\}$
- Se ha detectado un nuevo cluster. Arbitrariamente agregamos un nuevo vértice a X.
  - $X \leftarrow X \cup \{\texttt{car}(V)\} = X \cup \{6\} = \{6,2,1,4,5,3\}$.
  - Retiramos de V el elemento agregado. $V \leftarrow \{\}$.
- Se intentan agregar nuevos elementos a Y y a X, pero ya no hay.
- Como ya no hay elementos en V y $X \cap Y = \varnothing$, se asegura que el grafo es bipartita.

Las particiones construidas son: $X \leftarrow \{4,5,1,3,2,6]\}$ y $Y \leftarrow \{7,8,10,11,9\}$.

9.4 Operaciones con grafos

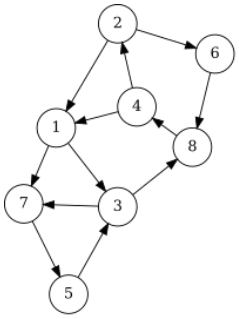
9.4.1 Grafo inverso

Definición 9.4.1 Si $G \leftarrow [\![V;A]\!]$ es un grafo, el grafo inverso de G se denota G^{-1} y es la estructura $G^{-1} \leftarrow [\![V;A^{-1}]\!]$, donde $A^{-1} \leftarrow \texttt{rInv}(A)$.

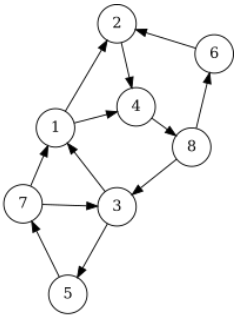
Observa que las aristas A son tratadas como una relación, entonces A^{-1} representa la relación inversa de A.

Ejemplo 9.24

El grafo $G = [\![V;A]\!]$ mostrado abajo a la izquierda, tiene su grafo inverso G^{-1}.



G



G^{-1}

Observa, por ejemplo, que las aristas que salen del vértice 1 en G, en el grafo inverso G^{-1} son aristas que llegan a 1. Todas las aristas cambian de dirección.

El grafo inverso de un grafo no dirigido es él mismo. Esto es así porque cada arista en un grafo no dirigido, representa de hecho un par de aristas dirigidas en sentidos opuestos en un grafo dirigido.

Código 9.9: *Grafo inverso de un grafo*

```
1  def gInv(G:Grafo)-> Grafo:
2      """
3      Genera el grafo inverso de un grafo.
4      """
5      Ainv =rInv(AInf(G.A))
6      return genGrafo(VInf(G.V), Ainv)
```

9.4.2 Grafo complementario

Definición 9.4.2 Si $G \leftarrow [\![V;A]\!]$ es un grafo. El grafo complementario de G se denota $G^c \leftarrow [\![V;A^c]\!]$, donde $A^c \subseteq V \times V$ definido por:

$$A^c \leftarrow \{\langle u,v\rangle \in V \times V \,|\, \langle u,v\rangle \notin A\}$$

El grafo complementario tiene las aristas no consideradas en $V \times V$. Donde la matriz de pertenencia de G tiene entradas 0, en la matriz de pertenencia de G^c son 1.

Ejemplo 9.25

Si $G \leftarrow [\![V;A]\!]$ tiene vértices $G.V \leftarrow \{1,2,3,4,5,6\}$ y aristas $G.A \leftarrow \{\langle 6,6\rangle,\langle 6,1\rangle,\langle 4,5\rangle,\langle 2,4\rangle,\langle 2,6\rangle,\langle 5,4\rangle,\langle 5,6\rangle,\langle 1,3\rangle,\langle 2,1\rangle,\langle 2,3\rangle\}$, el grafo complementario G^c tiene vértices $G^c.V \leftarrow \{1,2,3,4,5,6\}$ y aristas $G^c.A\{\langle 2,2\rangle,\langle 2,5\rangle,\langle 6,2\rangle,\langle 6,5\rangle,\langle 6,4\rangle,\langle 6,3\rangle,\langle 5,2\rangle,\langle 5,5\rangle,\langle 5,1\rangle,\langle 5,3\rangle,\langle 4,2\rangle,\langle 4,6\rangle,\langle 4,4\rangle,\langle 4,1\rangle,\langle 4,3\rangle,\langle 1,2\rangle,\langle 1,6\rangle,\langle 1,5\rangle,\langle 1,4\rangle,\langle 1,1\rangle,\langle 3,2\rangle,\langle 3,6\rangle,\langle 3,5\rangle,\langle 3,4\rangle,\langle 3,1\rangle,\langle 3,3\rangle\}$.

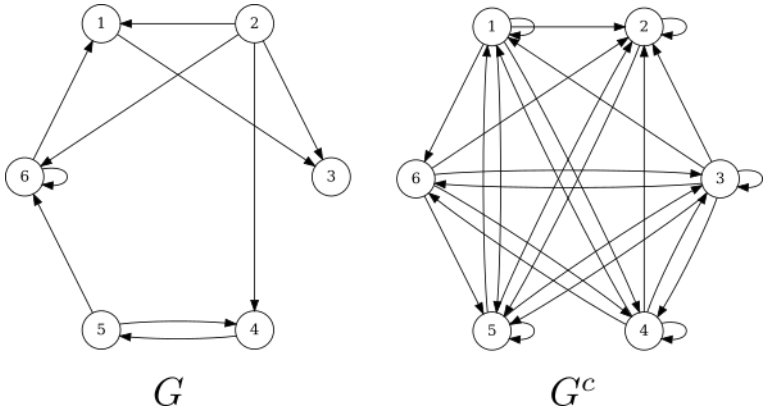


Figura 9.11: *Grafo complementario. En G^c se encuentras las aristas que no están en G.*

$$M_G \leftarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & 1 & 1 & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & 1 \\ 1 & \cdot & \cdot & \cdot & \cdot & 1 \end{pmatrix} \end{array} \qquad M_{G^c} \leftarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & 1 & \cdot & 1 & 1 & 1 \\ \cdot & 1 & \cdot & \cdot & 1 & \cdot \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & \cdot & 1 \\ 1 & 1 & 1 & \cdot & 1 & \cdot \\ \cdot & 1 & 1 & 1 & 1 & \cdot \end{pmatrix} \end{array}$$

Observa que en M_{G^c} hay entradas con 1 en aquellos lugares que hay un 0 [punto] en M_G.

Código 9.10: *Grafo complementario*

```
def gCompl(G:Grafo)-> Grafo:
    """
    Crea el grafo complementario de un grafo.
    """
    V = VInf(G.V)
    A = AInf(G.A)
    Ac = subc(lambda a:neg(en(a, A)), pCart(V,V))
    return genGrafo(V,Ac)
```

9.4.3 Unión de grafos

Definición 9.4.3 Si $G_1 \leftarrow [\![V;A]\!]$ y $G_2 \leftarrow [\![V;A]\!]$ son dos grafos, la unión de G_1 con G_2 es denotada $G_1 \cup G_2$, y es un nuevo grafo $G = (G_1 \cup G_2) \leftarrow [\![V;A]\!]$ donde $G.V \leftarrow G_1.V \cup G_2.V$ y $G.A \leftarrow G_1.A \cup G_2.A$.

Cuando se hacen operaciones con grafos, es importante no perder de vista que el grafo es una representación visual de una relación, por lo que cada vértice y arista tienen un significado. El resultado de una operación con grafos debe tener también una interpretación.

Por ejemplo, supongamos que los números en $V \leftarrow \{1,2,3\}$ representan a cada uno de las personas que compiten por un puesto. Se hace una relación xRy donde x está relacionado con y si x obtiene una mejor puntuación que y. Así se pueden tener, por ejemplo, la relación $\{\langle 1,3\rangle, \langle 3,2\rangle, \langle 1,2\rangle\}$. Llamemos $G \leftarrow [\![V,R]\!]$ al grafo que representa esta relación.

Por otro lado, se tiene una mismo conjunto $\{1,2,3\}$, pero ahora estos tres números representan platillos en un menú de alimentos. Con estos objetos se crea una relación xSy donde x está relacionado con y si x tiene los mismos ingredientes que y. Así se puede tener un conjunto de pares $\{\langle 1,1\rangle,\langle 2,2\rangle,\langle 3,3\rangle,\langle 3,2\rangle,\langle 2,3\rangle\}$. Llamemos $H \leftarrow [\![V;S]\!]$ al grafo de esta otra relación.

¿Qué sentido tiene $G \cup H$?, si tiene alguno, es difícil saberlo. Las operaciones matemáticas con los objetos deben estar siempre «atadas» a un significado. Las definiciones se han puesto aquí porque matemáticamente es posible hacerlo, pero no significa que se deban utilizar sin sentido.

Ejemplo 9.26

En un evento hay 2 jueces, que individualmente observan y hacen anotaciones del mismo evento. En el evento participan 5 personas, identificadas por cada uno de los números en $V \leftarrow \{1,2,3,4,5\}$. Cada juez debe anotar qué persona se ha comunicado con qué persona. Así el primer juez anota $G_1.A \leftarrow \{\langle 2,3\rangle,\langle 2,1\rangle,\langle 4,2\rangle,\langle 2,4\rangle,\langle 3,5\rangle,\langle 4,1\rangle\}$, diciendo con esto que el participante 2 se comunicó con el participante 3; el 2 con el 1; el 4 con el 2 y así sucesivamente [ver figura 9.12]; el segundo juez anota $G_2.A \leftarrow \{\langle 4,1\rangle,\langle 1,5\rangle,\langle 2,1\rangle,\langle 3,5\rangle,\langle 2,3\rangle\}$ con una interpretación similar [ver figura 9.12].

Al final, ambos jueces reúnen sus anotaciones para formar el reporte $G \leftarrow G_1 \cup G_2$ conformado por V y $G.A \leftarrow \{\langle 2,3\rangle,\langle 2,1\rangle,\langle 4,2\rangle,\langle 2,4\rangle,\langle 3,5\rangle,\langle 4,1\rangle,\langle 1,5\rangle\}$, que reúne la información de ambos jueces [ver figura 9.12].

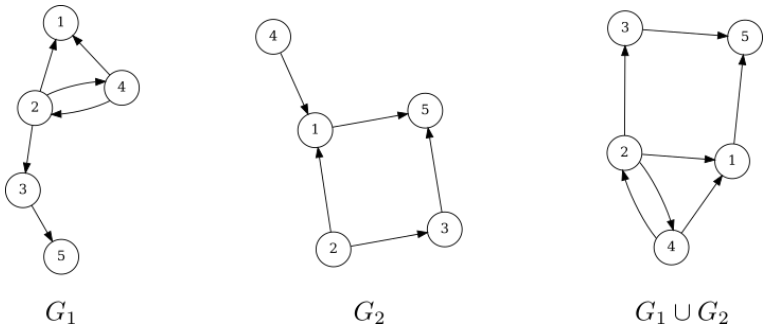


Figura 9.12: *Unión de grafos*

Además de la unión, con los grafos es posible realizar otras operaciones definidas para conjuntos como la intersección, la diferencia y la diferencia simétrica. Esto es posible porque las aristas se pueden ver como una relación entre los vértices.

9.4.4 Potencia de un grafo

Una operación que resalta por su importancia es la potencia de un grafo, que está basada en la operación de la potencia de una relación [definición 7.3.3, ver página 164].

Definición 9.4.4 Sean $G \leftarrow [\![V;A]\!]$ y $n \in \mathbb{Z}^*$, la n-ésima potencia de G es un nuevo grafo $G^n \leftarrow [\![V;A]\!]$, con $G^n.V = G.V$ y $G^n.A = \texttt{rPot}\,(A,n)$.

Ejemplo 9.27

Una compañía de envíos, recorre 7 destinos, que por conveniencia se han indexado con los números $D \leftarrow \{1,2,3,4,5,6,7\}$. Los vehículos de la compañía van de ciudad en ciudad como lo establece la

siguiente relación $A \leftarrow \{\langle 4,3\rangle, \langle 5,2\rangle, \langle 6,3\rangle, \langle 3,1\rangle, \langle 6,5\rangle, \langle 2,4\rangle, \langle 7,6\rangle, \langle 2,7\rangle, \langle 3,7\rangle, \langle 1,3\rangle\}$, diciendo con esto que si $\langle u,v\rangle \in A$, significa que un vehículo puede salir de la ciudad u y llegar a la ciudad v. El mapa de las ciudades y las rutas posibles, se muestra en el grafo $G \leftarrow [\![V;A]\!]$ de la figura 9.13.

En la figura 9.13 en medio, se muestra el grafo que corresponde a $G^2 \leftarrow [\![V;A^2]\!]$ y a la derecha $G^3 \leftarrow [\![V;A^3]\!]$.

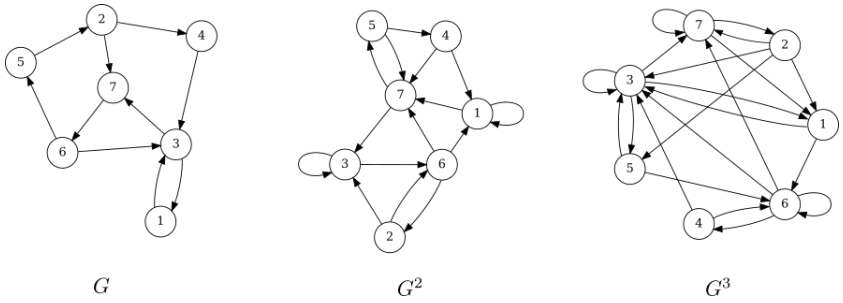


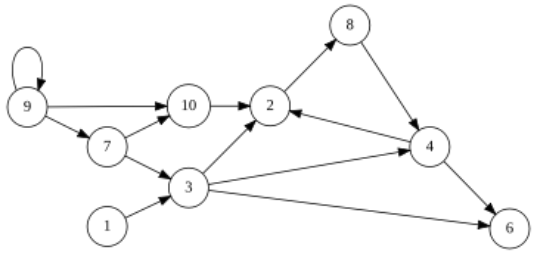
Figura 9.13: *Potencia de un grafo*

La interpretación es que si hay aristas $\langle u,v\rangle \in G.A$ y $\langle v,w\rangle \in G.A$, entonces habrá una arista $\langle u,w\rangle \in G.A^2$; si además hay una arista $\langle w,x\rangle \in G.A$, entonces habrá una arista $\langle u,x\rangle \in G.A^3$. En términos de los vehículos de la empresa, en G^2 se muestran los destinos alcanzables en 2 pasos en G, mientras que en G^3 se muestran los destinos alcanzables en 3 pasos en G.

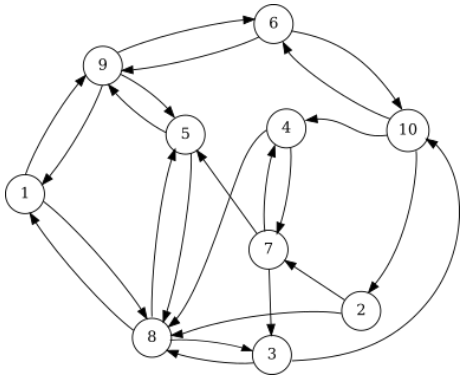
Ejercicios

Nota: Los grafos en esta sección tienen vértices $V \leftarrow \{1,2,3,4,5,6,7,8,9,10\}$.

1. Determina si los siguientes grafos puede ser dibujados como grafos dirigidos o no dirigidos.
   a) $G \leftarrow [\![V;A]\!]$ con $A \leftarrow \{\langle 4,1\rangle,\langle 4,2\rangle,\langle 5,2\rangle,\langle 10,1\rangle,\langle 2,4\rangle,\langle 5,9\rangle,\langle 7,2\rangle,\langle 7,9\rangle,\langle 1,10\rangle,\langle 8,7\rangle,\langle 8,8\rangle,\langle 10,4\rangle,\langle 7,6\rangle,\langle 9,7\rangle,\langle 7,7\rangle,\langle 10,9\rangle,\langle 9,10\rangle,\langle 3,9\rangle,\langle 6,7\rangle,\langle 4,10\rangle,\langle 9,5\rangle,\langle 2,7\rangle,\langle 9,3\rangle,\langle 1,4\rangle,\langle 7,8\rangle,\langle 2,5\rangle\}$
   b) $G \leftarrow [\![V;A]\!]$ con $A \leftarrow \{\langle 5,8\rangle,\langle 2,7\rangle,\langle 3,9\rangle,\langle 3,10\rangle,\langle 10,6\rangle,\langle 9,2\rangle,\langle 5,1\rangle,\langle 4,9\rangle,\langle 3,4\rangle,\langle 4,1\rangle,\langle 1,6\rangle,\langle 7,2\rangle,\langle 10,3\rangle,\langle 3,7\rangle,\langle 6,6\rangle,\langle 7,5\rangle,\langle 10,7\rangle,\langle 2,4\rangle,\langle 9,6\rangle\}$
2. Muestra la representación gráfica de los dos grafos anteriores.
3. Analiza los siguientes grafos y menciona qué propiedades tiene la relación dada por el conjunto de aristas de cada grafo.
   a) $G \leftarrow [\![V;A]\!]$ con $A \leftarrow \{\langle 4,2\rangle,\langle 6,3\rangle,\langle 1,4\rangle,\langle 5,2\rangle,\langle 10,2\rangle,\langle 2,9\rangle,\langle 9,8\rangle,\langle 6,5\rangle,\langle 3,4\rangle,\langle 10,8\rangle,\langle 8,4\rangle,\langle 9,1\rangle,\langle 7,9\rangle,\langle 3,5\rangle,\langle 3,1\rangle\}$.
   b) $G \leftarrow [\![V;A]\!]$ con $A \leftarrow \{\langle 7,8\rangle,\langle 3,2\rangle,\langle 7,10\rangle,\langle 10,8\rangle,\langle 5,3\rangle,\langle 6,8\rangle,\langle 8,1\rangle,\langle 1,4\rangle,\langle 3,3\rangle,\langle 7,7\rangle,\langle 10,10\rangle,\langle 5,5\rangle,\langle 6,6\rangle,\langle 8,8\rangle,\langle 1,1\rangle,\langle 2,2\rangle,\langle 4,4\rangle,\langle 7,1\rangle,\langle 10,1\rangle,\langle 5,2\rangle,\langle 6,1\rangle,\langle 8,4\rangle,\langle 7,4\rangle,\langle 10,4\rangle,\langle 6,4\rangle\}$.
   c) $G \leftarrow [\![V;A]\!]$ con $A \leftarrow \{\langle 1,6\rangle,\langle 9,6\rangle,\langle 5,3\rangle,\langle 2,8\rangle,\langle 3,6\rangle,\langle 3,10\rangle,\langle 1,1\rangle,\langle 9,9\rangle,\langle 5,5\rangle,\langle 2,2\rangle,\langle 3,3\rangle,\langle 8,8\rangle,\langle 6,6\rangle,\langle 10,10\rangle,\langle 6,1\rangle,\langle 6,9\rangle,\langle 3,5\rangle,\langle 8,2\rangle,\langle 6,3\rangle,\langle 10,3\rangle,\langle 1,9\rangle,\langle 1,3\rangle,\langle 9,1\rangle,\langle 9,3\rangle,\langle 5,6\rangle,\langle 5,10\rangle,\langle 3,1\rangle,\langle 3,9\rangle,\langle 6,10\rangle,\langle 6,5\rangle,\langle 10,6\rangle,\langle 10,5\rangle,\langle 10,1\rangle,\langle 10,9\rangle,\langle 1,10\rangle,\langle 1,5\rangle,\langle 9,10\rangle,\langle 9,5\rangle,\langle 5,1\rangle,\langle 5,9\rangle\}$.
4. Escribe la matriz de pertenencia de la relación que definen las aristas del siguiente grafo:



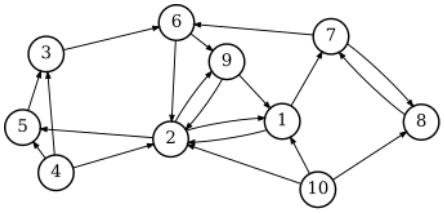
5. Considerando el grafo $G \leftarrow [\![V;A]\!]$ del ejercicio 4, contesta las siguientes preguntas:
   a) $\texttt{vGent}(4,G)$_____
   b) $\texttt{vGsal}(10,G)$_____
   c) $\texttt{vGr}(6,G)$_____
6. Describe una relación basada en la vida real, que se pueda modelar mediante un grafo de tipo rueda y otra basada en un grafo completo.
7. El siguiente grafo es bipartita. Demuestra que el grafo es bipartita proporcionando la bipartición $\{X,Y\}$ de los vértices.



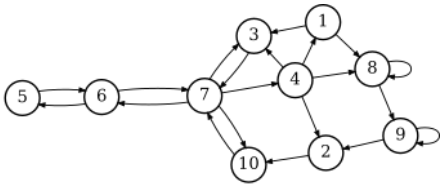
8. Escribe una función en `Python` llamada `esBipartita`, que recibe un grafo G como único argumento y que ofrece como resultado `True` si el grafo G es bipartita; de otro modo debe devolver `False`.

```
>>> H = genGrafo([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [[1, 9], [1, 10], [2, 8], [2, 7],
     [3, 10], [3, 9], [4, 10], [4, 7], [5, 9], [5, 7], [6, 8], [6, 10], [7, 6], [7,
     3], [7, 1], [8, 4], [8, 2], [8, 5], [9, 6], [9, 5], [9, 1], [10, 5], [10, 4],
     [10, 2]]])
>>> esBipartita(H)
True
>>>
```

9. Considera los siguientes grafos que se muestran en las siguientes figuras:
   $G \leftarrow [\![V;A]\!]$, con $G.A \leftarrow \{\langle 6,9\rangle,\langle 2,1\rangle,\langle 1,2\rangle,\langle 4,3\rangle,\langle 6,2\rangle,\langle 1,7\rangle,\langle 10,2\rangle,\langle 7,8\rangle,$
   $\langle 9,2\rangle,\langle 9,1\rangle,\langle 8,7\rangle,\langle 5,3\rangle,\langle 4,5\rangle,\langle 4,2\rangle,\langle 10,8\rangle,\langle 10,1\rangle,\langle 2,5\rangle,\langle 2,9\rangle,\langle 3,6\rangle,\langle 7,6\rangle\}$
   $H \leftarrow [\![V;A]\!]$, con $H.A \leftarrow \{\langle 8,8\rangle,\langle 7,6\rangle,\langle 4,3\rangle,\langle 8,9\rangle,\langle 1,8\rangle,\langle 7,4\rangle,\langle 4,1\rangle,\langle 7,3\rangle,$
   $\langle 1,3\rangle,\langle 5,6\rangle,\langle 6,5\rangle,\langle 2,10\rangle,\langle 7,10\rangle,\langle 9,9\rangle,\langle 9,2\rangle,\langle 4,8\rangle,\langle 4,2\rangle,\langle 10,7\rangle,\langle 3,7\rangle,\langle 6,7\rangle\}$



G

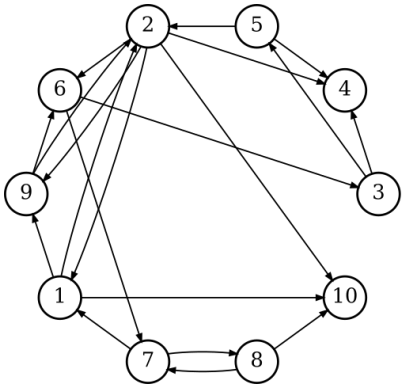


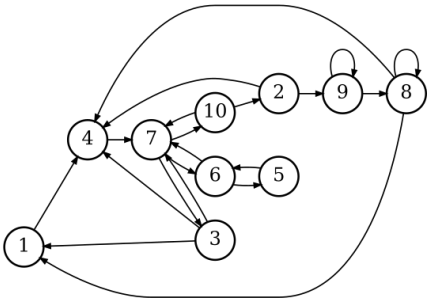
H

Realiza las siguientes actividades:

a) ¿Cuántas aristas tiene G?
b) ¿Cuántas aristas tiene H?
c) ¿Cuántas aristas tiene $G \cup H$?
d) ¿Cuántas aristas tiene G^3?
e) ¿Cuántas aristas tiene H^3?

10. Considera nuevamente los grafos G y H del ejercicio anterior. ¿Cuál de los siguientes grafos corresponde a G^{-1} y a H^{-1}?

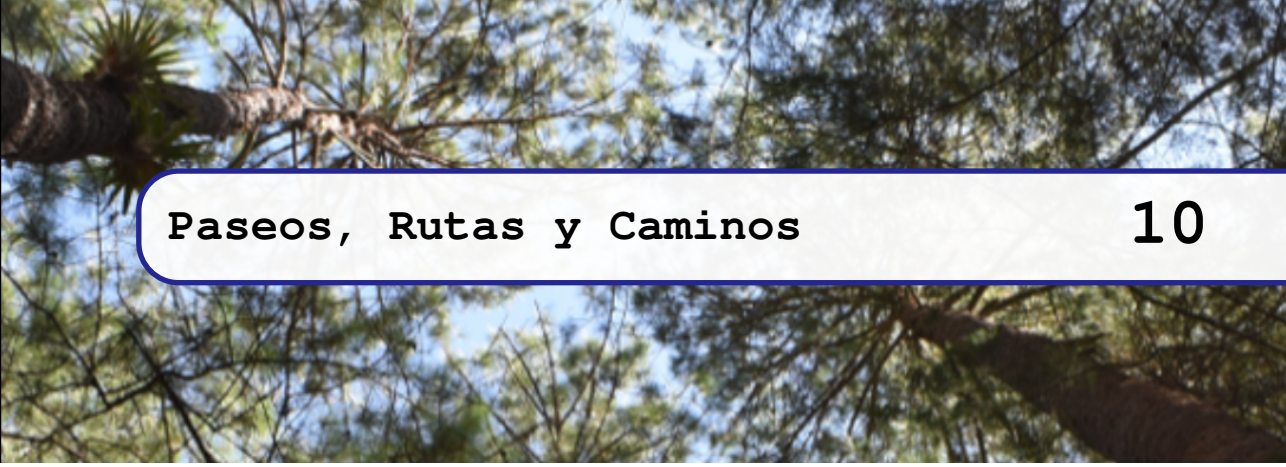




Describe cada grafo con:

a) Matriz de pertenencia

b) Listas de adyacencias



Paseos, Rutas y Caminos 10

10.1 Paseos

Imagina un paseo en donde recorres caminos y lugares. Los caminos, en el sentido coloquial, unen los lugares, de modo que en un paseo se inicia en un lugar, se avanza por un camino, se llega a otro lugar, y así hasta terminar el paseo.

La posibilidad de crear paseos en los grafos, ha permitido que éstos sean una de las herramientas de modelado más útiles en las ciencias computacionales. Tienen aplicaciones en la planificación de actividades [GNT04, p. 113] o de rutas que se pueden utilizar en robótica [LaV09, p. 507], procesos [Deo16], ciencias del conocimiento [CM09] entre muchas otras aplicaciones.

El siguiente grafo nos servirá para estudiar los conceptos de este capítulo, de modo que se repetirá la imagen, haciendo énfasis en diferentes aspectos.

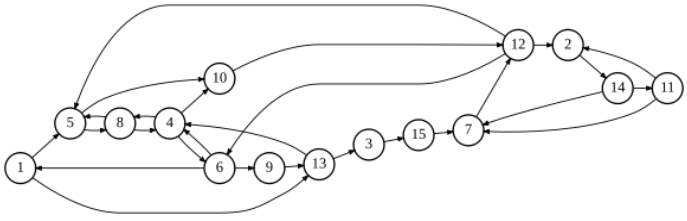


Figura 10.1: *Grafo* $G \hookleftarrow [\![V;A]\!]$ *utilizado en el estudio de paseos, rutas y caminos; con* $V \hookleftarrow \{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\}$ *y aristas* $A \hookleftarrow \{\langle 12,2\rangle,\langle 11,2\rangle,\langle 9,13\rangle,\langle 4,6\rangle,\langle 14,7\rangle,\langle 12,6\rangle,\langle 11,7\rangle,\langle 12,5\rangle,\langle 14,11\rangle,\langle 1,5\rangle,\langle 5,10\rangle,\langle 6,1\rangle,\langle 2,14\rangle,\langle 4,10\rangle,\langle 10,12\rangle,\langle 8,4\rangle,\langle 6,9\rangle,\langle 7,12\rangle,\langle 5,8\rangle,\langle 13,4\rangle,\langle 1,13\rangle,\langle 13,3\rangle,\langle 3,15\rangle,\langle 15,7\rangle,\langle 6,4\rangle,\langle 4,8\rangle,\langle 8,5\rangle\}$

10.1.1 Definición

Definición 10.1.1 -- Paseo. Si $G \leftarrow [\![V;A]\!]$ es un grafo, un paseo en G es una secuencia de longitud $\ell \geq 1$ de vértices alternados con aristas $\omega \leftarrow \langle v_1, a_1, v_2, \ldots, a_{\ell-1}, v_\ell \rangle$, cuyos términos son al menos un vértice y cero o más aristas. Cada arista a_j con $1 \leq j < \ell$ es de la forma $\langle v_{j-1}, v_j \rangle$ es colocada entre los vértices v_{j-1} y v_j, en ese orden.

La secuencia vacía $\langle\,\rangle$ no es un paseo. El paseo más pequeño tiene un vértice. En general, tampoco es un paseo una secuencia con términos $\ldots, v_{j-1}, \langle v_j, v_{j-1} \rangle, v_j, \ldots$, porque el orden de los vértices no concuerda con la dirección de la arista. El siguiente algoritmo sirve para verificar que una secuencia de elementos sea un paseo.

```
Algoritmo esPaseo(ω, G):
Requiere: ω una secuencia de vértices y aristas
          G↤[[V;A]] un grafo
Devuelve: booleano

1. Si ω=∅ ∨ esPar(|ω|),↦False. Los paseos no son vacíos y son de longitud impar
2. En otro caso:
3.     A_ω ↤ lista con las aristas en ω en orden de ocurrencia.
4.     p ↤ ∀a∈A_ω : a∈G.A. Todas las aristas en ω, pertenecen al grafo
5.     q ↤ ∀i∈[2,4,...,|ω|−1]:tIguales(ω_i,tupla(ω_{i−1},ω_{i+1}))
6.     ↦ p∧q
```

Algunas cosas importantes que observar en el algoritmo, es que en el paso 4 es suficiente verificar que todas las tuplas en ω son aristas en el grafo, esto es así porque al verificar las tuplas también se verifican los vértices que la componen. Luego, en un paso siguiente, se verifica que las aristas en ω sean iguales a las tuplas conformadas por los vértices que aparecen antes y después de la arista.

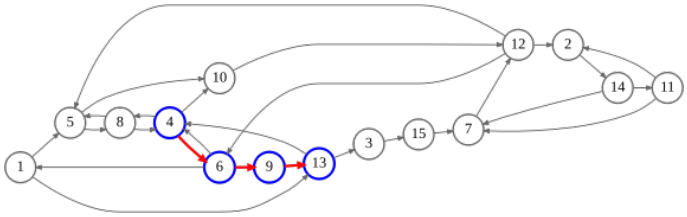


Figura 10.2: *Paseo* $\omega \leftarrow \langle 4, \langle 4,6 \rangle, 6, \langle 6,9 \rangle, 9, \langle 9,13 \rangle, 13 \rangle$.

Ejemplo 10.1

Determina si la secuencia $\omega \leftarrow \langle 4, \langle 4,6 \rangle, 6, \langle 6,9 \rangle, 9, \langle 9,13 \rangle, 13 \rangle$ es un paseo válido en el grafo G de la figura 10.1. Observa la figura 10.2 para ver el paseo de este ejemplo.

1. $(\omega = \varnothing \vee \texttt{esPar}(|\omega|)) \mapsto \texttt{False}$
2. Como ℓ no es vacío y es de longitud impar, sigue en el paso 3 del algoritmo.
3. $A_\omega \leftarrow \langle \langle 4,6 \rangle, \langle 6,9 \rangle, \langle 9,13 \rangle \rangle$
4. $p \leftarrow \forall a \in A_\omega : a \in G.A = \texttt{True}$.
5. $q \leftarrow (\forall i \in [2,4,6] : \omega_i = \langle \omega_{i-1}, \omega_{i+1} \rangle) = \texttt{True}$.
   $\omega \leftarrow \langle 4_{\omega_1}, \langle 4,6 \rangle_{\omega_2}, 6_{\omega_3}, \langle 6,9 \rangle_{\omega_4}, 9_{\omega_5}, \langle 9,13 \rangle_{\omega_6}, 13_{\omega_7} \rangle$
   a) $i \leftarrow 2$; $\omega_2 = \langle \omega_1, \omega_3 \rangle$; $(\langle 4,6 \rangle = \langle 4,6 \rangle) \mapsto \texttt{True}$
   b) $i \leftarrow 4$; $\omega_4 = \langle \omega_3, \omega_5 \rangle$; $(\langle 6,9 \rangle = \langle 6,9 \rangle) \mapsto \texttt{True}$
   c) $i \leftarrow 6$; $\omega_6 = \langle \omega_5, \omega_7 \rangle$; $(\langle 9,13 \rangle = \langle 9,13 \rangle) \mapsto \texttt{True}$
6. $\mapsto p \wedge q = \texttt{True}$.

10.1.2 Partes de un paseo

En cualquier paseo $\omega \leftarrow \langle v_1, a_1, v_2, \ldots, a_{\ell-1}, v_\ell \rangle$, con $\ell \geq 1$, el **origen** del paseo es el vértice v_1 y el **término** del paseo es v_ℓ, por lo que es útil tener herramientas computacionales para obtener el origen y término de un paseo. En las siguientes funciones, se supone que el parámetro `w` es una secuencia de vértices y aristas que ya tiene la forma de un paseo, así que simplemente se verifica que sea un paseo válido en el grafo y luego se devuelve el vértice adecuado. En caso que `w` no sea un paseo, se devuelve `False`. Todos los vértices que no sean ni el inicio del paseo ni el término, son **vértices internos**.

Código 10.1: *Origen y término de un paseo*

```
def wO(w:list, G:Grafo):
    """
    Devuelve el origen de un paseo.
    """
    return car(w) if esPaseo(w,G) else False

def wT(w:list, G:Grafo):
    """
    Devuelve el término de un paseo.
    """
    return w[-1] if esPaseo(w, G) else False
```

Ejemplo 10.2

Considera el grafo G de la figura 10.1 y utiliza las herramientas `wO` y `wT` para recuperar el origen y término de las siguientes secuencias.:

1. $\langle 4, \langle 4, 10 \rangle, 10, \langle 10, 12 \rangle, 12, \langle 12, 2 \rangle, 2 \rangle$
2. $\langle 4, \langle 4, 8 \rangle, 8, \langle 8, 5 \rangle, 5, \langle 5, 1 \rangle, 1 \rangle$

```
>>> wO([4, [4, 10], 10, [10, 12], 12, [12, 2], 2], G)
4
>>> wT([4, [4, 10], 10, [10, 12], 12, [12, 2], 2], G)
2
>>> wO([4, [4, 8], 8, [8, 5], 5, [5, 1], 1], G)
False  # No es un paseo válido
>>> wT([4, [4, 8], 8, [8, 5], 5, [5, 1], 1], G)
False  # No es un paseo válido
>>>
```

En el paseo $\langle 4, \langle 4, 10 \rangle, 10, \langle 10, 12 \rangle, 12, \langle 12, 2 \rangle, 2 \rangle$, los vértices 10 y 12 son internos.

Como la secuencia vacía no es un paseo, tampoco tiene inicio ni término, por supuesto tampoco tiene vértices internos. Por su parte, el paseo que tiene una sola arista, por ejemplo $\langle 4, \langle 4, 8 \rangle, 8 \rangle$ tiene su vértice inicial 4, su vértice término 8, pero no tiene vértices internos. Cuando en el grafo hay un autociclo, un paseo que sigue este autociclo tiene vértices inicial y término iguales; tienen la forma $\langle v, \langle v, v \rangle, v \rangle$, con $v \in G.V$ y $\langle v, v \rangle \in G.A$.

10.1.3 Paseo nulo

El **paseo nulo** en un grafo G es un solitón en $G.V$ [ver definición 6.3.2, página 125]. Un grafo G con $|G.V| = n$ tiene n paseos nulos. Un paseo nulo sirve de punto de referencia para iniciar un paseo. Al pensar en un verdadero paseo, el paseo nulo sería como estar «en casa», listo para iniciar el viaje. Como un paseo puede iniciar en cualquiera de los n vértices, hay n paseos nulos.

Para verificar que cualquier secuencia ω es un paseo nulo en un grafo G, es suficiente verificar que ω sea un paseo en el grafo G y sea solitón.

Código 10.2: *Verifica que un paseo sea nulo*

```
def esPaseoNulo(w:list, G)-> bool:
    """
    Verifica que w sea un paseo nulo en el grafo G.
    """
    return y(esPaseo(w, G), esSoliton(w))
```

Ejemplo 10.3

Considera el grafo de la figura 10.1:

- Las secuencias $\langle 1\rangle$,$\langle 13\rangle$,$\langle 2\rangle$ son tres diferentes paseos nulos.
- $\langle\ \rangle$. No es un paseo en el grafo G, ni es solitón.
- $\langle\langle 9,13\rangle\rangle$. Si es un solitón, pero no es un paseo en G.
- $\langle 9,13\rangle$. No es ni un paseo, ni es solitón.

10.1.4 Longitud de un paseo

La **longitud** de un paseo ω se denota $|\omega|$ y es la cantidad de aristas que tiene. La longitud de un paseo se puede obtener al observar que en cualquier paseo de n términos (vértices y aristas), con n un número positivo impar, hay $\lfloor \frac{n}{2} \rfloor$ aristas y $\lceil \frac{n}{2} \rceil$ vértices. Cualquier paseo nulo tiene longitud 0.

Código 10.3: *Longitud de un paseo*

```
def wLong(w:list)-> int:
    """
    Devuelve la longitud de un paseo.
    """
    return (card(w)-1) // 2
```

Cualquier secuencia ω es un **paseo unitario**, si w es un paseo en el grafo G y tiene longitud unitaria. Un paseo unitario tiene una sola arista, significa que ha habido un traslado desde el vértice inicial hasta el vértice término sin pasar por algún vértice intermedio.

Código 10.4: *Es paseo unitario*

```
def esPaseoUnit(w:list, G:Grafo)-> bool:
    """
    Determina si w es un paseo unitario en G.
    """
    return y(esPaseo(w,G), wLong(w) == 1)
```

Ejemplo 10.4

Utiliza la herramienta `esPaseoUnit` para determinar si las siguientes secuencias son paseos unitarios.

1. $\langle 8,\langle 8,7\rangle,7,\langle 7,2\rangle,2,\langle 2,9\rangle,9,\langle 9,14\rangle,14,\langle 14,3\rangle,3\rangle$
2. $\langle 13\rangle$
3. $\langle 15,\langle 15,7\rangle,7\rangle$

```
>>> esPaseoUnit([8, [8, 7], 7, [7, 2], 2, [2, 9], 9, [9, 14], 14, [14, 3], 3], G)
False  # No es un paseo ni es de longitud 1
>>> esPaseoUnit([13], G)
False # No es de longitud 1
>>> esPaseoUnit([15, [15, 7], 7], G)
True # Es un paseo de longitud 1
>>>
```

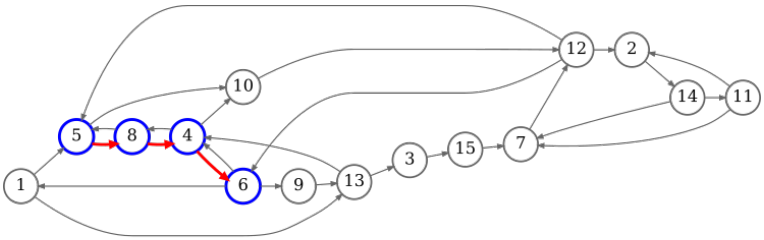
10.1.5 Paseo inverso

Sea $\omega \leftarrow\!\shortmid \langle v_1, a_1, v_2, \ldots, a_{\ell-1}, v_\ell \rangle$ un paseo. La secuencia $\omega^{-1} \leftarrow\!\shortmid \langle v_\ell, a_{\ell-1}^{-1}, \ldots, v_2, a_1^{-1}, v_1 \rangle$ es el **paseo inverso** de ω, si tal secuencia es un paseo en el mismo grafo G.

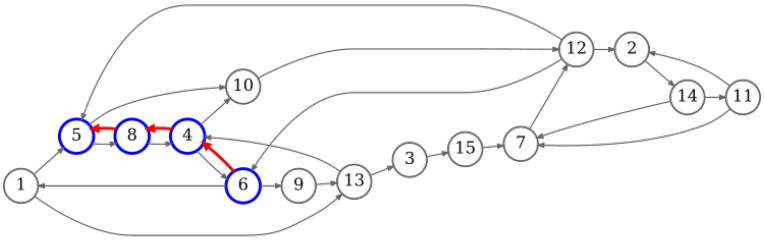
El paseo inverso ω^{-1} tiene como vértice inicial el vértice término de ω y como vértice término, el vértice inicial de ω. Los vértices internos de ω^{-1} son colocados en orden inverso de como aparecen en ω; las aristas del paseo inverso son las aristas inversas [ver la definición 6.4.5 en la página 131 y la definición 9.4.1 en la página 213] y están colocadas en el orden inverso en que aparecen en ω.

Código 10.5: *Paseo inverso*

```
1  def wInv(w:list, G:Grafo)-> list:
2      """
3      Devuelve el paseo inverso de w si es un paseo.
4      """
5      res = tInversa([tInversa(i) if esTupla(i) else i for i in w])
6      return res if esPaseo(res,G) else False
```



$\omega \leftarrow\!\shortmid \langle 5, \langle 5,8 \rangle, 8, \langle 8,4 \rangle, 4, \langle 4,6 \rangle, 6 \rangle$



$\omega^{-1} \leftarrow\!\shortmid \langle 6, \langle 6,4 \rangle, 4, \langle 4,8 \rangle, 8, \langle 8,5 \rangle, 5 \rangle$

Figura 10.3: *Paseo y paseo inverso.*

Ejemplo 10.5

Considerando el grafo G de la figura 10.1 encuentra el paseo inverso de los siguientes paseos:

1. $\langle 5, \langle 5,8 \rangle, 8, \langle 8,4 \rangle, 4, \langle 4,6 \rangle, 6 \rangle$
2. $\langle 1, \langle 1,5 \rangle, 5, \langle 5,10 \rangle, 10, \langle 10,12 \rangle, 12 \rangle$

```
>>> wInv([5, [5, 8], 8, [8, 4], 4, [4, 6], 6])
[6, [6, 4], 4, [4, 8], 8, [8, 5], 5] # Ver la figura 10.3
>>> wInv([1, [1, 5], 5, [5, 10], 10, [10, 12], 12])
False # El resultado no es un paseo en G
>>>
```

En un grafo no dirigido siempre es posible encontrar un paseo inverso, esto se debe a la característica de simetría de las aristas, sin embargo en los grafos dirigidos no siempre se tienen las condiciones para ofrecer el paseo inverso.

10.1.6 Concatenación de paseos

Definición 10.1.2 Sean $\omega \leftarrowtail \langle u_1, a_1, u_2, \ldots, a_{\ell-1}, u_\ell \rangle$ y $\kappa \leftarrowtail \langle v_1, e_1, v_2, \ldots, e_{m-1}, v_m \rangle$ dos paseos en el mismo grafo G con $\mathtt{wT}(\omega) = \mathtt{wO}(\kappa)$. La **concatenación** de ω con κ es denotada $\omega\kappa$ y el paseo $\langle u_1, a_1, u_2, \ldots, a_{\ell-1}, u_\ell, e_1, v_2, \ldots, e_{m-1}, v_m \rangle$ que inicia en $\mathtt{wO}(\omega)$ y termina en $\mathtt{wT}(\kappa)$.

Observa que la condición importante es que el vértice término de ω sea el mismo que el vértice inicial de κ, esto asegura que el resultado sea un paseo en ese grafo G, ya que ambas secuencias son paseos en G.

Si $\omega\kappa$ es posible, entonces $|\omega\kappa| = |\omega| + |\kappa|$. Esto es así porque en la concatenación de paseos, la cantidad de vértices utilizados son todos los que hay en ω mas todos los vértices de κ, menos uno, que es el vértice inicial de κ, pero la cantidad de aristas es la suma de la cantidad de aristas en cada paseo, y es justamente la cantidad de aristas la que proporciona la longitud del paseo.

Ejemplo 10.6

Construye la concatenación de $\omega \leftarrowtail \langle 2, \langle 2,14\rangle, 14, \langle 14,11\rangle, 11, \langle 11,7\rangle, 7, \langle 7,12\rangle, 12, \langle 12,6\rangle, 6\rangle$ con $\kappa \leftarrowtail \langle 6, \langle 6,9\rangle, 9, \langle 9,13\rangle, 13, \langle 13,3\rangle, 3\rangle$.

Como tanto ω como κ son paseos en G y además el vértice término de κ es el mismo que el vértice inicial de κ, es posible hacer la concatenación de los paseos, resultando

$$\omega\kappa \leftarrowtail \langle 2, \langle 2,14\rangle, 14, \langle 14,11\rangle, 11, \langle 11,7\rangle, 7, \langle 7,12\rangle, 12, \langle 12,6\rangle, 6, \langle 6,9\rangle, 9, \langle 9,13\rangle, 13, \langle 13,3\rangle, 3\rangle$$

10.2 Rutas

10.2.1 Concepto de ruta

Definición 10.2.1 Una secuencia de aristas diferentes $\langle a_1, \ldots, a_{\ell-1} \rangle$ de un grafo $G \leftarrowtail [\![V; A]\!]$ con $\ell \geq 1$ es una **ruta** en G, si la secuencia $\langle a_1.vi, a_1, a_1.vf, \ldots, a_{\ell-1}, a_{\ell-1}.vf \rangle$ es un paseo.

Un detalle importante en la definición 10.2.1, es la unicidad de las aristas en la secuencia. La interpretación dentro del grafo es que el paseo no debe recorrer una misma arista más de una vez. Este detalle debe ser considerado en la definición de un programa de cómputo que verifique si una secuencia de aristas es una ruta. Además, la otra condición importante es la conectividad entre el vértice final de una arista con el vértice inicial de la siguiente arista.

10.2.2 Verificación de una ruta válida

Podemos establecer entonces que una secuencia $\rho \leftarrowtail \langle a_1, \ldots, a_\ell \rangle$ es una ruta si cumple las siguientes condiciones:

1. No hay elementos repetidos en ρ. Un algoritmo sencillo para determinar si en una

secuencia de aristas ρ hay alguna repetida es el siguiente:

$$\texttt{sinRep}(\rho) \mapsto \begin{cases} \texttt{True} & \text{si } \rho = \langle\,\rangle \\ \texttt{False} & \text{si } \texttt{car}(\rho) \in \texttt{cdr}(\rho) \\ \texttt{sinRep}(\texttt{cdr}(\rho)) & \textit{eoc.} \end{cases}$$

2. La secuencia de vértices y aristas generada con ρ, debe ser un paseo en el grafo G.

Como cada arista $\langle u, v\rangle$ es un «puente» entre el vértice inicial u y el vértice final v de la arista, es posible deducir un paseo usando una secuencia de vértices.

Código 10.6: *Transforma una secuencia de aristas a tipo paseo*

```
def aristasApaseo(r:list)-> list:
    """
    Transforma una secuencia de aristas en una secuencia tipo paseo.
    """
    if esTvacia(r):
        return tupla()
    else:
        wr = [car(car(r))]
        for a in r:
            wr = tEscribe(wr, a)
            wr = tEscribe(wr, sPar(a))
    return wr
```

La idea general del programa es iniciar una tupla con el vértice inicial de la primera arista, luego iterativamente agregar la siguiente arista y luego el vértice final de la arista en curso, así hasta terminar con las aristas. Este procedimiento no garantiza que el producto final es un paseo válido, solamente se genera una secuencia que tiene la forma de un paseo.

Ejemplo 10.7

Siguiendo el programa 10.6, convierte la secuencia $\langle\langle 3,4\rangle,\langle 5,6\rangle,\langle 7,8\rangle,\langle 9,10\rangle\rangle$ a una secuencia con la forma de un paseo.

1. La secuencia resultante `wr` inicia con el primer elemento de la primera tupla, así $\texttt{wr} \leftarrow\!\shortmid \langle 3\rangle$.
2. para cada arista a en $\langle\langle 3,4\rangle,\langle 5,6\rangle,\langle 7,8\rangle,\langle 9,10\rangle\rangle$:
   - *a*) $a \leftarrow\!\shortmid \langle 3,4\rangle$
     1) $\texttt{wr} \leftarrow\!\shortmid \langle 3,\langle 3,4\rangle\rangle$
     2) $\texttt{wr} \leftarrow\!\shortmid \langle 3,\langle 3,4\rangle,4\rangle$.
   - *b*) $a \leftarrow\!\shortmid \langle 5,6\rangle$
     1) $\texttt{wr} \leftarrow\!\shortmid \langle 3,\langle 3,4\rangle,4,\langle 5,6\rangle\rangle$
     2) $\texttt{wr} \leftarrow\!\shortmid \langle 3,\langle 3,4\rangle,4,\langle 5,6\rangle,6\rangle$.
   - *c*) $a \leftarrow\!\shortmid \langle 7,8\rangle$
     1) $\texttt{wr} \leftarrow\!\shortmid \langle 3,\langle 3,4\rangle,4,\langle 5,6\rangle,6,\langle 7,8\rangle\rangle$
     2) $\texttt{wr} \leftarrow\!\shortmid \langle 3,\langle 3,4\rangle,4,\langle 5,6\rangle,6,\langle 7,8\rangle,8\rangle$.
   - *d*) $a \leftarrow\!\shortmid \langle 9,10\rangle$
     1) $\texttt{wr} \leftarrow\!\shortmid \langle 3,\langle 3,4\rangle,4,\langle 5,6\rangle,6,\langle 7,8\rangle,8,\langle 9,10\rangle\rangle$
     2) $\texttt{wr} \leftarrow\!\shortmid \langle 3,\langle 3,4\rangle,4,\langle 5,6\rangle,6,\langle 7,8\rangle,8,\langle 9,10\rangle,10\rangle$.

El resultado es $\langle 3,\langle 3,4\rangle,4,\langle 5,6\rangle,6,\langle 7,8\rangle,8,\langle 9,10\rangle,10\rangle$, que ya tiene la forma de un paseo. En un siguiente paso se debe verificar que sea un paseo legítimo en algún grafo [ver algoritmo 10.1.1, página 222].

```
>>> aristasApaseo([[3,4],[5,6],[7,8],[9,10]])
[3, [3, 4], 4, [5, 6], 6, [7, 8], 8, [9, 10], 10]
>>>
```

Código 10.7: *Verifica que una secuencia de aristas sea paseo*

```
def esRuta(r:list, G:Grafo)-> bool:
    """
    Verifica que una secuencia de aristas sea una ruta.
    """
    return y(sinRep(r), esPaseo(aristasApaseo(r), G))
```

Ejemplo 10.8

Verifica que las secuencias siguientes sean rutas válidas en el grafo G de la figura 10.1.

1. $\langle\langle 7,12\rangle,\langle 12,10\rangle,\langle 10,4\rangle,\langle 4,8\rangle\rangle$
2. $\langle\langle 7,12\rangle,\langle 12,5\rangle,\langle 5,8\rangle,\langle 8,4\rangle\rangle$

```
>>> esRuta([[7,12], [12,10], [10,4], [4,8]], G)
False
>>> esRuta([[7,12], [12,5], [5,8], [8,4]], G)
True
>>>
```

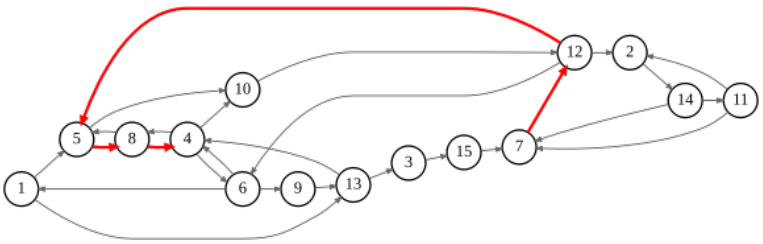


Figura 10.4: *Ruta* $\langle\langle 7,12\rangle,\langle 12,5\rangle,\langle 5,8\rangle,\langle 8,4\rangle\rangle$.

10.2.3 Operaciones con rutas

Si ρ es una ruta válida en un grafo $G = [\![V;A]\!]$, la **ruta inversa** de ρ se denota ρ^{-1} y es la ruta que contiene la secuencia inversa de aristas inversas de ρ. Así

$$\rho \leftarrow \langle a_1,\ldots,a_\ell\rangle; \quad \rho^{-1} \quad \begin{array}{l} \leftarrow \langle a_1^{-1},\ldots,a_\ell^{-1}\rangle^{-1} \\ \leftarrow \langle a_\ell^{-1},\ldots,a_1^{-1}\rangle \end{array}$$

En grafos dirigidos, la ruta inversa de una ruta no siempre existe, pero en los grafos no dirigidos, si hay una ruta válida, también hay una ruta inversa válida. Esto es así porque cada arista no dirigida $\{u,v\}$ se puede ver como dos aristas simétricas entre sí $\langle u,v\rangle,\langle v,u\rangle$, de modo que la arista inversa de una arista siempre existe.

Si $\rho \leftarrow \langle a_1,\ldots,a_\ell\rangle$ y $\tau \leftarrow \langle \mathtt{a}_1,\ldots,\mathtt{a}_m\rangle$ son dos rutas válidas en un grafo $G \leftarrow [\![V;A]\!]$, la **concatenación de las rutas** ρ y τ se escribe $\rho\tau$, y es la secuencia

$$\rho\tau \leftarrow \langle a_1,\ldots,a_\ell,\mathtt{a}_1,\ldots,\mathtt{a}_m\rangle$$

A pesar de que ρ y τ sean rutas en G, la secuencia obtenida no siempre puede generar un paseo en el grafo, por eso se requiere una verificación `esRuta` que permita decidir si la secuencia de aristas es una ruta o no.

Código 10.8: *Ruta inversa y concatenación de rutas*

```
def rutaInv(r:list, G:Grafo)-> list or bool:
    """
    Obtiene la ruta inversa de una ruta si existe, si no devuelve False.
    """
    ri = tInversa(enCada(lambda a:tInversa(a), r))
    return ri if esRuta(ri, G) else False

def rutaConcat(r:list,t:list, G:Grafo)-> list or bool:
    """
    Concatena dos rutas
    """
    tc = tConcat(r,t)
    return tc if esRuta(tc,G) else False
```

10.3 Caminos

10.3.1 Concepto de camino

Definición 10.3.1 Una secuencia no vacía de vértices diferentes $\langle v_1,\ldots,v_\ell\rangle$ en un grafo $G \hookleftarrow [\![V;A]\!]$ es un **camino**, si la secuencia $\langle v_1,\langle v_1,v_2\rangle,v_2,\ldots,\langle v_{\ell-1},v_\ell\rangle,v_\ell\rangle$ es un paseo. Un sinónimo de «camino» es «itinerario».

Un camino es un paseo en el que se han omitido las aristas que interconectan los vértices. Debido a que en un camino se tiene toda la información necesaria para obtener un paseo, no se requiere escribir vértices y tuplas. Utilizando caminos es una manera más sencilla de trabajar con paseos. En lo que resta del capítulo, trabajaremos con caminos en lugar de paseos o rutas, salvo en ocasiones en que sea necesario hacerlo.

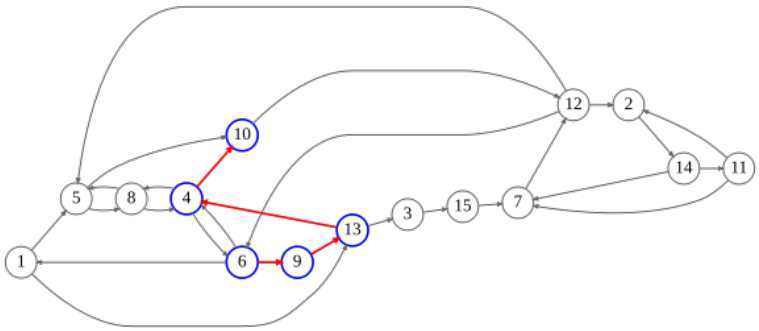


Figura 10.5: $\wp \hookleftarrow \langle 6,9,13,4,10\rangle$

Al trabajar con paseos en un grafo, se enfatizan los vértices; al trabajar con rutas se enfatizan las aristas. Los paseos tienen la información completa, mientras que las rutas y los caminos tienen información parcial, pero suficiente para que se pueda hacer una equivalencia.

Ejemplo 10.9

Un ministro de alguna religión, quiere visitar a sus feligreses en sus hogares. Como el ministro tiene conocimientos de matemáticas discretas, piensa en dibujar un grafo donde cada vértice significa el hogar de uno de los feligreses, y dibuja una arista desde un vértice u hasta otro vértice v, si hay una calle que le pueda llevar desde u hasta v.

En este ejemplo, se hace énfasis en el lugar en donde viven los feligreses, que es modelado por medio de un conjunto de vértices. Si se ha modelado un grafo, es mejor trabajar con caminos, en lugar de rutas o paseos.

El universo de los caminos

Por conveniencia en la notación, definiremos el siguiente conjunto.

Definición 10.3.2 Sea $G \leftarrow [\![V;A]\!]$ un grafo. El **universo de caminos** en G, lo denotamos $\mathscr{C}_G$, que es el conjunto de todas las secuencias no vacías de longitud $n \leq |G.V|$, formalmente

$$\mathscr{C}_G \leftarrow \{\langle v_1,\ldots,v_n\rangle \,|\, 1 \leq n \leq |G.V|\}$$

Cuando es claro y no hay confusión acerca del grafo al que se hace referencia, es posible omitir G de la notación.

Así $\mathscr{C}_G$ contiene todos los caminos que se pueden obtener en un grafo G. Un grafo G que tiene n vértices, tendrá al menos n elementos, cada uno de ellos es un camino nulo con origen en alguno de sus vértices. Claramente no puede haber un camino de longitud mayor que la cantidad de vértices, si lo hubiera, tendría al menos un vértice repetido, lo que es suficiente para que no sea un camino.

10.3.2 Verificación de un camino válido

Supongamos que $\langle v_1,\ldots,v_\ell\rangle$ es una secuencia no vacía de vértices tomados de un grafo $G \leftarrow [\![V;A]\!]$. Podemos asegurar que $\langle v_1,\ldots,v_\ell\rangle$ es un camino en G si al transformar $\wp$ a una secuencia del tipo paseo, se verifica que es un paseo.

Esto es cierto porque si un vértice v es adyacente a otro vértice u, significa que hay una arista $\langle u,v\rangle$, entonces si u es el origen y v el término, entonces se construye el paseo $\langle u,\langle u,v\rangle,v\rangle$.

Aunque el procedimiento descrito es correcto, no hay necesidad de hacerlo, pues se puede verificar una de las siguientes:

$$\texttt{esCamino}(\wp) \mapsto \begin{cases} \texttt{False} & \text{si } \wp = \langle\,\rangle \\ \texttt{True} & \text{si } \texttt{esSoliton}(\wp) \\ \texttt{sinRep}(\wp) \wedge \forall j \in \langle 2,\ldots,\ell\rangle : \texttt{esAdy}\left(v_j, v_{j-1}\right) & \text{si } \ell > 1. \end{cases}$$

Código 10.9: *Verificación camino válido*

```
1  def esCamino(cam:list, G:Grafo)-> bool:
2      """
3      Verifica que una lista de vértices sea un camino.
4      """
5      if esTvacia(cam):
6          return False
7      else:
8          p = sinRep(cam)
9          q = paraTodo(lambda i: esAdy(cam[i], cam[i-1], G), range(1,tLong(cam)))
10         return y(p,q)
```

Ejemplo 10.10

Verifica que las siguientes secuencias de vértices sean caminos en el grafo G de la figura 10.1 en la página 221.

1. $\langle\,\rangle$. No es un camino porque es una secuencia vacía.
2. $\langle 14\rangle$. Si es un camino, genera un paseo nulo.
3. $\langle 14,7,12,6,1\rangle$. Si es un camino.
4. $\langle 14,7,12,5,8,4,8,5\rangle$. No es un camino porque hay vértices repetidos.
5. $\langle 14,7,12,5,1\rangle$. No es un camino porque hay vértices no adyacentes.

```
>>> esCamino([],G)
False
>>> esCamino([14],G)
True
>>> esCamino([14,7,12,6,1],G)
True
>>> esCamino([14,7,12,5,8,4,8,5],G)
False
>>> esCamino([14,7,12,5,1],G)
False
>>>
```

10.3.3 Elementos de un camino

Origen y término de un camino

En un camino $\wp \in \mathscr{C}_G$ de la forma $\wp \leftarrowtail \langle v_1,\ldots,v_\ell\rangle$, el origen del camino lo denotaremos $\wp_O$ y es el vértice v_1 de la secuencia; el término del camino lo denotaremos $\wp_T$ y es el vértice v_ℓ de $\wp$. Observa que el símbolo $\wp$ es el identificador de la secuencia; otro ejemplo es la secuencia $\kappa \leftarrowtail \langle v_1 \ldots . v_f\rangle$, que tiene origen κ_O y término κ_T, y corresponden a los vértices v_1 y v_f de la secuencia κ respectivamente.

Código 10.10: *Origen y término de un camino*

```
1  def cO(cam:list, G:Grafo):
2      """
3      Devuelve el vértice inicial de un camino.
4      """
5      return car(cam) if esCamino(cam, G) else False
6
7  def cT(cam:list, G:Grafo):
8      """
9      Devuelve el vértice terminal de un camino.
10     """
11     return cam[-1] if esCamino(cam, G) else False
```

Todos los vértices, excepto el inicio y el término del camino, son llamados los vértices internos. Un camino que consta de un solo vértice no tiene vértices internos y el inicio es el mismo que el término del camino.

Longitud de un camino

Definición 10.3.3 La longitud de un camino $\wp \leftarrowtail \langle v_1,\ldots,v_l\rangle$ en un grafo G es la cantidad de vértices que hay en el resto de aristas de $\wp$, excepto el origen del camino. La longitud de $\wp$ se denota $|\wp|_G$ y es un número entero mayor que 0.

Convenientemente consideramos el caso en que la secuencia dada no sea un camino en el grafo en cuestión, tal fallo se responde con `False`. De modo que la longitud de un

camino puede ser o bien un número entero positivo, o `False`:

$$|\wp|_G \mapsto \begin{cases} \texttt{False} & \text{si } \wp = \langle\,\rangle, \\ \texttt{tLong}(\texttt{cdr}(\wp)) & \text{si } \texttt{esCamino}(\wp). \end{cases}$$

Código 10.11: *Calcula la longitud de un camino válido*

```
def cLong(cam:list, G:Grafo)-> int or bool:
    """
    Devuelve la longitud de un camino.
    """
    return tLong(cdr(cam)) if esCamino(cam,G) else False
```

Ejemplo 10.11

Calcula la longitud de los siguientes caminos en el grafo G [ver la figura 10.1, en la página221]:

1. $\langle 14,7,12,6,1\,\rangle$. Como la secuencia si es un camino, la longitud es 4.
2. $\langle 14,7,12,5,8,4,8,5\rangle$. No es un camino, por lo que se devuelve `False`.

```
>>> cLong([14,7,12,6,1], G)
4
>>> cLong([14,7,12,5,8,4,8,5],G)
False
>>>
```

Podemos basarnos en este concepto para introducir una clase especial de caminos.

Definición 10.3.4 Una familia de caminos de longitud k que tienen como vértice origen v en un grafo G, se puede escribir como $\complement^{(k)}_{v\in G.V}$, que es definido formalmente:

$$\complement^{(k)}_{v\in G.V} \leftarrow \{\wp \in \mathscr{C}_G \,|\, \wp_O = v \wedge |\wp| = k\}.$$

Cuando es obvio el grafo al cual se hace referencia, es posible omitirlo de la notación.

Tenemos entonces que

$$\complement^{(k)}_{v\in G.V} \subseteq \mathscr{C}_G.$$

10.3.4 Caminos especiales

Camino nulo

Definición 10.3.5 Una secuencia de vértices $\wp$ en un grafo G es un **camino nulo**, si $|\wp|_G = 0$.

Decimos que el camino es **nulo** porque existe únicamente el vértice que da origen al paseo, pero aún no hay rutas recorridas [aristas]. Si $v \in G.V$, el camino nulo tiene origen en v y es el camino $\langle v\,\rangle$. Observa que es un camino de longitud 0. En caso contrario, es decir cuando $v \notin G.V$, entonces podemos marcar un fallo con un valor `False`.

Para crear un camino nulo se requiere un vértice v de un grafo G, y la siguiente estrategia:

$$\texttt{cNulo}(v,G)\begin{cases} \langle v\,\rangle & \text{si } v \in G.V, \\ \texttt{False} & eoc \end{cases}$$

Código 10.12: *Crea un camino nulo*

```
def cNulo(v, G:Grafo)-> list:
    """
    Genera un camino nulo con el vértice v.
    """
    V = VInf(G.V)
    return tupla(v) if en(v, V) else False
```

Un grafo con $|G.V| = n$, es capaz de iniciar n caminos nulos, uno por cada vértice. Esto es $|\complement_v^{(0)}| = n$. Es imposible que un grafo no tenga caminos nulos, esto se debe a que el conjunto de vértices debe, por definición, no ser vacío. De modo que al menos tendrá un camino nulo.

Caminos unitarios

Definición 10.3.6 -- Camino unitario. Un camino $\wp$ es unitario en un grafo G, si $|\wp|_G = 1$.

Código 10.13: *Verifica camino unitario*

```
def esCunit(p:list,G:Grafo)-> bool:
    """
    Determina si un camino es unitario.
    """
    return cLong(p,G) == 1
```

Observa que esta definición no verifica que `p` sea un camino válido en `G`. Esta verificación se hace al calcular la longitud de la secuencia, que será un número positivo si resulta ser un camino, o bien `False` si no lo es. Consecuentemente, si no fue posible calcular la longitud, el resultado es `False`.

Cuando se tiene un camino nulo $\langle u \rangle$ en $G.V$, es posible obtener la familia de caminos unitarios con origen en u. Esto se logra al escribir por la derecha de $\langle u \rangle$ cada uno de sus vértices adyacentes.

$$\complement_v^{(1)} \leftarrow \{\texttt{tEscribe}(\langle u \rangle, v)\};\ \text{con}\ v \in \texttt{vAdy}(u, G)$$

```
def csUnit(v, G:Grafo)-> list:
    """
    Obtiene todos los caminos unitarios con origen en un vértice dado.
    """
    co = cNulo(u,G)
    return enCada(lambda v:tEscribe(co, v), vAdy(u, G))
```

Ejemplo 10.12

El conjunto de caminos unitarios que tienen como vértice inicial 12 en el grafo G de la figura 10.1, se obtiene al anexar los vértices adyacentes al paseo nulo $\langle 12 \rangle$, obteniendo lo que se muestra en la figura 10.6.

```
>>> csUnit(12,G)
[[12, 2], [12, 6], [12, 5]]
>>>
```

Así, la cantidad de caminos unitarios en un grafo coincide con la suma de los caminos unitarios de cada vértice, y también coincide con la cantidad de aristas.

$$\complement_G^{(1)} = \sum_{v \in G.V} \complement_v^{(1)} = |G.A|$$

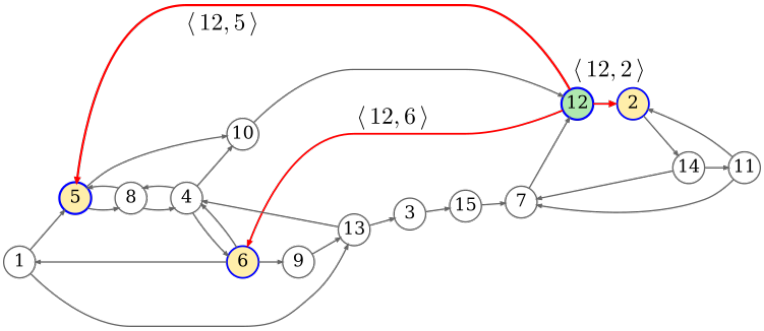


Figura 10.6: *Caminos unitarios con origen en 12:* $\{\langle 12,5\rangle, \langle 12,6\rangle, \langle 12,2\rangle\}$.

10.3.5 Operaciones con caminos

Extender un camino

Decimos que un camino es 1-extendido, cuando se agrega un vértice adyacente al vértice terminal de un camino, generando una secuencia que sigue siendo un camino. No siempre es posible obtener un camino 1-extendido por dos razones:

1. El vértice terminal no tiene vértices adyacentes.
2. El vértice adyacente al término del camino, ya pertenece al camino.

Definición 10.3.7 La función $\otimes^1_G : \mathscr{C}_G \to \mathcal{P}(\mathscr{C}_G)$, que llamaremos 1-extiende, permite extender un camino un solo vértice, generando así un conjunto de caminos:

$$\otimes^1_G(\wp) \mapsto \{\kappa \in \mathscr{C}_G \,|\, \kappa = \texttt{tEscribe}(\wp, v)\};\ \text{con}\ v \in \texttt{vAdy}(v, G) \setminus \wp.$$

La familia de caminos 1-extendidos de $\wp$, es el conjunto denotado por $\complement^{(k+1)}_{\wp_O}$, definido formalmente como:

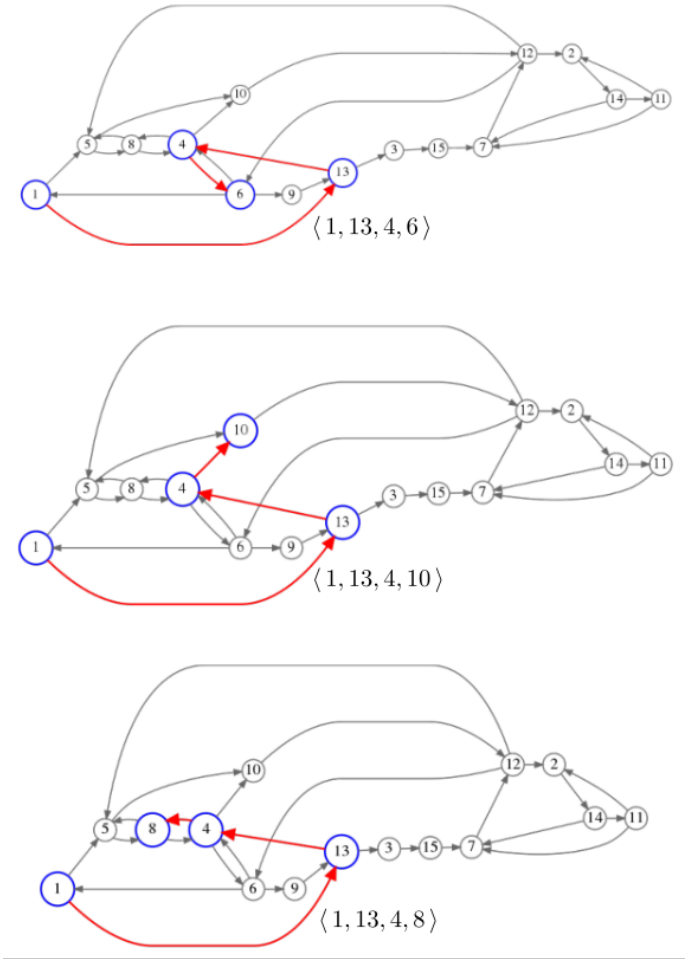
$$\complement^{(k+1)}_{\wp_O} \leftarrowtail \otimes^1_G(\wp).$$

El procedimiento para construir el conjunto generado por $\otimes^1_G(\wp)$ a partir de un camino $\wp$ requiere:

1. Obtener el vértice termino de $\wp$, digamos que $u \leftarrowtail \wp_T$.
2. Calcular el conjunto de vértices adyacentes a u del paso anterior, pero como una de las condiciones es la unicidad de los vértices en el camino, debemos considerar solamente aquellos vértices adyacentes que no se hayan considerado en el camino, por lo que los vértices a considerar son $va \leftarrowtail \texttt{vAdy}(u, G) \setminus \wp$
3. Por cada vértice en va, escribir por la derecha de $\wp$ uno de los vértices, esto produce los nuevos caminos en G.
4. Devolver la familia con los caminos generados.

Código 10.14: *Extiende un vértice cualquier camino*

```
1  def clext(c:list, G:Grafo)-> list:
2      """
3      Extiende un vértice un camino dado.
4      """
5      va = difc(vAdy(cT(c,G), G), c)
6      return enCada(lambda v:tEscribe(c, v), va)
```



$$\mathbb{C}^{(2+1)}_{\langle 1,13,4\rangle_T} = \mathbb{C}^{(3)}_4 \mapsto \{\langle 1,13,4,6\rangle, \langle 1,13,4,10\rangle, \langle 1,13,4,8\rangle\}$$

Figura 10.7: *Familia de caminos generados al extender un paso el camino* $\langle 1,13,4\rangle$

Ejemplo 10.13

Considera el camino $\wp \leftarrow\!\shortmid \langle 1,13,4\rangle$ del grafo G en la figura 10.1. Calcular los caminos 1-extendidos de $\wp$. El resultado se muestra en la figura 10.7

1. Obtenemos el término del camino $\wp_T \mapsto 4$.
2. Calculamos los nuevos vértices, que son aquellos que no han sido considerados antes en el camino.
$$\begin{aligned} \texttt{vAdy}(\wp_T, G)\backslash\wp &= \{6,10,8\}\backslash\{1,13,4\} \\ &\mapsto \{6,10,8\}. \end{aligned}$$
3. Construimos la nueva familia de conjunto, agregando un camino nuevo escribiendo por la derecha de $\wp$, cada vértice adyacente no considerado antes $\{6,10,8\}$.
   - $\langle 1,13,4,6\rangle$
   - $\langle 1,13,4,10\rangle$
   - $\langle 1,13,4,8\rangle$

4. Devolvemos el conjunto con los nuevos caminos. $\mapsto \{\langle 1,13,4,6\rangle, \langle 1,13,4,10\rangle, \langle 1,13,4,8\rangle\}$.

Utilizando la herramienta `c1ext` se obtiene la siguiente interacción.

```
>>> c1ext([1,13,4], G)
[[1, 13, 4, 6], [1, 13, 4, 10], [1, 13, 4, 8]]
>>>
```

Observa que las secuencias generadas son caminos, porque los vértices agregados no han sido considerados previamente, esto lo garantiza la diferencia de conjuntos $\texttt{vAdy}(\wp_T, G) \setminus \wp$.

Extender una familia de caminos

La idea de extender un camino se puede ampliar con el propósito de extender toda una familia de caminos $\complement_v^{(k)}$ un solo vértice. Definimos entonces la función extiende caminos del siguiente modo:

Definición 10.3.8 Llamaremos **extiende-caminos** a la función con dominio y codominio en las familias de caminos de un grafo G, definida:

$$\otimes_G^{*1} : \mathcal{P}(\mathscr{C}_G) \to \mathcal{P}(\mathscr{C}_G),$$

$$\otimes_G^{*1}\left(\complement_v^{(k)}\right) \mapsto \bigcup_{\wp \in \complement_v^{(k)}} \otimes_G^1(\wp)$$

El procedimiento se realiza extendiendo un vértice cada uno de los caminos en la familia dada, luego unir todas las familias, de lo que se obtiene una nueva familia de caminos, pero ahora todos los caminos de longitud incrementada en 1.

Ejemplo 10.14

En el ejemplo 10.13 se extendió el camino $\langle 1,13,4\rangle$ para obtener la familia

$$\complement_1^{(3)} \leftarrowtail \{\langle 1,13,4,6\rangle, \langle 1,13,4,10\rangle, \langle 1,13,4,8\rangle\}.$$

Extendamos todos los caminos en esta familia un solo vértice:

$$\otimes_G^{*1}\left(\complement_1^{(3)}\right) \mapsto \bigcup_{\wp \in \complement_1^{(3)}} \otimes_G^1(\wp)$$

1. Extendamos cada camino $\wp \in \complement_1^{(3)}$ un solo vértice:
   a) Extendiendo $\wp \leftarrowtail \langle 1,13,4,6\rangle$ se obtiene $\otimes_G^1(\wp) \mapsto \{\langle 1,13,4,6,9\rangle\}$. A partir del vértice 6 es posible alcanzar el vértice 4, pero la secuencia de vértices $\langle 1,13,4,6,4\rangle$ no es un camino porque 4 ya ha sido considerado.
   b) Extendiendo $\wp \leftarrowtail \langle 1,13,4,10\rangle$ se obtiene $\otimes_G^1(\wp) \mapsto \{\langle 1,13,4,10,12\rangle\}$.
   c) Extendiendo $\wp \leftarrowtail \langle 1,13,4,8\rangle$ se obtiene $\otimes_G^1(\wp) \mapsto \{\langle 1,13,4,8,5\rangle\}$.
2. Unir todos los resultados

$$\begin{aligned}\otimes_G^{*1}\left(\complement_1^{(3)}\right) &= \bigcup_{\wp \in \complement_1^{(3)}} \otimes_G^1(\wp)\\ &= \{\langle 1,13,4,6,9\rangle\} \cup \{\langle 1,13,4,10,12\rangle\} \cup \{\langle 1,13,4,8,5\rangle\}\\ &\mapsto \{\langle 1,13,4,6,9\rangle, \langle 1,13,4,10,12\rangle, \langle 1,13,4,8,5\rangle\}\end{aligned}$$

La nueva familia es $\otimes_G^{*1}\left(\complement_1^{(3)}\right) \mapsto \{\langle 1,13,4,6,9\rangle, \langle 1,13,4,10,12\rangle, \langle 1,13,4,8,5\rangle\} = \complement_1^{(4)}$.

```
def cs1ext(FC:list, G:Grafo)-> list:
    """
    Extiende una familia de caminos.
    """
    NF = enCada(lambda C: c1ext(C,G), FC)
    return Union(*NF)
```

Ejemplo 10.15

Utiliza la herramienta `cs1ext` para extender los caminos en la familia de caminos del ejemplo 10.13:

```
>>> p = [1,13,4]
>>> c1ext(p, G)
[[1, 13, 4, 6], [1, 13, 4, 10], [1, 13, 4, 8]]
>>> cs1ext(c1ext(p, G), G)
[[1, 13, 4, 8, 5], [1, 13, 4, 6, 9], [1, 13, 4, 10, 12]]
>>>
```

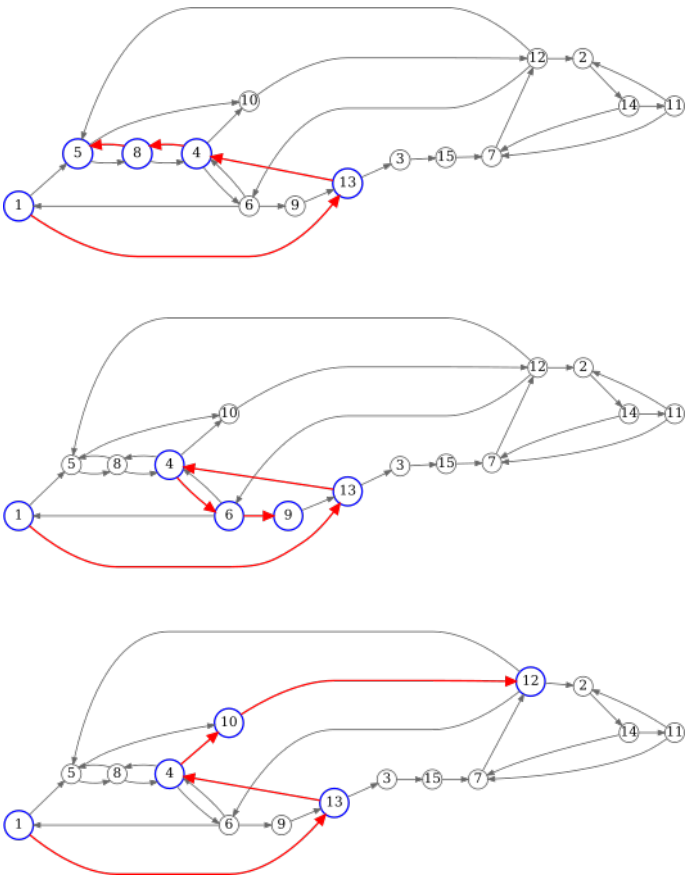


Figura 10.8: *Familia de caminos generados al extender la familia* $\mathcal{C}_1^{(3)}$.

k-caminos

Los k-caminos a partir de un vértice v de un grafo G, son caminos de longitud k que se pueden generar al aplicar repetidas veces las funciones anteriores. Definimos una función $\otimes_G^{*k} : G.V \times \mathbb{Z}^+ \times \mathcal{P}(\mathscr{C}_G) \to \mathcal{P}(\mathscr{C}_G)$

$$\otimes_G^{*k}(v,k,C \leftarrow \{\langle v \rangle\}) \mapsto \begin{cases} C & \text{si } k = 0 \\ \otimes_G^{*k}\left(v,k-1,C \leftarrow \otimes_G^{1}(C)\right) & eoc. \end{cases}$$

La función $\otimes_G^{*k}$ requiere:

1. Un vértice $v \in G.V$ que servirá como origen de los caminos.
2. Un número entero positivo k, que tendrá la longitud esperada de todos los caminos encontrados.
3. De manera opcional, una familia de caminos $C \in \mathcal{P}(\mathscr{C}_G)$ con valor inicial $\{\langle v \rangle\}$, que es la familia que contiene solo el camino nulo con origen en v.

Código 10.15: *Todos los caminos de longitud k a partir de un vértice*

```
def kCaminos(k:int, v, G:Grafo, C:list = None)-> list:
    """
    Calcula todos los caminos de longitud k desde un vértice v.
    """
    if C is None: C = conj(cNulo(v,G))
    if k == 0:
        return C
    else:
        return kCaminos(k-1, v, G, C=cs1ext(C, G))
```

Ejemplo 10.16

Calcula todos los caminos de longitud 3 a partir del vértice 11 del grafo G de la figura 10.1.

Debemos calcular $\otimes_G^{*k}(11,3)$ [Con $C \leftarrow \{\langle 11 \rangle\}$, [que es su valor predefinido]. Como $k \leftarrow 3$ es diferente que 0:

1. Con $k \leftarrow 3$:
   a) Con $C \leftarrow \{\langle 11 \rangle\}$, el nuevo valor de C es $\otimes_G^{*1}(C) \mapsto \{\langle 11,2 \rangle, \langle 11,7 \rangle\}$, ahora
   b) Hacemos $\otimes_G^{*k}(11,2,\{\langle 11,2 \rangle, \langle 11,7 \rangle\})$
2. Con $k \leftarrow 2$:
   a) Con $C \leftarrow \{\langle 11,2 \rangle, \langle 11,7 \rangle\}$, el nuevo valor de C es $\otimes_G^{*1}(C) \mapsto \{\langle 11,2,14 \rangle, \langle 11,7,12 \rangle\}$, ahora
   b) Hacemos $\otimes_G^{*k}(11,1,\{\langle 11,2,14 \rangle, \langle 11,7,12 \rangle\})$
3. Con $k \leftarrow 1$:
   a) Con $C \leftarrow \{\langle 11,2,14 \rangle, \langle 11,7,12 \rangle\}$, el nuevo valor de C es $\otimes_G^{*1}(C) \mapsto \{\langle 11,2,14,7 \rangle, \langle 11,7,12,2 \rangle, \langle 11,7,12,6 \rangle, \langle 11,7,12,5 \rangle\}$, ahora
   b) Hacemos $\otimes_G^{*k}(11,0,\{\langle 11,2,14,7 \rangle, \langle 11,7,12,2 \rangle, \langle 11,7,12,6 \rangle, \langle 11,7,12,5 \rangle\})$
4. Como ya $k = 0$ se devuelve el valor actual de C:
   a) $\mapsto \{\langle 11,2,14,7 \rangle, \langle 11,7,12,2 \rangle, \langle 11,7,12,6 \rangle, \langle 11,7,12,5 \rangle\}$

Usando la herramienta `kCaminos` del código 10.15, podemos obtener el mismo resultado. Una visión gráfica se muestra en la figura 10.9.

```
>>> kCaminos(11,3,G)
[[11, 2, 14, 7], [11, 7, 12, 2], [11, 7, 12, 6], [11, 7, 12, 5]]
>>>
```

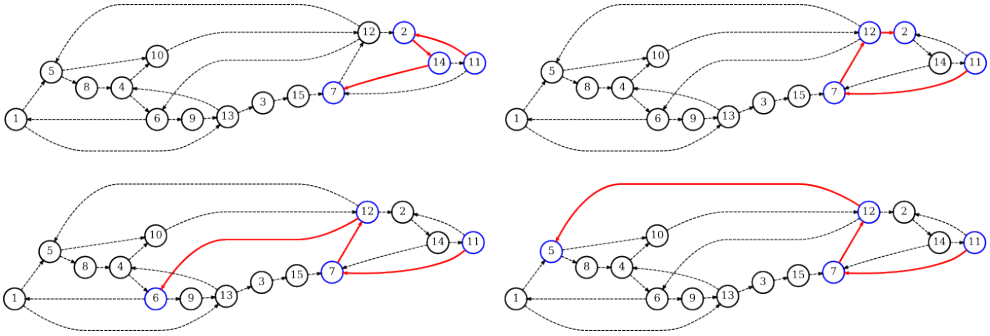


Figura 10.9: *Caminos de longitud 3 a partir de vértice 11 de G.*

10.3.6 Caminos hamiltonianos

Definición 10.3.9 Dado un grafo $H \hookleftarrow [\![V;A]\!]$ con $|H.V| \mapsto n$, una secuencia de vértices $\wp \hookleftarrow \langle v_1,\ldots,v_n\rangle$ es un **camino hamiltoniano** si $\wp$ es un camino y $|\wp| = n-1$.

Como la longitud de un camino hamiltoniano es $n-1$, se deben recorrer $n-1$ aristas, tocando n vértices diferentes, se concluye entonces que un camino hamiltoniano recorre todos sus vértices.

Consideremos para los siguientes ejemplos, este nuevo grafo que llamaremos H, definido con los vértices en $\{1,2,3,4,5,6,7\}$ y aristas $\{\langle 1,2\rangle,\langle 1,3\rangle,\langle 1,8\rangle,\langle 2,1\rangle,\langle 2,3\rangle,\langle 3,4\rangle,\langle 3,5\rangle,\langle 3,6\rangle,\langle 4,1\rangle,\langle 4,3\rangle,\langle 4,5\rangle,\langle 5,1\rangle,\langle 5,7\rangle,\langle 5,8\rangle,\langle 6,2\rangle,\langle 6,5\rangle,\langle 6,7\rangle,\langle 7,2\rangle,\langle 7,6\rangle,\langle 7,8\rangle,\langle 8,4\rangle,\langle 8,7\rangle\}$

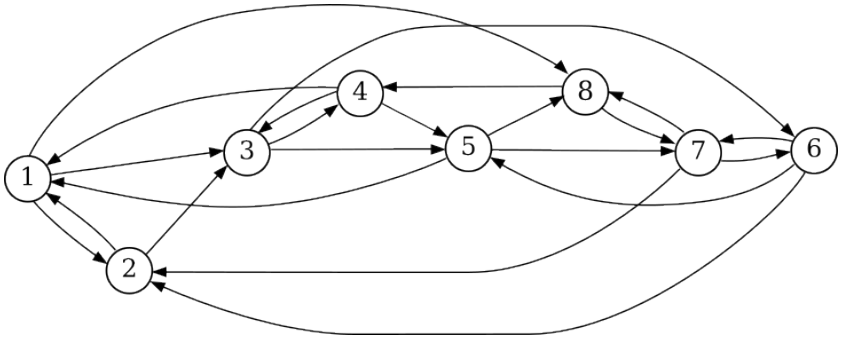


Figura 10.10: *Grafo $H \hookleftarrow [\![V;A]\!]$.*

Ejemplo 10.17

El camino $\langle 1,8,4,5,7,6,2,3\rangle$ es hamiltoniano en el grafo H de la figura 10.10. Otros caminos hamiltonianos son [no son todos]:

1. $\langle 4,3,6,5,8,7,2,1\rangle$
2. $\langle 7,8,4,1,2,3,6,5\rangle$

Se puede obtener un grafo H que tenga un único camino hamiltoniano tocando todas sus aristas si se construye como una lista ligada. Si el grafo tiene esta forma, todos sus

vértices tienen grado de entrada 1, excepto v_O que tiene grado de entrada 0; y todos sus vértices tienen grado de salida 1, excepto v_T que tiene grado de salida 0.

Para encontrar los caminos hamiltonianos que inician en un vértice, podemos utilizar el programa `kCaminos` del página 238, configurando la entrada para indicar la longitud adecuada `k` $\leftarrow\!\!\!\!\mapsto |V|-1$.

10.4 Ciclos

10.4.1 Concepto general

Definición 10.4.1 Una secuencia $\langle v_1 \cdots v_\ell, v_{\ell+1} \rangle$ de vértices en un grafo G es un **ciclo**, si cumple:

1. $\langle v_1, \ldots, v_\ell \rangle$ es un camino en G.
2. $v_{\ell+1} \in \texttt{vAdy}(v_\ell, G)$
3. $v_1 = v_{\ell+1}$.

La condición 1 establece que la secuencia de vértices propuesta, exceptuando el último vértice, debe ser un camino en el grafo G; la segunda condición exige que el último vértice sea adyacente al penúltimo de la secuencia, esto significa que existe una arista dirigida que salga de v_ℓ y llegue a $v_{\ell+1}$; finalmente la condición 3 indica que el último vértice de la secuencia debe ser el primero, lo que permite cerrar el camino.

Observa que si la secuencia $\langle v_1, \ldots, v_\ell \rangle$ no es un camino, no hay necesidad de verificar las otras dos condiciones. Verificar en primer lugar si la secuencia $\langle v_1, \ldots, v_\ell \rangle$ es o no un camino, ayuda a ahorrar tiempo de cálculo.

Código 10.16: *Verifica si una secuencia de vértices es un ciclo*

```
1  def esCiclo(c:list,G:Grafo)-> bool:
2      """
3      Determina si c es un ciclo.
4      """
5      d = c[:-1]
6      if esCamino(d,G):
7          return y(en(c[-1], vAdy(cT(d,G),G)), cO(d,G)==c[-1])
8      else:
9          return False
```

Ejemplo 10.18

Considera nuevamente el grafo H de la figura 10.10 que se muestra en la página 239.

Los siguientes caminos son ciclos en H:

1. $\langle 1,2,1 \rangle$
2. $\langle 8,4,3,5,7,8 \rangle$

```
>>> esCiclo([1,2,1], H)
True
>>> esCiclo([8,4,3,5,7,8], H)
True
>>>
```

10.4.2 Longitud de un ciclo

Definición 10.4.2 La longitud del ciclo es el número de aristas que están involucradas. Si $\varphi \leftarrow\!\!\!\!\mapsto \langle v_1, \ldots, v_\ell, v_1 \rangle$ es un ciclo, la longitud de φ se denota $|\varphi|$ y es $|\langle v_1, \ldots, v_\ell \rangle| + 1$, que es la longitud del camino que es la base del ciclo, incrementado en una unidad.

Ejemplo 10.19

Considera los siguientes ejemplos de ciclos tomados del grafo de la figura 10.1 en la página 221:

1. $\langle 1,5,8,4,6,1\rangle$ tiene longitud 5.
2. $\langle 9,13,3,15,7,12,5,8,4,6,9\rangle$ tiene longitud 10.
3. $\langle 8,4,8\rangle$ tiene longitud 2.
4. El grafo no tiene ciclos de longitud 1.

Calcular la longitud de un ciclo es sencillo una vez que se tiene la certeza de que la secuencia de vértices es un ciclo en el grafo. Cuando esto ocurre simplemente hay que devolver la cantidad de vértices que hay en la secuencia, exceptuando el vértice que se repite, por comodidad contamos los elementos del resto de la secuencia [todos excepto el primero].

```
1 def cicloLong(c:list, G:Grafo)-> int:
2     """
3     Devuelve la longitud de un ciclo.
4     """
5     return len(cdr(c)) if esCiclo(c, G) else False
```

Ejemplo 10.20

Tomando en cuenta el grafo H [página 239], vamos a obtener la longitud de los ciclos:

1. $\langle 4,1,3,4\rangle$. $|\langle 4,1,3,4\rangle| \mapsto 3$.
2. $\langle 2,1,8,4,3,6,2\rangle$. $|\langle 2,1,8,4,3,6,2\rangle| \mapsto 6$.

```
>>> cicloLong([4,1,3,4], H)
3
cicloLong([2,1,8,4,3,6,2], H)
6
>>>
```

10.4.3 Corrimientos en un ciclo

Por su naturaleza, un ciclo puede iniciar en cualquier vértice del ciclo para terminar en el vértice origen. Si $\varphi \leftarrowtail \langle v_1,\ldots,v_\ell,v_1\rangle$ es un ciclo, un **corrimiento simple** [de un solo paso] se logra haciendo que el ciclo inicie en el segundo vértice de φ, esto es:

$$\begin{aligned}\texttt{corr}(\varphi,G) &= \texttt{tEscribe}(\texttt{cdr}(\varphi),\texttt{car}(\texttt{cdr}(c)))\\ &= \texttt{tEscribe}(\langle v_2,\ldots,v_\ell,v_1\rangle,\texttt{car}(\texttt{cdr}(c)))\\ &= \texttt{tEscribe}(\langle v_2,\ldots,v_\ell,v_1\rangle,\texttt{car}(\langle v_2,\ldots,v_\ell,v_1\rangle))\\ &= \texttt{tEscribe}(\langle v_2,\ldots,v_\ell,v_1\rangle,v_2)\\ &\mapsto \langle v_2,\ldots,v_\ell,v_1,v_2\rangle\end{aligned}$$

Ejemplo 10.21

Escribe el primer corrimiento de ciclo $\varphi \leftarrowtail \langle 9,13,3,15,7,12,5,8,4,6,9\rangle$.

$$\begin{aligned}\texttt{corr}(\varphi) &= \texttt{tEscribe}(\texttt{cdr}(\varphi),\texttt{car}(\texttt{cdr}(\varphi)))\\ &= \texttt{tEscribe}(\langle 13,3,15,7,12,5,8,4,6,9\rangle,\texttt{car}(\texttt{cdr}(\varphi)))\\ &= \texttt{tEscribe}(\langle 13,3,15,7,12,5,8,4,6,9\rangle,\texttt{car}(\langle 13,3,15,7,12,5,8,4,6,9\rangle))\\ &= \texttt{tEscribe}(\langle 13,3,15,7,12,5,8,4,6,9\rangle,13)\\ &\mapsto \langle 13,3,15,7,12,5,8,4,6,9,13\rangle\end{aligned}$$

Código 10.17: *Corrimiento simple de un ciclo*

```
def corr(c:list):
    """
    Hace un corrimiento de un ciclo.
    """
    return tEscribe(cdr(c), car(cdr(c)))
```

Ejemplo 10.22

El primer corrimiento de $\langle 9,13,3,15,7,12,5,8,4,6,9\rangle$.

```
>>> c = [9,13,3,15,7,12,5,8,4,6,9]
>>> corr(C)
[13, 3, 15, 7, 12, 5, 8, 4, 6, 9, 13]
>>>
```

Es posible hacer que un ciclo inicie en cualquiera de los vértices del ciclo, simplemente hay que repetir el corrimiento simple hasta que el vértice inicial sea el deseado.

Si el vértice que deseamos se encuentra en la posición k del ciclo, se deben hacer k corrimientos simples. Pero se puede ahorrar tiempo y comparaciones al dividir el ciclo en dos partes, que llamaremos parte inicial y parte final. Luego concatenando por la derecha de la parte final, la parte inicial. Veamos como.

Si $\varphi \leftarrow\!\shortmid \langle v_1, v_2, \ldots, v_k, v_{k+1}, \ldots, v_{\ell-1}, v_\ell, v_1\rangle$ es un ciclo, después de k corrimientos simples se tiene:

$$\begin{array}{rcl}
0 & : & \langle v_1, v_2, v_3, \ldots, v_k, v_{k+1}, \ldots, v_{\ell-1}, v_\ell, v_1\rangle \\
1 & : & \langle v_2, v_3, \ldots, v_k, v_{k+1}, \ldots, v_{\ell-1}, v_\ell, v_1, v_2\rangle \\
2 & : & \langle v_3, \ldots, v_k, v_{k+1}, \ldots, v_{\ell-1}, v_\ell, v_1, v_2, v_3\rangle \\
\vdots & & \vdots \\
k-1 & : & \langle v_k, v_{k+1}, \ldots, v_{\ell-1}, v_\ell, v_1, v_2, v_3, \ldots, v_k\rangle \\
k & : & \langle v_{k+1}, \ldots, v_{\ell-1}, v_\ell, v_1, v_2, v_3, \ldots, v_k, v_{k+1}\rangle
\end{array}$$

Observamos que se logra una secuencia equivalente a k corrimientos simples, manipulando la secuencia original. El siguiente algoritmo describe cómo se manipula la secuencia [ver también la figura 10.11]:

```
Algoritmo kcorr(k, φ):
Requiere: k≤|φ| un número entero positivo
          φ un ciclo
Devuelve: un ciclo.

1. α ←\!↩ ⟨v1,...,vk⟩
2. β ←\!↩ ⟨vk+1,...,vℓ,v1⟩
3. α' ←\!↩ ⟨v2,...,vk⟩ # El resto de α
4. β0 ←\!↩ vk+1 # El primero de β
5. b ←\!↩ tEscribe(α',β0) ↦ ⟨v2,...,vk,vk+1⟩
6. ↦ tConcat(β,b) ↦ ⟨vk+1,...,vℓ−1,vℓ,v1,v2,v3,...,vk,vk+1⟩
```

Para asegurar que $0 \le k < |\varphi|$, se toma $k \leftarrow\!\shortmid k \bmod |\varphi|$, si $k \ge |\varphi|$.

Código 10.18: *k-corrimiento de un ciclo*

```
def kcorr(k:int, c:list)-> list:
    """
    Hace k corrimientos del ciclo c.
    """
    k = k % tLong(c)
    alpha = c[:k]
    beta = c[k:]
    alphaprim = cdr(alpha)
    beta0 = car(beta)
    b = tEscribe(alphaprim, beta0)
    return tConcat(beta, b)
```

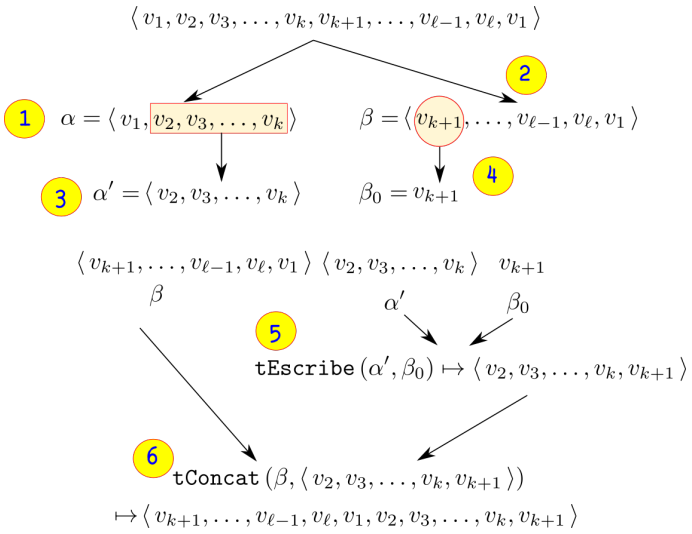


Figura 10.11: *Algoritmo para un k-corrimiento de un ciclo*

10.4.4 Igualdad en ciclos

Definición 10.4.3 Si ϕ y φ son dos ciclos, decimos que ϕ y φ son **ciclos iguales** y lo denotamos $\phi \stackrel{\circ}{=} \varphi$, si ambos ciclos recorren exactamente la misma secuencia de vértices.

Cuando los ciclos son de diferente longitud, es claro que los ciclos no son iguales, sin embargo, cuando los ciclos son de la misma longitud, pero no inician en el mismo vértice, es posible que aún puedan ser iguales.

Ejemplo 10.23

Considera el ciclo $\phi \leftarrow \langle 1,2,3,4,1 \rangle$, que representa el ciclo $1 \to 2 \to 3 \to 4 \to 1$; considera también el ciclo $\varphi \leftarrow \langle 3,4,1,2,3 \rangle$, que representa al ciclo $3 \to 4 \to 1 \to 2 \to 3$.

Al comparar los ciclos en forma de tuplas, evidentemente resulta que son diferentes:

$\phi \leftarrow \langle 1,2,3,4,1 \rangle$
$\varphi \leftarrow \langle 3,4,1,2,3 \rangle$

Sin embargo, al hacer un 2-corrimiento de φ, se obtiene:

$\phi \leftarrow \texttt{kcorr}(2,\phi)$;
$\phi \leftarrow \langle 3,4,1,2,3 \rangle$
$\varphi \leftarrow \langle 3,4,1,2,3 \rangle$.

Y ya se puede ver que $\phi \stackrel{\circ}{=} \varphi$.

Suponiendo que $|\phi| = |\varphi|$, podemos decir entonces que la siguiente expresión es `True`:

$$\phi \stackrel{\circ}{=} \varphi \leftrightarrow \exists k \in \{0,\ldots,|\phi|-1\} : \texttt{kcorr}(k,\phi) = \varphi$$

Observando que $\texttt{kcorr}(k,\phi) = \phi$ es la comparación entre tuplas [ver la sección 6.4.1, página 125].

Esto significa que para determinar que dos ciclos de la misma longitud son iguales, podemos hacer un corrimiento unitario [de un solo vértice], y probar si son tuplas iguales, en caso de que no lo sean, se hace un nuevo corrimiento unitario y se vuelve a

probar, esto se hace así hasta que se cumplan $|\phi|$ corrimientos y se determine que no son iguales, o bien sean iguales después de $0 \leq k < |\phi|$ corrimientos.

Código 10.19: *Tuplas iguales*

```
def ciclosIguales(c:list, d:list)-> bool:
    """
    Determina la igualdad entre dos ciclos.
    """
    if tLong(c) != tLong(d):
        return False
    else:
        i = 0
        flg = False
        while ox(i < tLong(c),flg):
            if tIguales(c,d):
                flg = True
            d = corr(d)
            i = i + 1
        return flg
```

```
>>> ciclosIguales([1,2,3,4,1], [3,4,1,2,3])
True
>>> ciclosIguales([1,2,3,5,1], [3,4,1,2,3])
False
>>> ciclosIguales([1,2,3,4,5,1], [3,4,1,2,3])
True
>>>
```

10.4.5 Ciclos de longitud k

Una vez que se tienen las herramientas necesarias para crear caminos [código 10.15, página 238] y escribir en una tupla [sección 6.8, página 129], podemos crear un procedimiento para generar todos los ciclos que inician en algún vértice en particular. Este procedimiento generará los ciclos de longitud 1, 2, en adelante, hasta incluir todos los vértices de un grafo G. El algoritmo es el siguiente

```
Algoritmo kCiclos(k, v, G):
Requiere: k≤|φ| un número entero positivo
          v: la información de un vértice
          G: un grafo
Devuelve: El conjunto de ciclos de longitud k que inician en v.

1. U ↤ {vértices u que tienen a v como adyacente.} [código 9.6, página 204]
2. C ↤ {Caminos de longitud k−1 que inician en v}. [código 10.15, página 238]
3. C_v ↤ {c ∈ C | c_T ∈ U} [código 10.10, página 231]
4. C_k ↤ enCada(λc•tEscribe(c,v),C_v) [código 5.1, página 106]
5. ↦ C_k
```

Código 10.20: *Todos los ciclos de longitud k del grafo G que inician en un vértice dado.*

```
def kCiclos(k, v, G):
    """
    Todos los ciclos de longitud k que inician en el vértice v del grafo G.
    """
    U = subc(lambda u: esAdy(v, u, G), VInf(G.V))
    C = kCaminos(k-1, v, G)
    Cv = subc(lambda c: en(cT(c, G), U), C)
    Ck = enCada(lambda c: tEscribe(c,v), Cv)
    return Ck
```

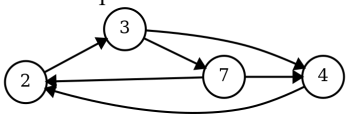
Un procedimiento inmediato es obtener todos los ciclos [de todos los tamaños] que inician en un vértice dado. Claramente el procedimiento involucra todas las longitudes de ciclos, desde 1 hasta el número de vértices $|G.V|$ inclusive.

Código 10.21: *Todos los ciclos del grafo G que inician en un vértice dado.*

```
def KCiclos(v, G):
    """
    Todos los ciclos de un grafo G que inician en el vértice v.
    """
    if en(v,VInf(G.V)):
        KC = conj()
        for k in range(1, card(G.V)+1):
            kc = kCiclos(k, v, G)
            if neg(esVacio(kc)):
                KC = KC + kc
        return KC
    else:
        return False
```

Ejemplo 10.24

Considera nuevamente el grafo G de la figura en el ejemplo 9.19 en la página 208, que aparece nuevamente aquí.



Todos los caminos que inician en el vértice 2 son:

```
>>> KCiclos(2,G)
[[2, 3, 4, 2], [2, 3, 7, 2], [2, 3, 7, 4, 2]]
>>>
```

Todos los caminos que inician en el vértice 7 son:

```
>>> KCiclos(7,G)
[[7, 2, 3, 7], [7, 4, 2, 3, 7]]
>>>
```

10.4.6 Ciclos hamiltonianos

Los caminos que son ciclos, ofrecen respuesta a varios problemas importantes y que son frecuentemente establecidos. Por ejemplo considera una empresa que se dedica a entregar paquetes que son enviados de un lado a otro, la empresa comisiona a un empleado para entregar los paquetes a diferentes localidades en su territorio. El administrador de la empresa se pregunta si es posible diseñar una ruta de entrega que pase por todos los lugares en los que que debe hacer una entrega, pero sin pasar dos veces por el mismo sitio, ya que es importante optimizar recursos.

Este problema se conoce como el «Problema del Agente Viajero» [Ver51, DFJ54] que es un problema clásico de Ciencias Computacionales, cuya base matemática se basa en el trabajo de que se puede resolver con teoría de grafos, y aquí te muestro una solución que no solamente responde sí o no, sino que ofrece todas las alternativas, si las hay, desafortunadamente es un algoritmo de fuerza bruta, lo que significa que el costo computacional puede aumentar significativamente con el número de vértices y el número de aristas.

El algoritmo esta basado en la siguiente definición.

Definición 10.4.4 Si φ es un ciclo en un grafo $H \hookleftarrow [\![V;A]\!]$ con $|H.V| \mapsto n$, decimos que φ es un **ciclo hamiltoniano** si $|\varphi| = n$.

Si un grafo H de n vértices tiene un ciclo hamiltoniano, entonces se pueden encontrar otros $n-1$ ciclos hamiltonianos iguales al primero, cada uno iniciando de un vértice

diferente, es decir, en un mismo grafo G con un ciclo hamiltoniano, los demás ciclos hamiltonianos son iguales, sin importar el vértice inicial, esto es debido a que un ciclo recorre todos los vértices, de modo que sin importar el vértice origen, el ciclo hamiltoniano garantiza dos cosas:

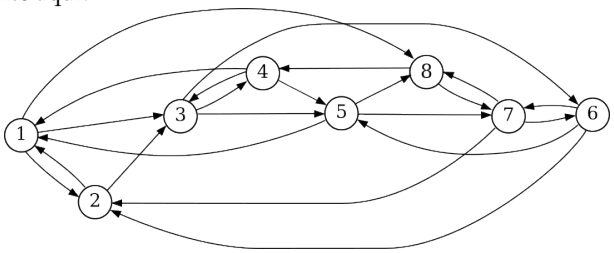
1. El ciclo recorre todos los vértices sin repetición. Porque la longitud del ciclo es igual a la cantidad de vértices.
2. Se alcanza el vértice origen del camino. Es obvio porque es un ciclo.

Código 10.22: *Todos los ciclos hamiltonianos de un grafo G*

```
def HCiclos(G:Grafo):
    """
    Devuelve todos los ciclos hamiltonianos de un grafo G.
    """
    return kCiclos(card(G.V), car(G.V).inf, G)
```

Ejemplo 10.25

Considera nuevamente el grafo H de la figura 10.10 en la página 239, el cual se muestra nuevamente aquí.



Para encontrar todos los ciclos hamiltonianos hacemos:

```
>>> HCiclos(H)
[[1, 2, 3, 6, 7, 8, 4, 5, 1],
 [1, 3, 4, 5, 8, 7, 6, 2, 1],
 [1, 8, 4, 3, 5, 7, 6, 2, 1],
 [1, 8, 4, 3, 6, 5, 7, 2, 1],
 [1, 8, 7, 6, 2, 3, 4, 5, 1],
 [1, 2, 3, 6, 5, 7, 8, 4, 1]]
>>>
```

Todos los ciclos encontrados se muestran en la figura 10.12.

Definición 10.4.5 Un grafo $G \hookleftarrow [\![V;A]\!]$ es un **grafo hamiltoniano**, si tiene al menos un camino hamiltoniano o un ciclo hamiltoniano.

El grafo de la figura 10.10 es un grafo hamiltoniano porque tiene al menos un ciclo hamiltoniano.

El procedimiento es sencillo. Primero consideramos el subconjunto de caminos de longitud $|G.V|-1$, luego se verifica que exista al menos un camino en el subconjunto que tenga al vértice origen como adyacente.

Código 10.23: *Determina si un grafo es hamiltoniano*

```
def esHamiltoniano(G:Grafo):
    """
    Determina si un grafo G es hamiltoniano.
    """
    k = card(G.V)-1
    v = car(G.V).inf
    C = kCaminos(k, v, G)
    return existeUn(lambda c: esAdy(v, cT(c,G), G) , C)
```

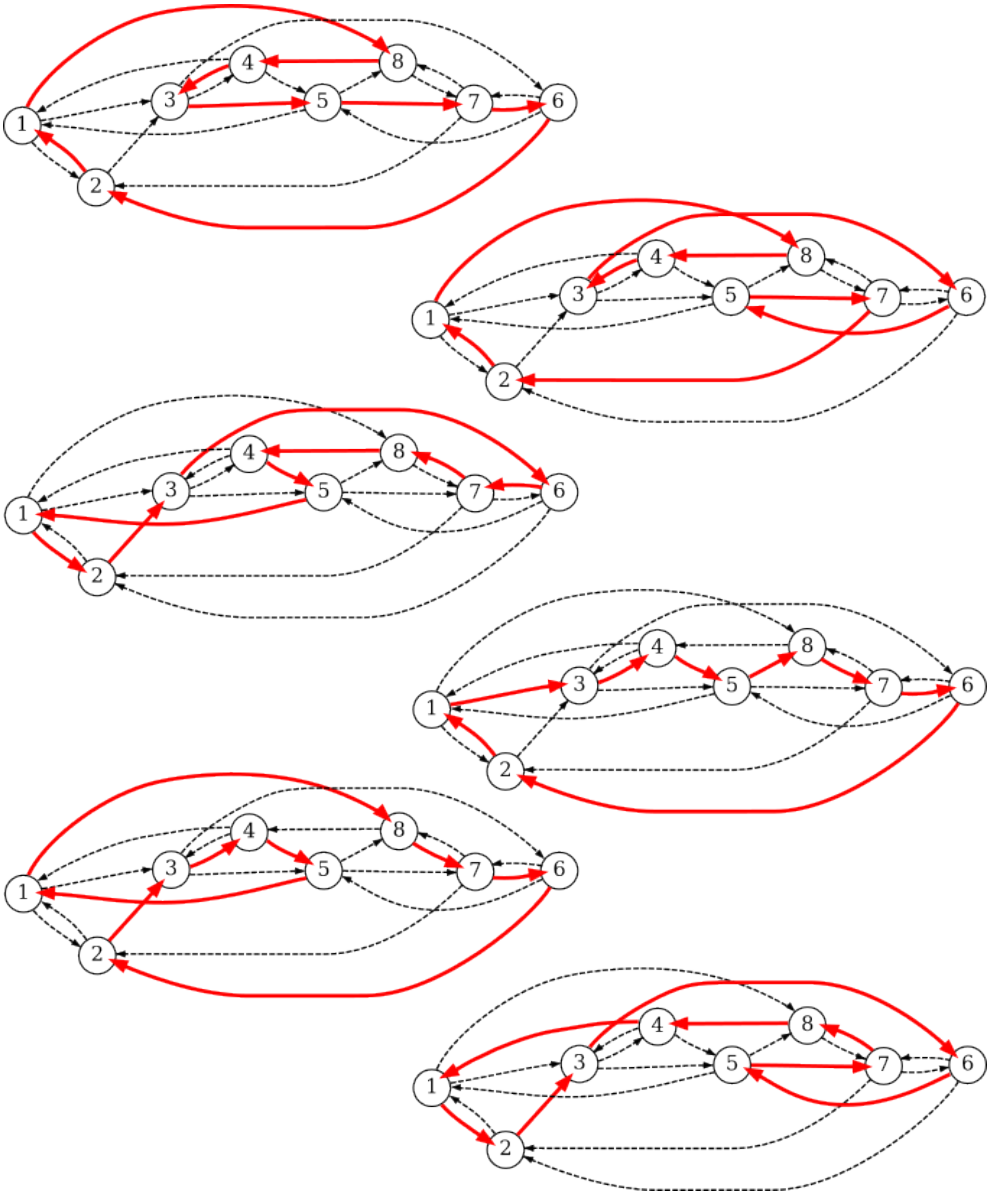


Figura 10.12: *Ciclos hamiltonianos del grafo H de la figura 10.10 en la página 239.*

Ejemplo 10.26

Determinar si el grafo *H* es un grafo hamiltoniano.

```
>>> esHamiltoniano(H)
True
>>>
```

Ejercicios

Para los ejercicios de esta sección, considera el siguiente grafo $D \leftarrow [\![V; A]\!]$, con las siguientes listas de adyacencias:

$\langle 1, 5, 8 \rangle$,
$\langle 2, 4, 10 \rangle$,
$\langle 3, 4, 5 \rangle$,
$\langle 4, 1, 3, 8 \rangle$,
$\langle 5, 6 \rangle$,
$\langle 6, 6, 7 \rangle$,
$\langle 7, 1, 8 \rangle$,
$\langle 8, 2, 4 \rangle$,
$\langle 9, 3 \rangle$,
$\langle 10, 2, 9 \rangle$.

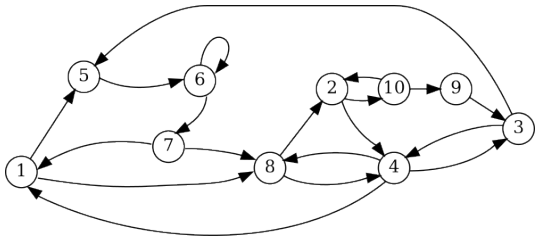


Figura 10.13: *Grafo D*

1. Considera el grafo D anterior y determina si las siguientes secuencias son paseos en D.
   a) $\langle 1, \langle 1, 5 \rangle, 5, \langle 5, 6 \rangle, 6, \langle 6, 6 \rangle, 6, \langle 6, 7 \rangle, 7, \langle 7, 8 \rangle, 8 \rangle$.
   b) $\langle 10, \langle 10, 2 \rangle, 2, \langle 2, 10 \rangle, 10, \langle 10, 2 \rangle, 2, \langle 2, 8 \rangle, 8, \langle 8, 4 \rangle, 4 \rangle$.
   c) $\langle 3, \langle 3, 5 \rangle, 5, \langle 5, 6 \rangle, 6, \langle 6, 7 \rangle, 7, \langle 7, 1 \rangle, 1, \langle 1, 8 \rangle, 8 \rangle$
2. En los siguientes paseos, ¿cuál es el vértice origen, el vértice término y los vértices internos?
   a) $\langle 7, \langle 7, 1 \rangle, 1, \langle 1, 5 \rangle, 5, \langle 5, 6 \rangle, 6, \langle 6, 6 \rangle, 6 \rangle$
   b) $\langle 8, \langle 8, 4 \rangle, 4, \langle 4, 3 \rangle, 3, \langle 3, 5 \rangle, 5, \langle 5, 6 \rangle, 6 \rangle$
   c) $\langle 10, \langle 10, 9 \rangle, 9, \langle 9, 3 \rangle, 3, \langle 3, 5 \rangle, 5 \rangle$
3. Calcula la longitud de los siguientes paseos:
   a) $\langle 7, \langle 7, 1 \rangle, 1, \langle 1, 5 \rangle, 5, \langle 5, 6 \rangle, 6, \langle 6, 6 \rangle, 6 \rangle$
   b) $\langle 8, \langle 8, 4 \rangle, 4, \langle 4, 3 \rangle, 3, \langle 3, 5 \rangle, 5, \langle 5, 6 \rangle, 6 \rangle$
   c) $\langle 10, \langle 10, 9 \rangle, 9, \langle 9, 3 \rangle, 3, \langle 3, 5 \rangle, 5 \rangle$
4. Calcula el paseo inverso de los siguientes paseos.
   a) $\langle 3 \rangle$
   b) $\langle 2, \langle 2, 10 \rangle, 10, \langle 10, 9 \rangle, 9, \langle 9, 3 \rangle, 3 \rangle$
   c) $\langle 8, \langle 8, 4 \rangle, 4, \langle 4, 3 \rangle, 3, \langle 3, 4 \rangle, 4 \rangle$
5. Determina si el grafo D es hamiltoniano, si lo es, enlista los ciclos hamiltonianos del grafo.

V

Apéndices

Código fuente por capítulo A

Código fuente en el capítulo 1

```
%% Capitulo 1

from inspect import signature

def aridad(fun):
    """
    Obtiene la aridad de una función.
    """
    return len(signature(fun).parameters)

def neg(p:bool)-> bool:
    """ La negación de una proposición
    Devuelve el valor de verdad opuesto a p.
    """
    if p:
       return False
    else:
       return True

def T(p:bool,q:bool)-> bool:
    """ Predicado Verdad
    Sin importar los par\'ametros, devuelve True.
    """
    return True

def F(p:bool,q:bool)-> bool:
    """ Predicado Falsedad
    Sin importar los par\'ametros, devuelve False.
    """
    return False

def P(p:bool, q:bool)-> bool:
    """ Primera componente
    Devuelve el valor del primer parámetro p.
    """
```

```
    return p

def Q(p:bool, q:bool)-> bool:
    """ Segunda componente
    Devuelve el valor del segundo parámetro q.
    """
    return q

def y(p:bool, q:bool)-> bool:
    """ Conjunción
    Devuelve True cuando ambas proposiciones son True,
    devuelve False en cualquier otro caso.
    """
    if p:
        return q
    else:
        return False

def o(p:bool, q:bool)-> bool:
    """ Disyunción de dos proposiciones:
    Devuelve True cuando al menos una de las proposiciones es True,
    si ambas proposiciones son False, entonces devuelve False.
    """
    if p:
        return True
    else:
        return q

def ox(p:bool, q:bool)-> bool:
    """ Disyunción exclusiva:
    Devuelve True cuando ambas proposiciones tienen valor diferente.
    Cuando las proposiciones coinciden en su valor, devuelve False.
    """
    if p:
        return neg(q)
    else:
        return q

def impl(p:bool, q:bool)-> bool:
    """ La implicación
    Devuelve True ya sea que el antecedente es False, o bien cuando
    el antecedente es True y el consecuente es True.
    Devuelve False cuando el antecedente es False y el consecuente es False.
    """
    if p:
        return q
    else:
        return True

def ssi(p:bool, q:bool)-> bool:
    """ Doble condicional
    Devuelve True cuando ambas proposiciones tienen el mismo valor
    y devuelve False si las proposiciones son de diferente valor.
    """
    if p:
        return q
    else:
        return neg(q)


def aridad(fun):
    """
    Obtiene la aridad de una función.
    """
    return len(signature(fun).parameters)


def casosPrueba(n, res=None):
    """
    Genera una lista de casos de prueba para predicados
    de aridad n.
    """
    if res == None: res=[[]]
```

```
    if n==0:
        return res
    else:
        res1 = list(map(lambda t:[True]+t, res))
        res2 = list(map(lambda t:[False]+t, res))
        res = res1 + res2
        return casosPrueba(n-1, res)

def vectorResultados(Pred)-> list:
    """
    Genera un vector de resultados para la evaluación
    del predicado Pred.
    """
    ar = aridad(Pred)
    Lvals = casosPrueba(ar)
    V = list(map(lambda t:Pred(*t), Lvals))
    return V

def logEqv(Pred1, Pred2)-> list:
    """
    Verifica la equivalencia lógica de los predicados
    Pred1 y Pred2.
    """
    v1 = vectorResultados(Pred1)
    v2 = vectorResultados(Pred2)
    return v1 == v2

def sheffer(p:bool, q:bool)-> bool:
    """
    sheffer(p,q)-> bool
    p: bool
    q: bool
    """
    pass  # @Escribe aquí tu código@

def peirce(p:bool, q:bool)-> bool:
    """
    peirce(p,q)-> bool
    p: bool
    q: bool
    """
    pass  # @Escribe aquí tu código@
```

Código fuente en el capítulo 2

```
def Y(*Lprop:list)-> bool:
    """
    Disyunción extendida
    Recibe una lista no determinada de proposiciones.
    """
    if Lprop == ():
        return True
    elif car(Lprop):
        return Y(*cdr(Lprop))
    else:
        return False

def O(*Lprop:list)-> bool:
    """
    Conjunción extendida
    Recibe una lista no determinada de proposiciones.
    """
    if Lprop == ():
        return False
    elif car(Lprop):
        return True
    else:
        return O(*cdr(Lprop))

def paraTodo(P, D:list)-> bool:
```

```
    """ Cuantificador universal
    Versión recursiva
    Se verifica el predicado P en todos los elementos del dominio D.
    """
    if D == []:
        return True
    elif P(car(D)):
        return paraTodo(P, cdr(D))
    else:
        return False

def existeUn(P, D:list)-> bool:
    """
    Versión recursiva del cuantificador existencial.
    Verifica que al menos un elemento del dominio D cumpla el predicado P.
    """
    if D == []:
        return False
    elif P(car(D)):
        return True
    else:
        return existeUn(P, cdr(D))

def paraTodo(P, *D)-> bool:
    """ Cuantificador universal
    Devuelve True cuando todas las proposiciones en el dominio D
    son True, de otro modo devuelve False.
    """
    return all(map(P, *D))

def existeUn(P, *D:list)-> bool:
    """ Cuantificador existencial
    Devuelve True cuando al menos una de las proposiciones
    son True, y devuelve False cuando todas son False.
    """
    return any(map(P, *D))

def existUnUnico(P, D:list)-> bool:
   """ Unicidad en la existencia
   Devuelve True cuando existe un único elemento
   en el dominio D, que cumple el predicado P.
   """
   pass # <- Escribe aquí tu función
```

Código fuente en el capítulo 3

```
def quitaDup(L:list, R:list = [])-> list:
    """
    Quita los elementos duplicados en una lista.
    """
    if esVacio(L):
        return R
    elif en(car(L), cdr(L)):
        return quitaDup(cdr(L), R)
    else:
        return quitaDup(cdr(L), R+[car(L)])

def conj(*items)-> list:
    """ Crea un conjunto de manera implícita o explícita.
    Si no hay argumentos, crea un conjunto vacío
    Si items es una lista no determinada, el conjunto es la lista
    con los elementos dados.
    Si items es un predicado, el conjunto es definido por ese predicado.
    """
    params = list(items)
    if items==():
        return []
    elif callable(params[0]):
        return params[0]
    else:
```

```
245         return quitaDup(params)
246
247 def en(a, C=None)-> bool:
248     """ Pertenencia.
249     Determina la pertenencia de un elemento a un conjunto
250     sin importar si se trata de un conjunto extensional o intencional
251     Se devuelve un valor booleano.
252     """
253     if isinstance(C, list):
254         return existeUn(lambda x: x == a, C)
255     elif callable(C):
256         try:
257             res = C(a)
258         except TypeError:
259             return False
260         else:
261             return res
262     else:
263         return False
264
265 def esVacio(C)-> bool:
266     """
267     Determina si C es el conjunto vacío.
268     """
269     return C == vacio or C == ()
270
271 def card(A:list, n:int=0)-> int:
272     """
273     Calcula la cardinalidad de un conjunto dado por extensión.
274     """
275     if esVacio(A):
276         return n
277     else:
278         return card(cdr(A), n+1)
279
280 def esUnit(A:list)-> bool:
281     """ Determina si un conjunto es unitario.
282     Un conjunto es unitario si tiene exactamente
283     un elemento.
284     """
285     return card(A) == 1
286
287 def esSubc(A:list, B:list)-> bool:
288     """ A es subconjunto de B.
289     Determina si A es subconjunto de B
290     si todos los elementos de A
291     pertenecen también al conjunto B.
292     """
293     return paraTodo(lambda a: en(a, B), A)
294
295 def subc(P:callable, D:list)-> list:
296     """ Crea un subconjunto.
297     Construye un subconjunto de D con aquellos
298     elementos que verifican el predicado P.
299     """
300     return list(filter(P, D))
301
302 def esSubcP(A:list, B:list)-> bool:
303     """ Determina si A es subconjunto propio de B.
304     Un conjunto A es subconjunto propio de B si
305     A es subconjunto de B y existe al menos un elemento de B que
306     no se encuentre en A.
307     """
308     res = y(esSubc(A,B), existeUn(lambda b:neg(en(b,A)),B))
309     return res
310
311 def cIguales(A:list, B:list)-> bool:
312     """ Igualdad en conjuntos.
313     Determina si A es igual a B
314     determinando si A es subconjunto de B y
315     B es subconjunto de A.
316     """
317     return y(esSubc(A,B), esSubc(B,A))
```

Código fuente en el capítulo 4

```
319 def agrega(e, A:list)-> list:
320     """
321     Agrega un elemento e al conjunto A.
322     El elemento es agregado cuando no pertenece al conjunto.
323     Si el elemento ya pertenece, el conjunto permanece sin cambio.
324     """
325     R = R = list(A) if en(e,A) else list(A)+[e]
326     return R
327 
328 def union(A: list, B: list)-> list:
329     """ Union de dos conjuntos
330     Genera un conjunto con la unión de A con B.
331     Se devuelve el conjunto A junto con los elementos de B que no
332     pertenezcan a A.
333     """
334     if esVacio(B):
335         return A
336     else:
337         return union(agrega(car(B), A), cdr(B))
338 
339 def intersec(A: list, B: list)-> list:
340     """ Intersecci\'on de dos conjuntos
341     Genera un conjunto con la intersección de A con B.
342     Devuelve aquellos elementos del conjunto A
343     que pertenecen también al conjunto B.
344     """
345     return subc(lambda b:en(b,A), B)
346 
347 def difc(A: list, B: list)-> list:
348     """ Diferencia del conjunto A respecto al conjunto B.
349     Devuelve un conjunto con aquellos elementos
350     del conjunto A que no pertenecen al conjunto B.
351     """
352     return subc(lambda a:neg(en(a,B)), A)
353 
354 def quita(e, A:list)-> list:
355     """ Quitar un elemento de un conjunto.
356     Devuelve un conjunto con todos los elementos
357     del conjunto A, excepto el elemento e.
358     """
359     return difc(A,conj(e))
360 
361 def difSim(A:list, B:list)-> list:
362     """ Diferencia simétrica de un conjunto respecto de otro.
363     Devuelve el subconjunto con los elementos de la unión de A con B
364     que pertenecen a la diferencia simétrica de esos conjuntos.
365     """
366     return subc(lambda x:ox(en(x,A), en(x,B)), union(A,B))
```

Código fuente en el capítulo 5

```
368 def enCada(P:callable, D:list)-> list:
369     """
370     Aplica P en cada elemento de D.
371     """
372     return list(map(P, D))
373 
374 def cPot(A:list, R:list=conj(vacio))-> list:
375     """ Conjunto potencia de un conjunto
376     Calcula el conjunto potencia de un conjunto
377     A en un conjunto.
378     R guarda el resultado. Es un conjunto de conjuntos.
379     """
380     if esVacio(A):
381         return R
382     else:
383         return cPot(cdr(A), enCada(lambda X:agrega(car(A), X), R) + R)
```

```
384
385 def Union(*FamC)-> list:
386     """ Unión generalizada.
387     Recibe una lista no determinada de conjuntos en forma de listas
388     Devuelve una lista que es interpretada como la unión
389     de todos los conjuntos dados.
390     """
391     if esVacio(FamC):
392         return vacio
393     elif esVacio(cdr(FamC)):
394         return car(FamC)
395     else:
396         nuevoC = union(car(FamC), car(cdr(FamC)))
397         nuevaF = agrega(nuevoC, FamC[2:])
398         return Union(*nuevaF)
399
400 def esCub(F, A):
401     """ Verifica un cubrimiento
402     determina si la familia de conjuntos F
403     es un cubrimiento para el conjunto A
404     devuelve un valor booleano.
405     """
406     return esSubc(A, Union(*F))
407
408 def esPart(FamC:list, A:list)-> bool:
409     """
410     Determina si la familia de conjuntos FamC
411     es una partición del conjunto A.
412     """
413     return y(esCub(FamC, A),
414              paraTodo(lambda X:
415                       paraTodo(lambda Y:
416                                ssi(neg(cIguales(X,Y)),
417                                    esVacio(inters(X,Y))),
418                                FamC),
419                            FamC))
420
421 def Inters(*FamC)-> list:
422     """
423     Intersección generalizada
424     de una lista no determinada de conjuntos.
425     """
426     pass  #<-- aquí escribe tu código
```

Código fuente en el capítulo 6

```
428 def pPar(t:list):
429     """ Primero de par.
430     Obtiene el primer elemento de un par ordenado.
431     """
432     return car(t)
433
434 def sPar(t:list):
435     """ Segundo de par.
436     Obtiene el primero del resto de un par ordenado.
437     """
438     return car(cdr(t))
439
440 def esTupla(t)-> bool:
441     """ Verifica que un objeto sea una tupla.
442     Devuelve True, si t es precisamente una lista.
443     """
444     return isinstance(t, list)
445
446 def tupla(*items)-> list:
447     """ Crea una tupla.
448     Recibe una lista no determinada de elementos
449     de cualquier tipo, pero devuelve una lista con ellos.
450     """
451     return list(items)
```

```
452
453 def esTvacia(t:list)-> bool:
454     """ Verifica que una tupla esté vacía.
455     Devuelve True si el argumento es la constante tvacia.
456     Devuelve False si no lo es.
457     """
458     return t == tvacia
459
460 def tLong(t:list, l:int=0)-> int:
461     """ Calcula la longitud de una tupla.
462     Versión recursiva que cuenta los elementos en t.
463     """
464     if esTvacia(t):
465         return l
466     else:
467         return tLong(cdr(t), l+1)
468
469 def esSoliton(t:list)-> bool:
470     """ Verifica que una tupla sea solitón.
471     Se proporciona una lista y se calcula su longitud,
472     si la longitud es 1, entonces es un solitón.
473     """
474     return tLong(t) == 1
475
476 def tIguales(a:list, b:list)-> bool:
477     """ Verifica recursivamente la igualdad entre dos tuplas.
478     El resultado es True si son iguales y False en otro caso.
479     """
480     if ox(esTvacia(a), esTvacia(b)):
481         return False
482     elif y(esTvacia(a), esTvacia(b)):
483         return True
484     elif car(a)==car(b):
485         return tIguales(cdr(a), cdr(b))
486     else:
487         return False
488
489 def tConcat(a:list, b:list)-> list:
490     """ Concatenación.
491     Concatena por la derecha la tupla b a la tupla a.
492     """
493     return a + b
494
495 def tEscribe(t:list,e)-> list:
496     """ Escribe por derecha.
497     Escribe el elemento e al final de la tupla t.
498     """
499     if esTupla(t):
500         return tConcat(t,tupla(e))
501     else:
502         return tConcat(tupla(t), tupla(e))
503
504 def tKdivide(t:list, k:int)-> list:
505     """ Divide una lista en la posición k.
506     Devuelve una tupla con las los partes de la tupla original.
507     """
508     if k<0:
509         return tupla([],t)
510     if k>tLong(t):
511         return tupla(t,[])
512     else:
513         return tupla(t[:k], t[k:])
514
515 def tEscribeEnPos(t,e,k)-> list:
516     """ Escribe un elemento en una posición dada.
517     Devuelve una nueva lista, incluyendo el elemento insertado.
518     """
519     Ta = tKdivide(t, k-1)
520     Tb = car(Ta)
521     Tc = car(cdr(Ta))
522     return tConcat(tEscribe(Tb, e), Tc)
523
524 def TConcat(*T)-> list:
```

```
    T = tupla(*T)
    if esTvacia(T):
        return tvacia
    if esSoliton(T):
        return car(T)
    else:
        x = tConcat(car(T), car(cdr(T)))
        r = tConcat(tupla(x), cdr(cdr(T)))
        return TConcat(*r)

def pCart(A:list, B:list)-> list:
    """
    Calcula el producto cartesiano de dos conjuntos.
    """
    PC = enCada(lambda a: enCada(lambda b: tEscribe(a,b), B), A)
    return TConcat(*PC)

def esRel(R:list, A:list, B:list)-> list:
    """
    Determina si una lista de pares es una relación válida.
    """
    return esSubc(R, pCart(A, B))

def dom(R:list)-> list:
    """
    Obtiene el dominio de una relación.
    """
    l = tLong(car(R))
    D = conj(*enCada(lambda t: car(tKdivide(t, l-1)), R))
    D = [car(x) if esUnit(x) else x for x in D]
    return D

def ran(R:list)-> list:
    """
    Obtiene el rango de una relación.
    """
    l = tLong(car(R))
    C = conj(*enCada(lambda t: car(car(cdr(tKdivide(t, l-1)))), R))
    return C

def im(R:list, a)-> list:
    """
    Calcula la imagen de un elemento del dominio de la relación.
    """
    l = tLong(car(R))-1
    if neg(esTupla(a)): a = tupla(a)
    C = subc(lambda t: a == car(tKdivide(t, l)), R)
    D = conj(*enCada(lambda t: car(car(cdr(tKdivide(t, l)))), C))
    return D

def Im(R:list, A:list)-> list:
    """
    Calcula la imagen de un conjunto de elementos del dominio.
    """
    return Union(*enCada(lambda e:im(R, e), A))
```

Código fuente en el capítulo 7

```
def esRef(R:list, D:list=None)-> bool:
    """
    Determina si una relación es reflexiva
    """
    if D is None: D = union(dom(R), ran(R))
    return paraTodo(lambda a: en(tupla(a, a), R), D)

def esIrref(R:list, D:list=None)-> bool:
    """
    Determina si una relación es irreflexiva
    """
    if D is None: D = union(dom(R), ran(R))
```

```
    return paraTodo(lambda a: neg(en(tupla(a, a), R)), D)

def esSim(R:list)-> bool:
    """
    Determina si una relación es simétrica
    R: list
    """
    return paraTodo(lambda t:en(tInversa(t), R), R)

def esAsim(R:list)-> bool:
    """
    Determina si una relación es asimétrica
    R: list
    """
    return paraTodo(lambda t:neg(en(tInversa(t), R)), R)

def esAntisim(R:list)-> bool:
    """
    Determina si una relación es antisimétrica
    """
    return paraTodo(lambda t:
                    impl(en(tInversa(t),R),
                         car(t)==car(cdr(t))), R)

def cerrRef(R):
    """
    Obtiene la cerradura reflexiva de una relación
    """
    A = union(dom(R), ran(R))
    pRef = enCada(lambda a:tupla(a,a), A)
    R = union(R, pRef)
    return R

def cerrSim(R):
    """
    Obtiene la cerradura simétrica de una relación
    """
    CS = enCada(lambda p:tInversa(p), R)
    R = union(R,CS)
    return R

def esROP(R:list, D = None)-> bool:
    """
    Determina si una relación es parcialmente ordenada
    """
    if D is None: D = union(dom(R), ran(R))
    return paraTodo(lambda P: P(R), [esRef, esAntisim, esTran])

def rInv(R:list)-> list:
    """
    Obtiene la relación inversa de una relación
    """
    return enCada(lambda t: tInversa(t), R)

def rComp(R:list, S:list)-> list:
    """
    La composición de la función R con S
    """
    RS = enCada(lambda a:enCada(lambda c: tupla(a,c),
                                Im(S,im(R,a))),
                dom(R))
    return Union(*RS)
```

Código fuente en el capítulo 8

```
def esFun(f:list, A:list=None, B:list=None)-> bool:
    """
    Verifica que f sea una función
    """
    if A is None: A = dom(f)
```

```
660     if B is None: B = ran(f)
661     return paraTodo(lambda a: esSoliton(im(f,a)), A)
662 
663 def ev(f:list, x):
664     """
665     Calcula la evaluación de la función en un elemento del dominio
666     """
667     y = im(f,x)
668     if esSoliton(y):
669         return car(y)
670     elif esVacio(y):
671         return "NoDef"
672     else:
673         return "NoFun"
674 
675 def preim(f:list, b)-> list:
676     """
677     Calcula la preimagen de un elemento del rango
678     """
679     return subc(lambda a: en(tupla(a,b),f))
680 
681 def esIny(f:list, A:list=None, B:list=None)-> bool:
682     """
683     Verifica que una función sea inyectiva
684     """
685     if A is None: A = dom(f)
686     if B is None: B = ran(f)
687     return paraTodo(lambda b: esUnit(preim(f,b)), B)
688 
689 def esSobre(f:list, A:list=None, B:list=None)-> bool:
690     """
691     Verifica que una función sea sobreyectiva
692     """
693     if A is None: A = dom(f)
694     if B is None: B = ran(f)
695     return B == ran(f)
696 
697 def esBiy(f:list, A:list=None, B:list=None)-> bool:
698     """
699     Verifica que una función sea biyectiva
700     """
701     if A is None: A = dom(f)
702     if B is None: B = ran(f)
703     return y(esIny(f,A,B), esSobre(f,A,B))
704 
705 def esInv(f:list)-> bool:
706     """
707     Verifica que una función sea invertible.
708     """
709     return esBiy(f)
710 
711 def esNoInv(f:list)-> bool:
712     """
713     Verifica que una función sea no invertible.
714     """
715     return neg(esBiy(f))
716 
717 def fPerm(p:list, A:list)-> list:
718     """
719     Genera una permutación de un conjunto A
720     dando un ciclo p.
721     """
722     B = tEscribe(cdr(p), car(p))
723     C = list(map(lambda a,b:tupla(a,b), p, B))
724     D = difc(A, p)
725     E = enCada(lambda d:tupla(d,d), D)
726     return tConcat(C, E)
727 
728 def pProd(p1:list, p2:list, A:list)-> list:
729     """
730     Producto de permutaciones cíclicas sobre
731     un conjunto A
732     """
```

```
733     return rComp(fPerm(p1,A), fPerm(p2,A))
```

Código fuente en el capítulo 9

```
734 class Vertice:
735     """
736     Unidad fundamental e información en Grafos.
737     Crea un vértice con la información dada.
738     """
739     def __init__(self, inf = None):
740         self.inf = inf
741         # otros atributos
742 
743     def __str__(self):
744         return f"[{self.inf}]"
745 
746 def VInf(LV:list):
747     """
748     Obtiene la información de una lista de vértices [objetos].
749     """
750     return enCada(lambda v: v.inf, LV)
751 
752 def vIguales(u:Vertice, v:Vertice)-> bool:
753     """
754     Determina si el vértice u es igual al vértice v.
755     """
756     return u.inf == v.inf
757 
758 def vEnV(u, V:list)-> Vertice or bool:
759     """
760     Busca el vértice u en el conjunto de vértices V,
761     dando la información del vértice
762     y un conjunto de vértices.
763     """
764     U = subc(lambda v: u == v.inf, V)
765     if esVacio(U):
766         return False
767     else:
768         return car(U)
769 
770 class Arista:
771     """
772     Se define una arista de la forma <vi, vf> donde
773     vi es el vértice inicial y vf es el vértice final.
774     """
775     def __init__(self, vi = None, vf = None):
776         self.vi = vi if isinstance(vi,Vertice) else Vertice(vi)
777         self.vf = vf if isinstance(vf,Vertice) else Vertice(vf)
778         self.inf = [self.vi.inf, self.vf.inf]
779 
780 def AInf(A:list)-> list:
781     """
782     Obtiene la información de todas las aristas de la lista dada.
783     """
784     return enCada(lambda a: a.inf, A)
785 
786 def aIguales(a:Arista, b:Arista):
787     """
788     Determina si dos aristas son iguales.
789     """
790     return y(vIguales(aVi(a), aVi(b)), vIguales(aVf(a),aVf(b)))
791 
792 class Grafo:
793     """
794     Estructura de vértices relacionados mediante aristas.
795     """
796     def __init__(self, V=None, A=None, Nombre = "G"):
797         self.V = V
798         self.A = A
799         self.Nombre = Nombre
```

```
        self.inf = None if V is None else [VInf(self.V), AInf(self.A)]
        self.Lady = paresArels(AInf(self.A))

    def __str__(self):
        return f"[{VInf(self.V)}; {AInf(self.A)}]"

def genGrafo(VInfo: list, AInfo:list)-> list:
    """
    Crea un grafo tomando la información de los vértices y aristas.
    """
    V = enCada(lambda vinf: Vertice(vinf), VInfo)
    A = []
    for a in AInfo:
        vi = vEnV(a[0], V)
        vf = vEnV(a[1], V)
        A += [Arista(vi,vf)]
    return Grafo(V,A)

def esAdy(v, u, G:Grafo)-> bool:
    """
    Determina si el vértice v es adyacente al vértice u en el grafo G.
    """
    A = AInf(G.A)
    return en(tupla(u,v), A)

def vAdy(v, G:Grafo)-> list:
    """
    Obtiene la lista de vértices adyacentes a un vértice en un grafo.
    """
    return im(AInf(G.A),v)

def esSubgrafo(G:Grafo, H:Grafo):
    """
    Determina si un grafo es subgrafo de otro.
    """
    return y(esSubc(VInf(G.V), VInf(H.V)), esSubc(AInf(G.A), AInf(H.A)))

def gInv(G:Grafo)-> Grafo:
    """
    Genera el grafo inverso de un grafo.
    """
    Ainv =rInv(AInf(G.A))
    return genGrafo(VInf(G.V), Ainv)

def gCompl(G:Grafo)-> Grafo:
    """
    Crea el grafo complementario de un grafo.
    """
    V = VInf(G.V)
    A = AInf(G.A)
    Ac = subc(lambda a:neg(en(a, A)), pCart(V,V))
    return genGrafo(V,Ac)
```

Código fuente en el capítulo 10

```
def wO(w:list, G:Grafo):
    """
    Devuelve el origen de un paseo.
    """
    return car(w) if esPaseo(w,G) else False

def wT(w:list, G:Grafo):
    """
    Devuelve el término de un paseo.
    """
    return w[-1] if esPaseo(w, G) else False

def esPaseoNulo(w:list, G)-> bool:
    """
    Verifica que w sea un paseo nulo en el grafo G.
```

```
    """
    return y(esPaseo(w, G), esSoliton(w))

def wLong(w:list)-> int:
    """
    Devuelve la longitud de un paseo.
    """
    return (card(w)-1) // 2

def esPaseoUnit(w:list, G:Grafo)-> bool:
    """
    Determina si w es un paseo unitario en G.
    """
    return y(esPaseo(w,G), wLong(w) == 1)

def wInv(w:list, G:Grafo)-> list:
    """
    Devuelve el paseo inverso de w si es un paseo.
    """
    res = tInversa([tInversa(i) if esTupla(i) else i for i in w])
    return res if esPaseo(res,G) else False

def aristasApaseo(r:list)-> list:
    """
    Transforma una secuencia de aristas en una secuencia tipo paseo.
    """
    if esTvacia(r):
        return tupla()
    else:
        wr = [car(car(r))]
        for a in r:
            wr = tEscribe(wr, a)
            wr = tEscribe(wr, sPar(a))
    return wr

def esRuta(r:list, G:Grafo)-> bool:
    """
    Verifica que una secuencia de aristas sea una ruta.
    """
    return y(sinRep(r), esPaseo(aristasApaseo(r), G))

def rutaInv(r:list, G:Grafo)-> list or bool:
    """
    Obtiene la ruta inversa de una ruta si existe, si no devuelve False.
    """
    ri = tInversa(enCada(lambda a:tInversa(a), r))
    return ri if esRuta(ri, G) else False

def rutaConcat(r:list,t:list, G:Grafo)-> list or bool:
    """
    Concatena dos rutas
    """
    tc = tConcat(r,t)
    return tc if esRuta(tc,G) else False

def esCamino(cam:list, G:Grafo)-> bool:
    """
    Verifica que una lista de vértices sea un camino.
    """
    if esTvacia(cam):
        return False
    else:
        p = sinRep(cam)
        q = paraTodo(lambda i: esAdy(cam[i], cam[i-1], G), range(1,tLong(cam)))
        return y(p,q)

def cO(cam:list, G:Grafo):
    """
    Devuelve el vértice inicial de un camino.
    """
    return car(cam) if esCamino(cam, G) else False

def cT(cam:list, G:Grafo):
```

```
940         """
941         Devuelve el vértice terminal de un camino.
942         """
943         return cam[-1] if esCamino(cam, G) else False
944
945     def cLong(cam:list, G:Grafo)-> int or bool:
946         """
947         Devuelve la longitud de un camino.
948         """
949         return tLong(cdr(cam)) if esCamino(cam,G) else False
950
951     def cNulo(v, G:Grafo)-> list:
952         """
953         Genera un camino nulo con el vértice v.
954         """
955         V = VInf(G.V)
956         return tupla(v) if en(v, V) else False
957
958     def esCunit(p:list,G:Grafo)-> bool:
959         """
960         Determina si un camino es unitario.
961         """
962         return cLong(p,G) == 1
963
964     def csUnit(v, G:Grafo)-> list:
965         """
966         Obtiene todos los caminos unitarios con origen en un vértice dado.
967         """
968         co = cNulo(u,G)
969         return enCada(lambda v:tEscribe(co, v), vAdy(u, G))
970
971     def c1ext(c:list, G:Grafo)-> list:
972         """
973         Extiende un vértice un camino dado.
974         """
975         va = difc(vAdy(cT(c,G), G), c)
976         return enCada(lambda v:tEscribe(c, v), va)
977
978     def cs1ext(FC:list, G:Grafo)-> list:
979         """
980         Extiende una familia de caminos.
981         """
982         NF = enCada(lambda C: c1ext(C,G), FC)
983         return Union(*NF)
984
985     def kCaminos(k:int, v, G:Grafo, C:list = None)-> list:
986         """
987         Calcula todos los caminos de longitud k desde un vértice v.
988         """
989         if C is None: C = conj(cNulo(v,G))
990         if k == 0:
991             return C
992         else:
993             return kCaminos(k-1, v, G, C=cs1ext(C, G))
994
995     def esCiclo(c:list,G:Grafo)-> bool:
996         """
997         Determina si c es un ciclo.
998         """
999         d = c[:-1]
1000        if esCamino(d,G):
1001            return y(en(c[-1], vAdy(cT(d,G),G)), cO(d,G)==c[-1])
1002        else:
1003            return False
1004
1005    def cicloLong(c:list, G:Grafo)-> int:
1006        """
1007        Devuelve la longitud de un ciclo.
1008        """
1009        return len(cdr(c)) if esCiclo(c, G) else False
1010
1011    def corr(c:list):
1012        """
```

```
    Hace un corrimiento de un ciclo.
    """
    return tEscribe(cdr(c), car(cdr(c)))

def kcorr(k:int, c:list)-> list:
    """
    Hace k corrimientos del ciclo c.
    """
    k = k % tLong(c)
    alpha = c[:k]
    beta = c[k:]
    alphaprim = cdr(alpha)
    beta0 = car(beta)
    b = tEscribe(alphaprim, beta0)
    return tConcat(beta, b)

def ciclosIguales(c:list, d:list)-> bool:
    """
    Determina la igualdad entre dos ciclos.
    """
    if tLong(c) != tLong(d):
        return False
    else:
        i = 0
        flg = False
        while ox(i < tLong(c),flg):
            if tIguales(c,d):
                flg = True
            d = corr(d)
            i = i + 1
        return flg

def kCiclos(k, v, G):
    """
    Todos los ciclos de longitud k que inician en el vértice v del grafo G.
    """
    U = subc(lambda u: esAdy(v, u, G), VInf(G.V))
    C = kCaminos(k-1, v, G)
    Cv = subc(lambda c: en(cT(c, G), U), C)
    Ck = enCada(lambda c: tEscribe(c,v), Cv)
    return Ck

def KCiclos(v, G):
    """
    Todos los ciclos de un grafo G que inician en el vértice v.
    """
    if en(v,VInf(G.V)):
        KC = conj()
        for k in range(1, card(G.V)+1):
            kc = kCiclos(k, v, G)
            if neg(esVacio(kc)):
                KC = KC + kc
        return KC
    else:
        return False

def HCiclos(G:Grafo):
    """
    Devuelve todos los ciclos hamiltonianos de un grafo G.
    """
    return kCiclos(card(G.V), car(G.V).inf, G)

def esHamiltoniano(G:Grafo):
    """
    Determina si un grafo G es hamiltoniano.
    """
    k = card(G.V)-1
    v = car(G.V).inf
    C = kCaminos(k, v, G)
    return existeUn(lambda c: esAdy(v, cT(c,G), G), C)
```

Bibliografía

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2 edition, 1996.

[Ben78] Oddur Benediktsson. Notes on argument-parameter association in fortran. *SIGPLAN Not.*, 13(1):16–20, jan 1978.

[BM76] J. A. Bondy and U.S.R. Murty. *Graph Theory with Applications*. North Holland, 1976.

[Boo09] George Boole. *The Mathematical Analysis of Logic: Being an Essay Towards a Calculus of Deductive Reasoning*. Cambridge Univ PR, 2009.

[CA16] Alberto Cuevas Álvarez. *Python 3: Curso Práctico*. Ra-Ma, 2016.

[CC13] Irving M. Copi and Carl Cohen. *Introducción a la Lógica*. LIMUSA, México, (español) 2 edition, 2013.

[CF08] Nino B. Cocchiarella and Max A. Freund. *Modal Logic: An Introduction to its Syntax and Semantics*. Oxford University Press, 2008.

[CG15] Willem Conradie and Valentin Goranko. *Logic and Discrete Mathematics. A Concise Introduction*. Wiley & Sons, Limited, John, 2015.

[Chu36] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58:345, 1936.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ, 1941.

[CM09] Michel Chein and Marie Laure Mugnier. *Graph-Based Knowledge Representation: Computational Foundations of Conceptual Graphs*. Advanced Information and Knowledge Processing. Springer, 2009.

[Cun16] Daniel W. Cunningham. *Set Theory A First Course*. Cambridge Mathematical Textbooks. Cambridge University Press, USA, 2016.

[Das14] Abhijit Dasgupta. *Set Theory, With al Introduction to Real Points Sets*. Birkh auser, 2014.

[Deo16] Narsingh Deo. *Graph Theory with Applications to Engineering & Computer Science*. Dover Publications, Inc, 2016.

[Dev03] Keith Devlin. *Sets, Functions, and Logic*. Chapman & Hall/CRC, 2003.

[DFJ54] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a Large-Scale Traveling-Salesman Problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.

[Eri09] Martin J. Erickson. *Pearls of Discrete Mathematics*. Discrete Mathematics and its Applications, Kenneth H., Rosen ed. CRC Press Taylor & Francis Group, 2009.

[Eul12] Leonhard Euler. *Lettres A Une Princesse D'Allemagne Sur Divers Sujets de Physique Et de Philosophie, Vol. 2.* SARASWATI PR, October 2012.

[FdPSTF04] Max Fernández de Castro, Asunción Preisser, Luis Felipe Segura, and Yolanda Torres Falcón. *Lógica Elemental*. Universidad Autónoma Metropolitana. U. Iztapalapa, México, 2004.

[Fer09] Kevin K. Ferland. *Discrete Mathematics An Introduction to Proofs and Combinatorics*. Houghton Mifflin Company, 2009.

[Gar01] Manuel Garrido. *Lógica Simbólica*. Tecnos Grupo Anaya S.A., 4 edition, 2001.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice*. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier - Morgan Kaufmann, 2004.

[Hal74] Paul Richard Halmos. *Naive Set Theory*. Undergraduate Texts in Mathematics. Springer Science+Business Media New York, USA, 1974.

[Hun20] John Hunt. *A Beginners Guide to Python 3 Programming*. Springer, 2020.

[Joy02] David Joyner. *Adventures in Group Theory; Rubik's Cube, Merlin's Machine, and Other Mathematical Toys*. The Johns Hopkins University Press, 2002.

[KB84] Bernard Kolman and Robert C. Busby. *Estructuras de Matematicas Discretas para Computación*. Prentice Hall Hispanoamericana, S. A., 1984. Traducción de la primera edición en inglés.

[LaV09] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2009.

[Lot19] Steven F. Lott. *Mastering Object-Oriented Python*. Packt Publishing, 2 edition, 2019.

[Lya20] David Lyalin. Applying Euler Diagrams and Venn Diagrams to Concept Modeling. *Business Rules Journal Newsletter*, 21:25p, 02 2020.

[MH81] Anthony N. Michel and Charles J. Herget. *Applied Algebra and Functional Analysis*. Dover Books on Mathematics. Dover Publications, 1981.

[MQ02] José María Muñoz Quevedo. *Introducción a la Teoría de Conjuntos*. Universidad Nacional de Colombia, 2002.

[O'L16] Michael L. O'Leary. *A First Course in Mathematical Logic and Set Theory*. John Wiley & Sons, Inc., 2016.

[Pyt23] Python Software Foundation. *Python Documentation, version 3.11*. 2023. web site URL https://docs.python.org/3/tutorial/classes.html.

[Rau10] Wolfgang Rautenberg. *A Concise Introduction to Mathematical Logic*. Springer New York, 2010.

[Ren13] Casandra Rendell, editor. *Network Topologies : Types, Performance Impact and Advantages/Disadvantages*. Computer Networks. Nova Science Pub Inc, 2013.

[Ros04] Kenneth. H. Rosen. *Matemática Discreta y sus Aplicaciones*. McGraw-Hill, 5 edition, 2004.

[Sai00] Mark Sainsbury. *Logical Forms: An Introduction to Philosophical Logic*. John Wiley and Sons Ltd, 2 edition, 2000.

[SH92] Patrick Suppes and Shirley Hill. *Introducción a la Lógica Matemática*. Reverté, 1992.

[Smu14] Raymond Merrill Smullyan. *A Beginner's Guide to Mathematical Logic*. Dover Books on Mathematics. Dover Publications, 2014.

[Sto63] Robert R. Stoll. *Set Theory and Logic*. Dover Publications, Inc. New York, republished 1979 edition, 1963.

[Ver51] S. Verblunsky. On the shortest path through a number of points. *Proceedings of the American Mathematical Society*, 2(6):904–913, 1951.

Índice alfabético

Wilfrido Miguel Contreras Sánchez
Secretario de Investigación, Posgrado y Vinculación

Pablo Marín Olán
Director de Difusión, Divulgación Científica y Tecnológica

Agustín Abreu Cornelio
Jefe del Departamento Editorial de Publicaciones No Periódicas

Matemáticas Discretas:
Una perspectiva funcional con Python 3.x

Matemáticas Discretas es una disciplina obligatoria en las Ciencias Computacionales. En este libro se propone un amplio conjunto de definiciones que son propias del área. Se abordan temas de Cálculo Proposicional, Lógica de Predicados, Conjuntos, Relaciones, Funciones y Teoría de Grafos; todos los temas se describen tanto en el lenguaje de matemáticas como en el lenguaje de programación Python.
Las definiciones ofrecidas están escritas en el lenguaje de programación Python. Se ha seleccionado este lenguaje de programación por diversas razones, entre estas, porque Python es un lenguaje de propósito general y multiparadigma. Escribir las definiciones matemáticas usando la sintaxis y semántica de Python, acerca a los programadores a la formalidad y abstracción matemática, y 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.
Nació en Ciudad de México, México.

9 786076 066324