Apunte de Introducción a los Algoritmos

Pedro Sánchez Terraf*

26 de abril de 2017

Resumen

Esta corta guía está destinada a las primeras clases de la materia *Introducción a los Algoritmos* de las carreras de Analista y Licenciatura en Ciencias de la Computación. El material práctico está mayoritariamente basado en los Prácticos que se usaron durante el primer cuatrimestre del año 2016.

El autor es matemático; los alumnos se beneficiarán discutiendo y comparando este enfoque (y las diversas opiniones presentadas) con los otros docentes de la materia.

1. Presentación

1.1. Grandes Áreas de la Computación

La Computación se puede dividir, *grosso modo*, en 4 áreas distintas:

1. Ingeniería en Sistemas.
2. Ingeniería en Computación.
3. Ingeniería del Software.
4. **Ciencias de la Computación**.

Las describimos a continuación.

1.1.1. Ingeniería en Sistemas

El nombre completo de este área es *Ingeniería en Sistemas de Información*. Es decir, se dedica al manejo eficiente de grandes lotes de información, o *bases de datos*. Por ejemplo, la nómina de contribuyentes de Inmuebles de Córdoba tiene más de un millón de registros; cada uno con los datos básicos (dirección del inmueble, superficie, deudas,...) e incluso diversas conexiones entre los mismos (dos inmuebles con el mismo titular, etc). Cuando hay que imprimir los cedulones para el pago de dicho impuesto, hay que cotejar esa información con datos relativos a premios por pago cumplidor, deudas, registros bancarios, y un larguísimo etcétera. Si uno no tiene cuidado manejando esta cantidad de información, el mero cálculo de cuánto tiene que pagar cada quién puede demorar más de un mes. Por ello, en esta parte de la disciplina se estudia cuáles son las mejores maneras de procesar esta cantidad de datos.

La Universidad Tecnológica Nacional (UTN) tiene una carrera de computación especializada en esta área.

*CIEM-FAMAF.

1.1.2. Ingeniería en Computación

En este caso, se estudia es el diseño del hardware (cpu, chips,...) y cómo interactuarán dentro de una computadora. En cualquier caso, hay que tener en cuenta que estamos usando "computadora" en un sentido muy general: un aparato electrónico que se puede *programar* para realizar una tarea. Con esta definición, una tablet, un celular, un smart-tv, una impresora e incluso la tarjeta para pagar el colectivo son computadoras.

Esta subdisciplina utiliza esencialmente conocimientos de la Ingeniería Electrónica, y se estudia en la Facultad de Ciencias Exactas, Físicas y Naturales de la UNC.

1.1.3. Ingeniería del Software

Esta área se encarga de otro problema de *escala* (cuando algo es tan grande que es difícil manejar, como las bases de datos). Pero en este caso, lo que hay que organizar es la producción del software (programas). Por ejemplo, el funcionamiento de los grandes servicios web (Google, Facebook, Hotmail,...) es el resultado de la articulación de miles de partes de "programas" escritos por distintos equipos alrededor del mundo, y que funcionan en sincronización. De hecho, la Internet misma es un sistema de este tipo. Para desarrollar estos sistemas monstruosos, la Ingeniería del Software desarrolla herramientas y estándares para permitir su planeamiento y futuro mantenimiento (solución de problemas que ocurran o actualizaciones).

1.1.4. Ciencias de la Computación

Esto es lo que se estudia en las carreras de Analista y Licenciatura de esta facultad.

El objetivo principal de la carrera es la *programación*, el desarrollo de *programas* o software. Una descripción parcial de lo que es un programa (que quizá varios de Uds. tengan como visión) sería una lista de instrucciones que le damos a la computadora para que realice. Esta descripción ingenua es útil para muchos propósitos pero obviamente no todo es así. Nos quedemos con ella por un momento para seguir la historia.

En esta carrera aprenderán técnicas para desarrollar programas que resuelvan problemas muy diversos, y las razones teóricas que justifican muchas de esas técnicas. A veces, se puede demostrar teóricamente que cierta solución a un problema es la mejor que puede conseguirse, y para obtener estos resultados se requiere de un grado importante de herramientas matemáticas (y lógicas).

La diversidad de problemas a resolver nos llevarán a aprender un poco de cada una de las otras áreas de especialidad, y hay docentes especialistas en cada una de ellas en esta facultad. También por este mismo motivo, enfocarse en un "único lenguaje de programación" no es una buena estrategia. De hecho, aprenderán varios lenguajes distintos, que tendrán distintos *paradigmas* (Sección 1.2).

En nuestra Facultad se trabaja de manera especializada en varias áreas de las Ciencias de la Computación. Algunas de ellas son las siguientes:

1. *Computación de alto rendimiento.* En inglés, "High Performance Computing" (HPC). Se trata del desarrollo de "supercomputadoras" (que consisten de muchos cpus interconectados) y cómo hacer que una tarea se distribuya eficientemente entre los mismos.

2. *Verificación de sistemas críticos.* En muchas situaciones, la respuesta de una computadora es una cuestión de vida o muerte (sistema de un avión en vuelo, dosaje en radioterapia). Para estos sistemas, además de su diseño hace falta una garantía de que funcionan como deben.

3. *Imágenes Satelitales.* El costo de transmisión de datos en el espacio es muy grande, y usualmente esa información viene con mucho ruido. Es necesario procesarla para eliminar este último y a su vez extraer la mayor cantidad de información de la misma imagen.

4. *Inteligencia Artificial.* Es un campo muy amplio; desde la programación de robots físicos y virtuales (por ejemplo, que pueden chatear) hasta sistemas que pueden ganar (y ser campeones mundiales) de concursos de preguntas y respuestas. Esta tecnología también se usa para "adivinar" la respuesta más adecuada para problemas donde la información es incompleta o contradictoria.

5. *Procesamiento de Lenguaje Natural.* El proceso por el cual se extrae y organiza automáticamente información a partir de texto escrito en un idioma humano (por ejemplo, tendencias nacionales o mundiales en Twitter, reportes médicos de pacientes con alguna enfermedad específica,...).

6. *Investigación científica* en todas estas áreas y más. La carrera de Licenciatura, además de complementar la de Analista (y asegurarte un aumento de sueldo), prepara a los estudiantes para dar sus primeros pasos en la carrera científica. Estos pasos se concretan en el *Doctorado* (y de hecho, el título de Doctor/a da derecho a pedir un aumento de sueldo ¡mucho más grande!).

1.2. Paradigmas de Programación

Retomemos la discusión de qué es un programa. Como primera aproximación, dijimos que un programa era un conjunto de órdenes que se le da a la computadora. Realmente, esto sólo describe uno de los posibles estilos de programación. Para este resumen, aislaremos los siguientes:

1. Programación Imperativa.

2. Programación Lógica.

3. Programación orientada a Objetos.

4. Programación Funcional.

1.2.1. Programación Imperativa

Este estilo es el más "intuitivo" y es el que describimos más arriba. Simplemente le damos a la computadora órdenes "Suma esto, muestra aquello,..."; casi podría pensarse que son las instrucciones que se le da a alguien para nos haga una cuenta con una calculadora. La diferencia principal con una calculadora (y que marca el comienzo de la computación en sí) es que los lenguajes imperativos tienen órdenes *condicionales* ("Si el resultado es mayor a $100, comprá carne; si es $100 o menos, comprá fideos") y *ciclos* que repiten alguna tarea ("Mientras te quede para bondi, apostá al 22").

1.2.2. Programación Lógica

Aquí el énfasis no es tanto obtener un resultado luego de operar con algunos datos, sino plantear relaciones lógicas y ver qué conclusiones se pueden sacar de ellas. En este caso, un programa lógico será una lista de propiedades "Todo lo grande es azul", "A es grande", "B no es azul", y una vez *compilado*, le podemos preguntar "¿Es $A = B$?".

El autor de estas notas confiesa que su ignorancia alcanza un máximo en este paradigma de programación.

1.2.3. Programación orientada a Objetos

En este paradigma, un programa es un descripción de "clases de objetos" que tienen ciertas propiedades (*atributos*) y operaciones asociadas (*métodos*). El ejemplo típico es el de una cuenta bancaria. Podemos pensarla como un objeto particular, que tendrá algunos datos asociados (titular, saldo, límite de extracción...) y las entre las operaciones asociadas se hallarán `mostrarSaldo` (imaginar qué hace), `extracción` (que dado un número entero entrega esa cantidad de dinero, si es menor al límite de extracción y al saldo), etcétera. En el programa se enumeran las enumeran todos los atributos y métodos que tienen todos los objetos de la misma *clase*.

1.2.4. Programación Funcional

En la Programación funcional, un programa es simplemente una lista de definiciones de funciones. Estas funciones son la solución al problema que queremos resolver. Éste es el paradigma que estudiaremos en esta materia.

En la escuela hemos aprendido a calcular con funciones numéricas como la suma, producto, y otras quizá más complicadas como el coseno y el logaritmo. Todas estas tienen en común que reciben como dato ("comen") números y devuelven números como resultado. Uno de los objetivos de esta materia es expandir el panorama de qué *tipo de datos* puede comer y devolver una función, y de esta manera ampliaremos en gran medida el universo de objetos con los cuales podemos "calcular".

1.3. Contenidos, objetivos y datos útiles de esta materia

1.3.1. Lo que aprenderemos

1. Extender, ampliar, lo que entendemos por cálculo o cómputo.
2. Calcular de manera ordenada, justificando cada paso.
3. Plantear problemas sencillos y resolverlos usando este cálculo.
4. Tener una herramienta (técnica) —¡que también es un cálculo!— que permita asegurarnos que nuestras soluciones a los problemas son las correctas (funcionan).

1.3.2. Cómo lo aprenderemos

1. Es eminentemente práctica.
2. La parte teórica consiste de las reglas de cálculo y definiciones.

1.3.3. Cómo aprobaremos esta materia

1. ¡Mucha práctica!
2. Esquema y fecha de parciales y recuperatorios. **Buena noticia**: Los recuperatorios *anulan la historia pasada*.

2. Material Teórico

2.1. Nuestro "idioma": el Formalismo Básico

Tipos: *Num*, *Char*, *Bool*. A partir de ellos, Listas y Tuplas.
Discusión: utilizar funciones para resolver problemas.

2.2. Definiciones de funciones

La construcción básica en un programa funcional es la definición de funciones. Para definir una función, necesito decir qué "come" y qué "devuelve". Lo que "come" o consume una función se denominan **argumentos** y lo que "devuelve" o su resultado se denomina su **valor** para dichos argumentos. Como un ejemplo muy zonzo, consideremos la función *suma* que suma dos enteros.

$$\begin{aligned} &suma : Int \to Int \to Int \\ &suma.x.y \doteq x + y \end{aligned} \tag{1}$$

Notemos que una definición de función como la (1) introduce dos cosas:

1. Un nuevo símbolo que nombra la función (en este caso, f) y qué tipo tiene.
2. Una nueva igualdad, que es verdadera para todos los valores de las variables (i.e., la igualdad $suma.x.y = x + y$).

Decimos que la igualdad $suma.x.y = x + y$ introducida por la definición (1) es **válida**: es verdadera para todos los valores de sus variables. En tal sentido, funciona como un axioma o un teorema (por ejemplo, como la conmutatividad de la suma, $a + b = b + a$).

Como las definiciones introducen igualdades (y la igualdad es **simétrica**, i.e. $a = b$ implica $b = a$), podemos pasar de la expresión $suma.x.y$ a $x + y$ y viceversa. Sin embargo le pondremos dos nombres distintos a cada una de los procesos. Por ejemplo, si calculamos $suma.9.16$,

$$\begin{array}{ll} & suma.9.16 \\ = \{ & \text{Definición de } suma \ \} \\ & 9 + 16 \\ = \{ & \text{Aritmética } \} \\ & 25, \end{array}$$

estamos usando la igualdad introducida por (1) de izquierda a derecha. En tal caso, decimos que estamos **desplegando** la definición. Por otro lado, si la usamos de derecha a izquierda, "haciendo aparecer" la función *suma* como a continuación:

$$\begin{array}{ll} & 256 \\ = \{ & \text{Aritmética } \} \\ & 128 + \underline{64 + 64}, \\ = \{ & \text{Definición de } suma \ \} \\ & 128 + suma.64.64 \end{array}$$

decimos que estamos **plegando** la definición.

2.3. Técnicas de definición de funciones

2.3.1. Análisis por casos

.

Ejemplo 2.1. La función $signo : Int \to Char$, que dado un entero retorna su signo, de la siguiente forma: retorna '+' si x es positivo, '-' si es negativo y '0' si el entero es cero.

Para empezar lo escribo informalmente, en castellano.

$$\begin{array}{lllll} signo.x \doteq & \textbf{si} & x > 0 & \textbf{da} & \texttt{'+'} \\ & \textbf{o si} & x < 0 & \textbf{da} & \texttt{'-'} \\ & \textbf{sino} & & \textbf{da} & \texttt{'0'}. \\ & \textbf{fin} & & & \end{array}$$

En Formalismo Básico (en papel) escribiremos:

$$signo : Int \to Char$$

$$\begin{array}{lllll} signo.x \doteq & (& x > 0 & \to & \texttt{'+'} \\ & \Box & x < 0 & \to & \texttt{'-'} \\ & \Box & x = 0 & \to & \texttt{'0'}. \\ &) & & & \end{array}$$

Y por último, en `haskell`hay dos opciones:

```
signo :: Int -> Char
signo x = if       x>0  then '+'
          else if  x<0  then '-'
          else               '0'
```

o bien,

```
signo :: Int -> Char
signo x | x>0 = '+'
    | x<0 = '-'
    | x==0 = '0'
```

En este caso, hay que dejar los espacios en blanco que aparecen en las últimas dos líneas para que funcione.

Comentario. Esta función *signo* es distinta de la del Práctico, que corresponde a la función `signum` de `haskell`.

2.4. Patrones

Una herramienta muy poderosa en programación funcional es la utilización de *patrones*.

Un **patrón** es la forma que tiene el argumento de una función. La utilidad de los patrones radica en que podemos estipular qué forma tienen los argumentos. Por ejemplo, para la función

$$\begin{array}{l} f : (Int, Int) \to Int \\ f.(x, y) \doteq x * y \end{array} \tag{2}$$

que toma un par de enteros y devuelve la suma de sus componentes, el patrón es $\boldsymbol{(x, y)}$.

Comentario. La función f **no es la misma** que la función *multiplicar* de la Guía 1. Aparentemente hace lo mismo, pero su tipo es distinto. *multiplicar* tiene dos argumentos, y f tiene uno que es un par (tupla de dos componentes).

Cada vez que le demos algo de comer a f, deberemos "emparejarlo" (*match*, en inglés) con el patrón antes de aplicar la función. Por ejemplo, si queremos calcular $f.(4*2,3)$, debemos entender cómo se corresponde el argumento $(4*2,3)$ con el patrón $(\boldsymbol{x},\boldsymbol{y})$:

$$\begin{array}{ccc} (& 4*2 & ,3 \quad) \\ & \downarrow & \downarrow \\ (\boldsymbol{} & \boldsymbol{x} & \boldsymbol{,y} \quad). \end{array}$$

Entonces, a la variable x le corresponde el valor $4*2$ y a y le corresponde 3. Entonces podemos desplegar la definición de f para obtener el resultado:

$$\begin{array}{ll} & f.\underline{(4*2,3)} \\ = \{ & \text{Definición de } f \text{ (de acuerdo al patrón, } x = 4*2 \text{ e } y = 3) \ \} \\ & 4*2*3 \\ = \{ & \text{Aritmética } \} \\ & 24. \end{array}$$

Veamos un ejemplo ligeramente distinto. Queremos calcular $f.(4+1,3)$. Si lo hacemos de la siguiente manera, estará **mal**:

$$\begin{array}{ll} & f.\underline{(4+1,3)} \\ = \{ & \text{Definición de } f \ \} \\ & 4+1*3 \\ = \{ & \text{Aritmética } \} \\ & 7. \end{array}$$

El resultado debería haber sido 15. Lo que sucedió aquí es que siempre que emparejamos dos expresiones, conviene poner paréntesis para no introducir efectos secundarios indeseados. En este caso, la expresión que se empareja con x es $4+1$, pero para estar seguros que se considerará como un paquete cerrado, debemos ponerlo entre paréntesis:

$$\begin{array}{ll} & f.\underline{(4+1,3)} \\ = \{ & \text{Definición de } f \ \} \\ & (4+1)*3 \\ = \{ & \text{Aritmética } \} \\ & 15. \end{array}$$

Dos patrones que utilizaremos mucho son los que describen números naturales (que denominamos *Nat* y contienen al 0 como $\mathbb{N}_0$) y listas. Para los primeros, los patrones son 0 y n. Lo veamos con un ejemplo de función más.

$$\begin{aligned} & g : Nat \to Int \\ & g.0 \doteq 1 && (3) \\ & g.(n+1) \doteq n+2. && (4) \end{aligned}$$

De acuerdo a su definición, g sólo puede consumir 0 o algo que se corresponda con el patrón $(n+1)$.

Ejercicio 2.2. Convencerse que un número natural es o bien 0 o se puede emparejar con el patrón $n+1$.

Supongamos que queremos calcular $g.6$. No podemos aplicar la definición de g a esta expresión, porque no le estamos dando de comer ni 0 ni una expresión de la forma $(n+1)$. Pero hacer un poco de aritmética antes y luego podremos aplicar el caso (4) de la definición de g:

$$
\begin{array}{ll}
 & g.\underline{6} \\
= \{ & \text{Aritmética } \} \\
 & g.(5+1) \\
= \{ & \text{Definición de } g \ \} \\
 & 5+2 \\
= \{ & \text{Aritmética } \} \\
 & 7.
\end{array}
$$

En el medio de la prueba, al emparejar la expresión $(5+1)$ con el patrón $(n+1)$ deducimos que $n = 5$. Luego, al desplegar la definición de g pasamos al término derecho de la ecuación (4) obteniendo $5+2$.

Ejercicio 2.3. Deducir qué hace la función g.

En la Sección 2.6 analizaremos el caso de las listas, para las cuales también hay dos patrones paradigmáticos.

2.4.1. Modularización (composición)

Descomponer un problema en partes.

Trabajo en clase

1. Definir la función $esBisiesto : Num \to Bool$, que indica si un año es bisiesto. Un año es bisiesto si es divisible por 400 o es divisible por 4 pero no es divisible por 100. (usar mód).

2. Definir la función $max3 : Num \to Num \to Num \to Num$, que dados tres números devuelve el mayor de los tres (usar máx).

Tarea

Entregar por escrito:

1. $mayor3 : (Int, Int, Int) \to (Bool, Bool, Bool)$, que dada una una terna de enteros devuelve una terna de valores booleanos que indica si cada uno de los enteros es mayor que 3.

   Por ejemplo: $mayor3.(1,4,3) = (False, True, False)$ y $mayor3.(5,1984,6) = (True, True, True)$.

2. Definir la función $dispersion : Num \to Num \to Num \to Num$, que toma los tres valores y devuelve la diferencia entre el más alto y el más bajo. ($Ayuda$: usar $max3$ y $min3$. De esa forma se puede definir $dispersion$ sin hace análisis por casos.)

2.5. Patrones y Recursión

El siguiente paso es utilizar múltiples patrones para definir una función de manera *recursiva*. Esto quiere decir que, a diferencia de los ejemplos anteriores, el nuevo símbolo de función introducido va a aparecer de ambos lados del signo $\doteq$.

Damos como ejemplo una "misteriosa" función h.

$$h : Int \to Int$$
$$h.0 \doteq 0 \tag{5}$$
$$h.(n+1) \doteq 1 + 2 * n + h.n \tag{6}$$

Observemos cómo h aparece de ambos lados de la Ecuación (6). Sin embargo, nos daremos cuenta con ejemplos que esto no conlleva ningún problema, siempre que $h.x$ con x un entero mayor o igual a 0.

Como dijimos en la Sección 2.2, toda (línea de) definición introduce una ecuación válida nueva. Ahora podemos calcular los valores para h.

Teorema 2.4 ("0")**.** $h.0 = 0$.

Este "Teorema 0" es trivial, porque es exactamente la Ecuación (5). Nosotros escribimos su prueba de la siguiente manera.

$$\begin{array}{ll} & h.0 \\ = \{ & \text{Definición de } h \ \} \\ & 0 \end{array}$$

Esto no sólo es el "cálculo" de $h.0$, sino que también es a la vez la *demostración* de que $h.0 = 0$. Se refuerza entonces lo que dijimos al principio: un objetivo de esta materia es "ampliar nuestra capacidad de cálculo", especialmente ampliando los tipos de objetos sobre los cuales podemos "calcular". Por ejemplo, acabamos de calcular que el valor de la expresión $h.0 = 0$ de tipo *Bool* es *True*. Demostrar teoremas es esencialmente calcular con booleanos.

Prosigamos descubriendo qué hace h, calculando $h.1$:

$$\begin{array}{ll} & h.\underline{1} \\ = \{ & \text{Aritmética} \ \} \\ & h.(0+1) \\ = \{ & \text{Definición de } h \ \} \\ & 1 + 2 * 0 + \underline{h.0} \\ = \{ & \text{Teorema "0"} \ \} \\ & 1 + 2 * 0 + 0 \\ = \{ & \text{Aritmética} \ \} \\ & 1. \end{array}$$

Hemos demostrado el siguiente

Teorema 2.5 ("1")**.** $h.1 = 1$.

¿Cómo es el Teorema "n"?

Ejercicio 2.6. Calcular $h.2$, $h.3$ y $h.4$. ¿Cuánto vale, en general $h.n$?

2.6. Listas

Uno de los tipos más importantes en Haskell son las *listas*. Los tipos de listas se indican poniendo entre corchetes ("[" y "]") otro tipo, que será el tipo de los elementos de la lista. Así, $[Int]$ es el tipo de todas las listas de enteros. Las propiedades básicas de las listas son dos.

1. Toda lista se construye a partir de la lista vacía $[\,]$, agregando elementos por la izquierda:

$$[2,3] = 2 \triangleright \underline{[3]} = 2 \triangleright (3 \triangleright [\,]). \tag{7}$$

2. Los elementos de una lista deben ser del **mismo tipo**.

El triángulo $\triangleright$ se lee "seguido de" y junto con $[\,]$ son los **constructores de listas**, porque con ellos se arma cualquier lista. Para simplificar podemos pensar que el tipo del triángulo es

$$(\triangleright) : A \to [A] \to [A],$$

que significa que toma un "elemento" (de algún tipo A) y una lista (de tipo $[A]$), y le agrega el elemento, obteniendo una lista más larga de tipo $[A]$.

Ejemplo 2.7.
1. $[x, y + z]$ (de tipo $[Num]$, puesto que el resultado de una suma es de tipo Num);
2. $[True, p]$ (de tipo $[Bool]$);
3. [`"hola"`, `"chau"`] (de tipo $[String]$).

Observando las Ecuaciones (7), podemos darnos cuenta que toda lista es o bien la lista vacía $[\,]$, o resulta de agregar un elemento a otra lista. (De hecho, el elemento agregado es el primero de la lista original).

Un ejemplo de función que toma listas es *head*:

$$\begin{aligned} &head : [A] \to [A] \\ &head.(x \triangleright xs) \doteq x \end{aligned} \tag{8}$$

Ejercicio 2.8. Calcular $head.[1,2]$, $head.$[`"hola"`, `"chau"`] y $head.[[\,], [1,2]]$.

Hagamos parte del primer ejemplo como ayuda. Como está escrito, el argumento $[1,2]$ no es digerible por la función *head*. Para que lo pueda consumir, hay que ponerlo de acuerdo al patrón $(x \triangleright xs)$.

$$\begin{aligned} & head.\underline{[1,2]} \\ = \{ & \text{ Definición de lista } \} \\ & head.(1 \triangleright [2]) \\ = \{ & \text{ Definición de } head \ \} \\ & 1. \end{aligned}$$

Para poder entender el último paso, hay que tomar conciencia quién es el "x" y quién el "xs" en la expresión $(1 \triangleright [2])$. Una vez que sabemos eso, podemos emparejar con el patrón $(x \triangleright xs)$ (y el resultado de *head* será el "x").

Prosiguiendo ahora con recursión, usaremos ambos patrones en una definición.

Ejercicio 2.9. Sea la siguiente función

$$\begin{aligned} &uno : [A] \to [Int] \\ &uno.[\,] \doteq [\,] && (9) \\ &uno.(x \triangleright xs) \doteq 1 \triangleright uno.xs. && (10) \end{aligned}$$

Calcular $uno.[3,6,7]$.

En la definición de *uno*, la línea (9) se llama **caso baso**, y la segunda (donde aparece *uno* de ambos lados), **caso inductivo**.

Hacemos los primeros pasos como ayuda:

$$
\begin{array}{ll}
 & \mathit{uno}.\underline{[3,6,7]} \\
= \{ & \text{Definición de lista } \} \\
 & \mathit{uno}.(3 \triangleright [6,7]) \\
= \{ & \text{Definición de } \mathit{uno} \ \} \\
 & 1 \triangleright \mathit{uno}.[6,7],
\end{array}
$$

etcétera. La función *uno* toma cada elemento de la lista y lo transforma en algo distinto. Las funciones de lista que trabajan así se llaman **MAP** (o de "aplicación").

Una función de listas que se comporta distinto es la que suma los elementos de una lista de números. Se llama *sum*. Por suerte, todos sabemos cómo se suma una lista de números: $\mathit{sum}.[2,1,3] = 2+1+3 = 6$ (si no lo sabemos a esto, estamos en problemas). Las funciones como *sum* que repiten una operación con todos los elementos de la lista, se llaman **FOLD**. El objetivo ahora es *encontrar* una definición recursiva para *sum*, como la que tenemos de *uno*:

$$
\begin{aligned}
&\mathit{sum} : [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sum}.[\,] \doteq ??? && (11)\\
&\mathit{sum}.(x \triangleright xs) \doteq ???. && (12)
\end{aligned}
$$

Supongamos que queremos calcular como en el principio del Ejercicio 2.9. Tendríamos algo como lo siguiente

$$
\begin{array}{ll}
 & \mathit{sum}.\underline{[2,1,3]} \\
= \{ & \text{Definición de lista } \} \\
 & \mathit{sum}.(2 \triangleright [1,3]) \\
= \{ & \text{Definición de } \mathit{sum} \ \} \\
 & (\text{algo con 2 y } \mathit{sum}.[1,3])
\end{array}
$$

Pero nosotros ya sabemos cómo queremos que se comporte *sum*: $\mathit{sum}.[2,1,3] = 6$ y $\mathit{sum}.[1,3] = 4$. Entonces tenemos lo siguiente:

$$
\begin{array}{ll}
 & 6 \\
= \{ & \text{valor de } \mathit{sum}.[2,1,3] \ \} \\
 & \mathit{sum}.\underline{[2,1,3]} \\
= \{ & \text{Definición de lista } \} \\
 & \mathit{sum}.(2 \triangleright [1,3]) \\
= \{ & \text{Definición de } \mathit{sum} \ \} \\
 & (\text{algo con 2 y } \mathit{sum}.[1,3])
\end{array}
$$

¿Cómo obtenemos 6 a partir de 2 y la suma del resto de los elementos, $\mathit{sum}.[1,3] = 4$? Simple, sumando. Luego, en este ejemplo particular, tenemos

$$
\begin{array}{ll}
 & 6 \\
= \{ & \text{valor de } \mathit{sum}.[2,1,3] \ \} \\
 & \mathit{sum}.\underline{[2,1,3]} \\
= \{ & \text{Definición de lista } \} \\
 & \mathit{sum}.(2 \triangleright [1,3]) \\
= \{ & \text{Definición de } \mathit{sum} \ (*) \ \} \\
 & 2 + \mathit{sum}.[1,3] \\
= \{ & \text{valor de } \mathit{sum}.[1,3] \ \} \\
 & 2+4
\end{array}
$$

El paso clave, indicado con (*),

$$sum.(2 \triangleright [1,3]) = 2 + sum.[1,3]$$

nos permite deducir qué hace *sum* cuando le damos de comer el patrón $(x \triangleright xs)$:

$$sum.(x \triangleright xs) \doteq x + sum.xs.$$

Ésta es la parte inductiva de la definición de *sum*. Para establecer el caso base, sigamos aplicando esta definición en nuestro ejemplo, calculando $sum.[1,3]$:

$$\begin{array}{ll} & sum.\underline{[1,3]} \\ = \{ & \text{Definición de lista } \} \\ & sum.(1 \triangleright [3]) \\ = \{ & \text{Definición de } sum \ \} \\ & 1 + sum.\underline{[3]} \\ = \{ & \text{Definición de lista } \} \\ & 1 + \underline{sum.(3 \triangleright [\,])} \\ = \{ & \text{Definición de } sum \ (*) \ \} \\ & 1 + 4 + sum.[\,] \end{array}$$

Ejercicio 2.10. Deducir a partir de este ejemplo cuánto debe valer $sum.[\,]$.

Ejercicio 2.11. Definir por recursión la función $duplica : [Int] \to [Int]$, que multiplica por 2 todos los elementos de una lista de enteros.

La última clase de funciones simples de listas son las **FILTER**. Estas funciones seleccionan los elementos de la lista que cumplen con algún criterio. En nuestro caso, un criterio será un **predicado**, es decir, una función que devuelve un *Bool*.

Ejercicio 2.12. Definir la función $esMultiplo2 : Int \to Bool$ que dado un entero devuelve *True* si y sólo si es par. (Ayuda: usar mód).

Ahora podemos dar un ejemplo de función FILTER (o "filtro"):

$$soloPares : [Int] \to [Int] \tag{13}$$

$$\begin{array}{lll} soloPares.[\,] \doteq [\,] & & \\ soloPares.(x \triangleright xs) \doteq & (\quad esMultiplo2.x & \to x \triangleright soloPares.xs \\ & \square \quad \neg esMultiplo2.x & \to soloPares.xs \\ &) & \end{array} \tag{14}$$

Ejercicio 2.13. Calcular $soloPares.[1,2,3]$.

Podemos resumir las características de las funciones de clases FILTER, MAP y FOLD para poder reconocerlas:

1. Una función FILTER toma una lista y devuelve otra *del mismo tipo*, y tiene la forma general:

$$\begin{array}{lll} filtro : [A] \to [A] & & \\ filtro.[\,] \doteq [\,] & & \\ filtro.(x \triangleright xs) \doteq & (\quad condicion.x & \to x \triangleright filtro.xs \\ & \square \quad \neg condicion.x & \to filtro.xs \\ &) & \end{array}$$

   De hecho, el resultado de una FILTER es una lista cuyos elementos estaban en la lista original, y son exactamente los que satisfacen el predicado $condicion : A \to Bool$.

2. Una función MAP toma una lista y devuelve otra *de la misma longitud*. Tiene la forma general

$$\begin{aligned} &aplicar : [A] \to [B] \\ &aplicar.[\,] \doteq [\,] \\ &aplicar.(x \triangleright xs) \doteq funcion.x \triangleright (aplicar.xs) \end{aligned}$$

   A cada elemento de la lista original le aplicamos la $funcion : A \to B$.

3. Una función FOLD toma una lista (el tipo del resultado puede no ser una lista). Su forma general es

$$\begin{aligned} &operar : [A] \to B \\ &operar.[\,] \doteq b \\ &operar.(x \triangleright xs) \doteq operacion.x.(operar.xs), \end{aligned}$$

   donde todos los elementos de la lista se los combina usando la $operacion : A \to B \to B$. En el caso de *sum*, esta operación es $(+) : Num \to Num \to Num$ (es decir, $A = B = Num$).

La función cardinal o **longitud** # no es ni MAP, ni FILTER, ni FOLD.[1]

Ejercicio 2.14. Escribir a # como la composición de una función MAP con una FOLD. (Ayuda: ya las vimos a esas dos funciones).

A continuación, se incluyen todas las definiciones de los operadores de listas. Estas funciones se pueden usar libremente (en un examen por ejemplo) para definir otras funciones nuevas.

longitud

$$\#[\,] \doteq 0$$
$$\#(x \triangleright xs) \doteq 1 + \#xs$$

head (cabeza)

$$head.(x \triangleright xs) \doteq x$$

tail (cola)

$$tail.(x \triangleright xs) \doteq xs$$

concatenar

$$[\,] +\!\!+ ys \doteq ys$$
$$(x \triangleright xs) +\!\!+ ys \doteq x \triangleright (xs +\!\!+ ys)$$

pegar por la derecha

$$[\,] \triangleleft y \doteq y \triangleright [\,]$$
$$(x \triangleright xs) \triangleleft y \doteq x \triangleright (xs \triangleleft y)$$

índice

$$(x \triangleright xs)\,!\,0 \doteq x$$
$$(x \triangleright xs)\,!\,(n+1) \doteq xs\,!\,n$$

tomar

$$xs \uparrow 0 \doteq [\,]$$
$$[\,] \uparrow n \doteq [\,]$$
$$(x \triangleright xs) \uparrow (n+1) \doteq x \triangleright (xs \uparrow n)$$

[1] En realidad esto es un poco mentira, sí es una función FOLD, pero es un poco más complicado verla así.

tirar

$$(\downarrow) : [A] \to Int \to [A]$$

$$\begin{aligned} xs \downarrow 0 &\doteq xs \\ [\,] \downarrow n &\doteq [\,] \\ (x \triangleright xs) \downarrow (n+1) &\doteq xs \downarrow n \end{aligned}$$

suma

$$sum : [Num] \to Num$$

$$\begin{aligned} sum.[\,] &\doteq 0 \\ sum.(x \triangleright xs) &\doteq x + sum.xs \end{aligned}$$

Ejercicio 2.15. Calcular $[3,4] +\!\!+ [5]$ y $[3,4,5] \uparrow 2$ usando las definiciones.

Las funciones que siguen las llamo "funciones tabú", porque (en esta materia) se debe escribir su definición para poder usarlas.

map

$$map : (A \to B) \to [A] \to [B]$$

$$\begin{aligned} map.f.[\,] &\doteq [\,] \\ map.f.(x \triangleright xs) &\doteq f.x \triangleright map.f.xs \end{aligned}$$

Ejemplo: $duplica = map.((*).2)$

filter

$$filter : (A \to Bool) \to [A] \to [A]$$

$$\begin{aligned} filter.p.[\,] &\doteq [\,] \\ filter.p.(x \triangleright xs) &\doteq (\ p.x \longrightarrow x \triangleright filter.p.xs \\ &\quad \Box\ \neg p.x \longrightarrow filter.p.xs \\ &\quad) \end{aligned}$$

Ejemplo: $soloPares = filter.esMultiplo2$

foldl

$$foldl : (A \to B \to A) \to A \to [B] \to A$$

$$\begin{aligned} foldl.f.z.[\,] &\doteq z \\ foldl.f.z.(x \triangleright xs) &\doteq foldl.f.(f.z.x).xs \end{aligned}$$

Ejemplos: $sum = foldl.(+).0$
$rev = foldl.(\triangleleft).[\,]$

2.7. Trabajo en clase

Definir las siguientes funciones y evaluarlas manualmente sobre los ejemplos dados:

1. $incPrim : [(Int, Int)] \to [(Int, Int)]$, que dada una lista de pares de enteros, le suma 1 al primer número de cada par.
   Ejemplos: $incPrim.[(20,5),(50,9)] = [(21,5),(51,9)]$, $incPrim.[(4,11),(3,0)] = [(5,11),(4,0)]$.

2. $expandir : String \to String$, pone espacios entre cada letra de una palabra.
   Ejemplo: $expandir$.`"hola"` = `"h o l a"` (¡sin espacio al final!).

2.8. Inducción

Retomemos la función h del comienzo de la Sección 2.5. A esta altura ya sabemos que el Teorema "n" es $h.n = n^2$. Para demostrar propiedades de funciones definidas recursivamente, utilizamos pruebas por **inducción**.

Comparemos variantes de las pruebas del Teorema "1" y el Teorema "2", ahora sabiendo que hay un cuadrado dando vueltas por ahí.

$$
\begin{array}{ll}
 & h.\underline{1} \\
= \{ & \text{Aritmética } \} \\
 & h.(0+1) \\
= \{ & \text{Definición de } h \ \} \\
 & 1 + 2 * 0 + \underline{h.0} \\
= \{ & \text{Teorema "0" } \} \\
 & 1 + 2 * 0 + 0^2 \\
= \{ & \text{Aritmética (binomio cuadrado) } \} \\
 & (1+0)^2 \\
= \{ & \text{Aritmética } \} \\
 & 1^2.
\end{array}
\qquad
\begin{array}{ll}
 & h.\underline{2} \\
= \{ & \text{Aritmética } \} \\
 & h.(1+1) \\
= \{ & \text{Definición de } h \ \} \\
 & 1 + 2 * 1 + \underline{h.1} \\
= \{ & \text{Teorema "1" } \} \\
 & 1 + 2 * 1 + 1^2 \\
= \{ & \text{Aritmética (binomio cuadrado) } \} \\
 & (1+1)^2 \\
= \{ & \text{Aritmética } \} \\
 & 2^2.
\end{array}
$$

Son casi iguales. Observemos, de todos modos, que la prueba del Teorema "2" requiere haber probado el Teorema "1" previamente. Por esto, demostrar el Teorema "487" (que enuncia que $h.487 = 487^2$) es posible, pero antes tendríamos que escribir las 487 pruebas de los Teoremas "n" para $n = 0, \ldots, 486$. Muy aburrido, considerando que las últimas 486 son lo mismo.

El salto de abstracción está encontrar una "forma general" de dicha demostración. Hay un numerito que va cambiando, y lo podemos llamar n. Entonces, la versión general de esta prueba queda así:

$$
\begin{array}{ll}
 & h.(n+1) \\
= \{ & \text{Definición de } h \ \} \\
 & 1 + 2 * n + \underline{h.n} \\
= \{ & \text{Teorema "}n\text{" } \} \\
 & 1 + 2 * n + n^2 \\
= \{ & \text{Aritmética (binomio cuadrado) } \} \\
 & (1+n)^2 \\
= \{ & \text{Aritmética } \} \\
 & (n+1)^2.
\end{array}
$$

Habiéndonos dado cuenta que existe esta forma general (para el Teorema "$n + 1$"), es trivial dar una *receta* para escribir la prueba del Teorema "487":

- Escribir la prueba del Teorema "0".
- Copiar la prueba general cambiando el n por todos los valores entre 0 y 486.

En lugar de hacer la (todavía inhumana) tarea del segundo ítem, podemos buscar un atajo. Esto es, suponer que ya hemos escrito la prueba del Teorema "n" y simplemente dar la del Teorema "$n + 1$". En resumidas cuentas, tenemos los siguientes pasos:

Elegir variable en la cual haremos la inducción (en este ejemplo, sólo hay una, la n).

Caso Base Probar el Teorema "0".

Caso Inductivo Suponiendo que hemos probado el Teorema "n" escribimos la prueba del Teorema "$n + 1$"

Esto es una prueba por **inducción en** *Nat*.

También se puede hacer inducción en listas. A diferencia con los naturales, donde los casos son 0 y $(n + 1)$ (o bien 1 y $(n + 1)$), los casos de listas son $[\,]$ y $(x \triangleright xs)$. Para trabajar un ejemplo, demostraremos que

$$sum.(xs +\!\!+ ys) = sum.xs + sum.ys \tag{15}$$

El esquema para hacer inducción en listas es exactamente el mismo que los naturales, pero lo describiremos en más detalle para este ejemplo. Los pasos a seguir son:

1. **Elegir una variable** para hacer la inducción.

   En el ejemplo que elegimos, los patrones relevantes aparecen siempre en la variable xs, así que elegimos esa para hacer la inducción.

2. **Caso Base**: reemplazamos la variable elegida por $[\,]$ y probamos lo que obtenemos:

   $$sum.([\,] +\!\!+ ys) = sum.[\,] + sum.ys. \tag{16}$$

3. **Caso inductivo**: Copiamos el teorema tal como venía (Ecuación (15)), y le llamamos **Hipótesis Inductiva** (HI):

   $$\text{(HI)} \qquad sum.(xs +\!\!+ ys) = sum.xs + sum.ys,$$

   reemplazamos ahora $(x \triangleright xs)$ en lugar de xs en todos lados:

   $$sum.((x \triangleright xs) +\!\!+ ys) = sum.(x \triangleright xs) + sum.ys \tag{17}$$

   y demuestramos lo que obtuvimos usando las definiciones y la HI.

Probemos el caso base.

Ejercicio 2.16. Escribir las justificaciones que faltan en la demostración que sigue, subrayando dónde se aplican los cambios.

$$\begin{array}{ll} & sum.([\,] +\!\!+ ys) \\ = \{ & \qquad\qquad \} \\ & sum.ys \\ = \{ & \qquad\qquad \} \\ & 0 + sum.ys \\ = \{ & \qquad\qquad \} \\ & sum.[\,] + sum.ys \end{array}$$

Para el caso inductivo, debemos probar la igualdad (14) usando eventualmente la HI. Para hacerlo, tenemos tres caminos básicos:

1. salir del lado izquierdo $(sum.((x \triangleright xs) +\!\!+ ys))$ y llegar al derecho $(sum.(x \triangleright xs) + sum.ys)$;
2. al revés, de derecha a izquierda; o bien
3. tomar toda la expresión booleana (17) y probar que es equivalente a *True*.

Usualmente, las primeras dos maneras resultan en pruebas más cortas, pero requieren un poco más de ingenio. Las pruebas que toman todo y llegan a *True* suelen ser al revés.

Haremos un ejemplo en el que probamos un caso particular de la hipótesis inductiva, suponiendo que ya probamos el caso base. Es decir, supongamos que sabemos

$$\text{(HI)} \qquad sum.([\,] +\!\!+ ys) = sum.[\,] + sum.ys.$$

Usando esto, vamos a probar que $sum.([4] +\!\!+ ys) = sum.[4] + sum.ys$

$$
\begin{array}{ll}
 & sum.([4] +\!\!+ ys) \\
= \{ & \text{Definición de lista } \} \\
 & sum.((4 \triangleright [\,]) +\!\!+ ys) \\
= \{ & \text{Definición de } +\!\!+ \ \} \\
 & sum.(4 \triangleright ([\,] +\!\!+ ys)) \\
= \{ & \text{Definición de } sum \ \} \\
 & 4 + sum.([\,] +\!\!+ ys) \\
= \{ & \text{HI } \} \\
 & 4 + sum.[\,] + sum.ys \\
= \{ & \text{Definición de } sum \ \} \\
 & sum.(4 \triangleright [\,]) + sum.ys \\
= \{ & \text{Definición de lista } \} \\
 & sum.[4] + sum.ys
\end{array}
$$

Ejercicio 2.17. Completar la prueba por inducción de este teorema, siguiendo la receta de más arriba.

La solución está al final del apunte.

Ejercicio 2.18. Considerando la función $quitarCeros : [Num] \to [Num]$ definida de la siguiente manera

$$
\begin{array}{lcll}
quitarCeros.[\,] & \doteq & [\,] & \\
quitarCeros.(x \triangleright xs) & \doteq & (& x \neq 0 \to x \triangleright quitarCeros.xs \\
 & & \square & x = 0 \to quitarCeros.xs \\
 & &) &
\end{array}
$$

demostrá que

$$sum.(quitarCeros.xs) = sum.xs$$

2.9. Lógica Proposicional

En esta sección introduciremos muchas herramientas que nos permitirán *calcular* expresiones lógicas, fiel al objetivo planteado en la Sección 1.3.1. Por esto, el material de esta parte también se denomina **cálculo proposicional**.

Las expresiones lógicas coinciden en nuestro formalismo con las expresiones de tipo *Bool*. En particular, llamaremos **variables proposicionales** a las variables de tipo *Bool*, y es común utilizar las letras $p, q, \ldots, P, Q, \ldots$ para ellas.

Clasificaremos las expresiones booleanas en cuatro categorías. Una expresión de tipo *Bool* puede ser:

1. **válida** si es *True* para todos los valores de sus variables (puedo **demostrar** que es equivalente a *True*). Ejemplo: $2 * x = x + x$.

2. **satisfactible** si hay al menos un valor de las variables que las hace *True* (hay un **ejemplo**). Ejemplo: $x < 5$.

3. **no válida** si es *False* para algún valor de sus variables; (hay un **contraejemplo**). Ejemplo: $2 * x = 0$.

4. **no satisfactible** si es *False* para todos los valores de sus variables (puedo **demostrar** que es equivalente a *False*). Ejemplo: $x + 1 = x$.

2.9.1. Tablas de verdad

Existe un método directo pero poco práctico de decidir a cuál categoría de las anteriores pertenece una expresión booleana. Es el uso de **tablas de verdad**, que fue introducido en el cursillo de ingreso. Las utilizaremos al principio, pero después cambiaremos a un método más sofisticado.

Para escribir la tabla de verdad de la expresión $p \vee q \equiv q$, se deben poner todas las combinaciones posibles de los valores *True* y *False* para las variables (dos en este caso, p y q, que resultan en 2^2 combinaciones), y se calcula progresivamente los valores de las subexpresiones:

p	q	$p \vee q$	$p \vee q \equiv q$
True	*True*	*True*	*True*
True	*False*	*True*	*False*
False	*True*	*True*	*True*
False	*False*	*False*	*True*

Una expresión será válida si sólo tiene *True* en su columna, satisfactible si tiene al menos un *True*; y para el resto de las opciones es similar.

El problema práctico de este método es que cuando hay muchas variables, la cantidad de líneas de la tabla se hace inmanejable.

2.9.2. Demostraciones

Repasemos nuestro sistema de prueba con el siguiente ejemplo:

$$\begin{array}{ll} & 5 * (\underline{x + 3}) = 20 \\ \equiv \{ & \text{Conmutativa } + \} \\ & 5 * (3 + x) = 20 \end{array}$$

El comentario "Conmutativa +", se refiere a que este paso de demostración está justificado por la propiedad conmutativa de la suma:

$$a + b = b + a. \tag{18}$$

¿Quiénes son a y b aquí? Cualesquiera números. Por eso decimos que es una propiedad o ley de la suma: la Ecuación (18) es válida. Precisamente, al ser válida, al reemplazar a y b por cualquier expresión de tipo *Num* obtendremos algo verdadero. Si reemplazamos a por x y b por 3, obtenemos

$$x + 3 = 3 + x.$$

Luego, cada vez que veamos la expresión $x + 3$ en alguna expresión, podemos reemplazarla por $3 + x$. Esto es lo que hicimos en el paso justificado como "Conmutativa +".

Toda justificación debe utilizar propiedades válidas (siendo un caso particular las definiciones, puesto que introducen ecuaciones válidas).

2.9.3. Demostraciones con Expresiones Booleanas

Justificaremos ahora usando expresiones booleanas válidas. Un ejemplo es la *Conmutatividad de* $\equiv$:

$$(P \equiv Q) \equiv (Q \equiv P) \tag{19}$$

Ejercicio 2.19. Comprobar que (19) es válida usando tablas de verdad.

Ejemplo 2.20. Mostraremos un paso de demostración usando la Conmutatividad de $\equiv$:

$$
\begin{array}{ll}
 & \underline{r \equiv p \wedge q} \equiv r \\
\equiv \{ & \text{Conmutatividad } \equiv \} \\
 & p \wedge q \equiv r \equiv r
\end{array}
$$

Para justificar, debemos mostrar que lo subrayado es igual a la expresión que lo reemplaza. Usando que (19) es válida, podemos sustituir P por r y Q por $p \wedge q$ para obtener

$$\underline{(r \equiv p \wedge q)} \equiv (p \wedge q \equiv r).$$

Otra expresión booleana válida es la *Definición* $\Rightarrow$

$$(P \Rightarrow Q) \equiv (P \vee Q \equiv Q) \tag{20}$$

Ejemplo 2.21. Una prueba justificada con la Definición de $\Rightarrow$ es la siguiente:

$$
\begin{array}{ll}
 & q \equiv \underline{r \Rightarrow q} \equiv r \\
\equiv \{ & \text{Definición } \Rightarrow \} \\
 & q \equiv r \vee q \equiv q \equiv r
\end{array}
$$

En este caso, la equivalencia

$$\underline{(r \Rightarrow q)} \equiv (r \vee q \equiv q)$$

se obtiene de (20) sustituyendo P por r y Q por q.

2.9.4. Variantes en demostraciones con booleanos

Hasta ahora todo es muy similar a lo que hicimos con números y listas. Pero las demostraciones con **fórmulas proposicionales** (las expresiones booleanas hechas con $\equiv$, $\wedge$, $\vee$,... y variables de tipo *Bool*) permiten mucha más flexibilidad.

Asociando con $\equiv$ Podemos *agrupar* las expresiones separadas por $\equiv$ de la manera que queramos, poniendo paréntesis.

Ejemplo 2.22. Para la *Conmutatividad de* $\equiv$, tenemos las siguientes variantes:

$$
\begin{array}{c}
(P \equiv Q) \equiv (Q \equiv P) \\
P \equiv (Q \equiv Q \equiv P) \\
(P \equiv Q \equiv Q) \equiv P \\
\ldots
\end{array}
$$

Todas resultan válidas y por ende las podemos utilizar para justificar nuestras demostraciones.

Conmutando con $\equiv$ Podemos *ordenar* las expresiones separadas por $\equiv$ de la manera que queramos (y luego agrupar a placer).

Ejemplo 2.23. Para la *Definición de* $\Rightarrow$, obtenemos:

$$
\begin{array}{c}
P \Rightarrow Q \equiv P \vee Q \equiv Q \\
P \vee Q \equiv P \Rightarrow Q \equiv Q \\
Q \equiv P \Rightarrow Q \equiv P \vee Q \\
\ldots
\end{array}
$$

2.9.5. Ejercicios

1. Decidir, usando tablas de verdad, si las siguientes fórmulas proposicionales son válida o no, satisfactibles o no.

   a) p
   b) $p \equiv p$
   c) $p \equiv p \equiv p$

2. Leer los axiomas, poniendoles paréntesis de acuerdo a la precedencia.

3. ¿De cuántas maneras se puede leer la *Definición de* $\neg$? (es el axioma **A4**).

2.9.6. Axiomas y Teoremas

Fijaremos un conjunto de fórmulas proposicionales válidas, las que llamaremos **axiomas** y justificaremos nuestras pruebas usando al principio sólo ellas.

Los axiomas que usaremos figuran en una lista que denominamos "Digesto", en la web de la materia.

Definimos intuitivamente (de manera "recursiva") la noción de *teorema*.

Definición 2.24. Un **teorema** es:

1. un axioma; o sino

2. una fórmula proposicional equivalente a un axioma o a un teorema ya demostrado.

Una versión mucho más detallada y formal de esta definición la pueden encontrar en el libro "Cálculo de Programas", Capítulo 3, pp.15-27.

En la práctica, admitiremos un par de reglas extra (que, de todos modos, son correctas desde el punto de vista del libro).

1. Todo teorema es equivalente a *True*.

2. Si pruebo que algo es equivalente a *True*, entonces es un teorema.

3. Si salgo de una fórmula E y llego a otra F justificando con teoremas, entonces $E \equiv F$ es un teorema.

A modo de ilustración, en el Ejemplo 2.21 probamos que

$$q \equiv r \Rightarrow q \equiv r \equiv r \vee q \equiv r$$

es un teorema.

2.9.7. Ejercicios

1. ¿Qué teorema demostramos en el Ejemplo 2.20?

2. Completar los espacios en blanco, justificando con los axiomas:

$$\begin{array}{ll} & \neg p \equiv q \vee r \\ \equiv & \{ \quad\quad\quad\quad\quad\quad\quad\quad \} \\ & \neg(p \equiv q \vee r) \\ \equiv & \{ \quad\quad\quad\quad\quad\quad\quad\quad \} \\ & p \not\equiv q \vee r \end{array}$$

3. Entender el ejemplo en el Ejercicio 6 del Práctico 3.

2.9.8. Estrategias Básicas de Prueba

Si quiero mostrar que una expresión de la forma $E \equiv F$ es un teorema, tengo tres opciones:

1. Salir de E y llegar a F con las reglas.
2. Tomar todo y llegar a un teorema (por ejemplo, cualquier axioma o *True*); o bien
3. Salir de E por un lado y de F por otro y llegar en ambos casos a la misma cosa.

Generalmente la primera opción da como resultado pruebas más cortas, y las otras requieren menos ingenio.

2.9.9. Estrategia "a lo bestia"

Una vez que decidimos cuál de los tres caminos anteriores tomaremos para probar un teorema, podemos considerar un esquema en tres pasos: **Eliminar conectivos, Distribuir y Simplificar**.

1) Eliminar conectivos Podemos ver la mayoría de los axiomas y algunos teoremas como definiciones de conectivos ($\not\equiv, \wedge, \Rightarrow, \Leftarrow$) en términos de los otros:

$$\begin{aligned} P \not\equiv Q &\equiv \neg(P \equiv Q) && (\textit{Def. } \not\equiv) \\ P \wedge Q &\equiv P \equiv Q \equiv P \vee Q && (\textit{R. Dorada}) \\ P \Rightarrow Q &\equiv P \vee Q \equiv Q && (\textit{Def. } \Rightarrow) \\ P \Leftarrow Q &\equiv P \vee Q \equiv P && (\textit{Def. } \Leftarrow) \end{aligned}$$

Entonces, el primer paso consiste en "desplegar" estas definiciones de manera que obtengamos una expresión en la que sólo aparezcan $\equiv$, $\neg$ y $\vee$.

2) Distribuir Usando los siguientes axiomas, puedo distribuir negaciones y disyunciones dentro de $\equiv$:

$$\begin{aligned} \neg(P \equiv Q) &\equiv \neg P \equiv Q && (\textit{Def. } \neg) \\ P \vee (Q \equiv R) &\equiv (P \vee Q) \equiv (P \vee R) && (\textit{Distr. } \vee \textit{ y } \equiv) \end{aligned}$$

3) Simplificar Por último, usando los teoremas siguientes, podemos hacer varias simplificaciones:

$$\begin{aligned} (P \equiv \mathit{True}) &\equiv P && (\textit{Neutro. } \equiv) \\ (P \equiv P) &\equiv \mathit{True} && (\textit{Reflex. } \equiv) \\ (P \equiv Q \equiv Q) &\equiv P && (\textit{Conmut. } \equiv) \\ (P \vee P) &\equiv P && (\textit{Idemp. } \vee) \\ (P \vee \neg P) &\equiv \mathit{True} && (\textit{Terc. Excl.}) \\ (\neg\neg P) &\equiv P && (\textit{Doble } \neg) \\ (P \vee \mathit{True}) &\equiv \mathit{True} && (\textit{Abs. } \vee) \\ (P \vee \mathit{False}) &\equiv P && (\textit{Neutro } \vee) \\ (P \vee \neg Q) &\equiv P \vee Q \equiv P && (\textit{Teo. } (*)) \end{aligned}$$

En este caso, reemplazo la expresión entre paréntesis por el resto.