Consejo de Formación en Educación

Departamento de Matemática

Dirección de Formación y Perfeccionamiento Docente

Instituto Normal de Enseñanza Técnica

Profesorado de Ciencia de la Computación.

Matemática I

Patricia Echenique
Saùl Tenenbaum
Paula Echenique

2017

9 789974 857704

Matemática

Patricia Echenique

Saúl Tenenbaum

Paula Echenique

Texto preparado especialmente para el curso de Matemática I del Profesorado de Informática.

Montevideo, Uruguay.

Marzo, 2017

Introducción

¿Por qué incluir un lenguaje de programación en un curso de matemática?
¿Por qué un lenguaje de programación funcional? ¿Por qué Haskell?

"Hay una razón fundamental para el estudio integrado de matemática y programación: los principios y técnicas para la resolución de problemas en una y otra disciplina, se aplican en cualquiera de ellas. Esto sugiere que existe un conjunto de ideas que subyace y establece un punto de contacto entre el comportamiento de los programas y el pensamiento matemático, y que influencian el modo de pensar humano cuando se trata de resolver un problema utilizando un computador. Este conjunto de ideas constituye un proceso de pensamiento llamado ***resolución algorítmica de problemas.***

En todas las épocas, las nuevas demandas que impone la tecnología, condicionan el desarrollo de viejas teorías, que superándose a sí mismas, dan lugar a nuevos y profundos problemas y exigen nuevas formas e ideas para lograr resolverlos. En las sociedades antiguas las condiciones necesarias para pasar a la matemática superior, no existían ni podían existir; las circunstancias adecuadas surgieron con el desarrollo de las ciencias naturales, condicionado a su vez por los avances de la tecnología en los siglos XVI y XVII. El componente educativo es parte vital de este proceso; sin él no hay transmisión de conocimiento, ni posibilidades de supervivencia. El sistema educativo debe acompañar y adecuarse permanentemente al avance teórico y sus aplicaciones en los diferentes campos científicos y tecnológicos. En los últimos años, el desarrollo tecnológico ha puesto a nuestra disposición herramientas que permiten realizar cómputos (cálculos) en forma automática, esto es, nos permiten aplicar distintas instancias de un algoritmo para resolver diferentes problemas. Esto no significa de ninguna manera que las nuevas herramientas simplifican la tarea ya sea del docente o del estudiante, sino que muy por el contrario nos enfrentan a un desafío sumamente exigente: el de investigar cómo enseñar y aprender a resolver nuevos problemas. Por ejemplo, el algoritmo de unión de conjuntos, se enseña a través de su aplicación a conjuntos particulares. Contando con un software que realice los cómputos necesarios para obtener la unión de conjuntos, pierde sentido que los estudiantes continúen realizando la misma tarea. Lo interesante es que el estudiante estudie ***el algoritmo genérico representado*** por la operación unión y aprenda a construirlo mediante un programa. Podemos establecer un paralelismo entre esta situación y el uso de las calculadoras en los cursos de la enseñanza primaria y media: son herramientas que

no enseñan a sumar o a dividir, pero que resultan muy poderosas para sumar y/o dividir, ***cuando el objeto de estudio es la esencia de la aritmética y no la herramienta en sí.*** Análogamente, tener un computador para ejecutar algoritmos, o para mostrar como esos algoritmos funcionan, es una cosa, ser capaz de enfrentarse con la esencia de esos conceptos, es otra. Tener clara esta distinción es especialmente importante hoy día dado que el énfasis exagerado en las herramientas informáticas puede convertirlas en objeto de estudio en sí mismas (plan ceibal).

La noción de algoritmo está presente en cursos tradicionales tanto de matemática como de computación; por ejemplo, se estudian el algoritmo de Euclides, el algoritmo de Ruffini, el algoritmo de Dijkstra, etc. En los cursos tradicionales de matemática se estudian funciones sin relacionarlas con el concepto de algoritmo y mucho menos con el concepto de programa, el cual está restringido a los cursos de computación. En los cursos modernos de matemática, que integran un lenguaje de programación, el concepto de algoritmo se estudia como función y como programa y los estudiantes aprenden el concepto matemático a través de su construcción en la computadora. Esto representa un salto cualitativo importante desde el punto de vista del aprendizaje y sus aplicaciones: el estudiante utiliza un manipulador simbólico (el computador) en un contexto conceptual apropiado (un curso de matemática).

Los lenguajes funcionales han sido diseñados teniendo en cuenta el acercamiento entre matemática y programación, lo cual hace que la sintaxis del lenguaje sea muy similar a la del formalismo matemático y que la propiedad de transparencia referencial de las expresiones matemáticas sea conservada en las expresiones del lenguaje de programación. Esto los hace muy adecuados para su introducción en cursos de matemática, ya que no es necesario dedicar demasiado tiempo al aprendizaje de una sintaxis muy distinta a la del lenguaje matemático.
Los lenguajes funcionales modernos se enmarcan en dos escuelas diferenciadas por la implementación de las estrategias de evaluación de expresiones: estricta y perezosa. ***Haskell*** pertenece a la segunda, que permite entre otras cosas, trabajar con objetos infinitos, lo que lo convierte en un lenguaje excelente para el trabajo con Matemática Discreta. Haskell sintetiza además las funcionalidades de los distintos lenguajes funcionales en su evolución hasta finales de la década de los 80 en que la comunidad de programación funcional decidió su creación y desarrollo, que no ha cesado desde entonces, convirtiéndolo en un lenguaje funcional de uso extendido tanto en la industria como en el ámbito educativo".

Dra. Sylvia Da Rosa

Sylvia Da Rosa es Doctora en Informática por la UDELAR, con una Maestría en la Universidad de Gothenburg, Suecia, y título de grado también en Suecia. Es Docente de la Facultad de Ingeniería, e investigadora grado 3 en el PEDECIBA. Ha publicado numerosos artículos de investigación académica, manteniendo contactos con numerosas universidades del exterior.

Para Sylvia Da Rosa, Cristina Ochoviet , toda la gente del InCo, y muy especialmente a Ingrid Hack, muchas gracias.

Los autores

Para Ignacio, Carolina, Nicolás, Fabiana y Alicia.

Saúl.

ÍNDICE ANALÍTICO

ISBN 978-9974-8577-0-4

Conjuntos

Conceptos primitivos: CONJUNTO, ELEMENTO, PERTENECE.

Pertenecer- Elemento

Sea el conjunto de los ríos del Uruguay. El Río Negro es un río del Uruguay. Entonces, este río es un **elemento** del conjunto "Ríos del Uruguay". Se dice también: el Río Negro **pertenece** al conjunto "Ríos del Uruguay". Se abrevia con el símbolo $\in$

Si representamos un cierto objeto (elemento) con la letra **x** y un conjunto con la letra **A**, diremos que: **x** es un elemento de **A** sí y sólo sí **x** pertenece a **A**

Escribimos: $\mathbf{x} \in \mathbf{A}$, para indicar la proposición: "x pertenece a A"

Escribimos: $\mathbf{y} \notin \mathbf{A}$, para indicar la proposición "y no pertenece a A"

Ejemplo: $\mathbf{H} = \{5, 6, 9, 14\}$ $5 \in \mathbf{H}$, $\mathbf{7} \notin \mathbf{H}$

Determinación de los conjuntos

Un conjunto está bien determinado si se sabe exactamente cuáles son los elementos que pertenecen a él y cuáles no.

Se puede determinar un conjunto:

1º) **Por extensión** (o por enumeración). Esta manera de determinar un conjunto consiste en nombrar cada uno de sus elementos.

2º) **Por comprensión**. Esta forma consiste en indicar la característica o propiedad común a todos los elementos del conjunto.

3º) Hay otra manera de determinar o definir un conjunto: **definición de un conjunto por inducción**. Lo veremos más adelante.

IGUALDAD DE CONJUNTOS

Definición: Dos conjuntos son iguales si y solo si tienen los mismos elementos.

La notación correspondiente es: A=B, y significa el cumplimiento de dos condiciones:

i) Todo elemento de A es elemento de B
ii) Todo elemento de B es elemento de A.

Esto también se puede expresar así: si a $\in$ A entonces a $\in$ B y si b $\in$ B entonces b $\in$ A.

Para abreviar se usan los siguientes símbolos:

$\Rightarrow$ Significa "entonces" o "implica"

$\forall$ Significa "para todo"

$\Leftrightarrow$ Significa "si y solo si"

La definición anterior de igualdad, entonces, puede escribirse en términos simbólicos del modo siguiente: $A = B \leftrightarrow (\forall a, a \in A \rightarrow a \in B) \wedge (\forall b, b \in B \rightarrow b \in A)$

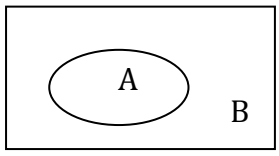
Propiedades básicas de la igualdad

1) Reflexiva: $A = A$
2) Simétrica : Si $A = B \Rightarrow B = A$
3) Transitiva: $Si \quad A = B \quad y \quad B = C \Rightarrow A = C$

INCLUSIÓN DE CONJUNTOS

Definición: Un conjunto A está incluido ampliamente en un conjunto B, si todo elemento de A es también elemento de B.

$$A \subseteq B \leftrightarrow (\forall x, x \in A \rightarrow x \in B)$$



Decimos que A es un subconjunto de B.

Observación: Ahora podríamos definir la igualdad de conjuntos de la siguiente manera

$$A = B \Leftrightarrow \left(A \subseteq B \wedge B \subseteq A \right)$$

Definición: Un conjunto A está incluido estrictamente en un conjunto B, si todo elemento de A es también elemento de B, pero hay elementos de B que no pertenecen a A.

$$A \subset B \Leftrightarrow A \subseteq B \; y \; A \neq B$$

Para afirmar que un conjunto A no está incluido en otro B (que un conjunto no es subconjunto de otro) alcanza con encontrar algún elemento del conjunto A que no pertenezca al conjunto B.

$$A \not\subset B \quad equivale \; a \; (\exists x)\left(x \in A \wedge x \notin B \right)$$

Observación: Todos conjunto es subconjunto de sí mismo. $A \subseteq A$

Propiedades básicas de la inclusión

Sean A, B y C tres conjuntos cualesquiera

1) Reflexiva: $A \subseteq A$

2) Transitiva: $Si \quad A \subseteq B \ y \ B \subseteq C \ \Rightarrow A \subseteq C$

3) Igualdad: $A \subseteq B \ \ y \ \ B \subseteq A \Leftrightarrow \ A = B$

Conjunto vacío

Un conjunto que no tiene elementos se llama conjunto vacío, y lo anotamos $\{\ \}\ o\ \varnothing$.

Por ejemplo, el conjunto de los triángulos con 4 lados no tiene elementos.

Observación: Cualquiera sea el conjunto A, se cumple que $\varnothing \subseteq A$

Conjunto referencial o universal

Llamaremos conjunto universal o referencial y lo anotaremos en general con la letra U, al conjunto que contiene al o los elementos con que nos encontramos trabajando.

Por ejemplo, si estamos trabajando en el sistema decimal con los dígitos, nuestro conjunto U podrían ser todos los dígitos, del 0 al 9.

Si estamos hablando de estudiantes, U podría ser el conjunto de los estudiantes de una clase.

Conjunto de Partes de un conjunto o conjunto potencia

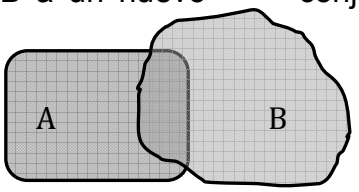
Definición: Dado un conjunto A, se llama conjunto de partes de A o conjunto potencia de A al conjunto cuyos elementos son todos los subconjuntos de A.

$$\wp(A) = \{X / X \subseteq A\}$$

OPERACIONES ENTRE CONJUNTOS

Unión de conjuntos

Definición: Sean A y B dos conjuntos: se llama unión de A y B a un nuevo conjunto cuyos elementos pertenecen a A o pertenecen a B.



$$A \cup B = \{x / x \in A \vee x \in B\}$$

Propiedades de la unión

1) $A \cup A = A$ (Idempotencia)

2) $A \cup B = B \cup A$ (Conmutativa)

3) $A \cup (B \cup C) = (A \cup B) \cup C$ (Asociativa)

4) $A \cup \varnothing = A$ (Existencia del elemento neutro)

5) $A \subseteq A \cup B$

6) $B \subseteq A \Leftrightarrow A \cup B = A$

Demostración de la propiedad 5)

$A \subseteq A \cup B$

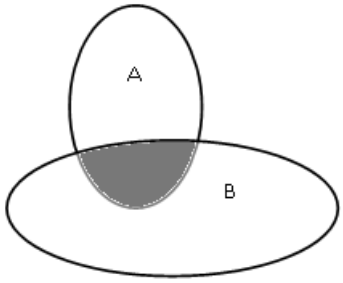
$$\forall x,\ x \in A \underbrace{\Rightarrow}_{(1)} \forall x, \left(x \in A \vee x \in B\right) \underbrace{\Rightarrow}_{(2)} \forall x, \left(x \in A \cup B\right)$$

(1) Regla lógica adición: $p \rightarrow p \vee q$

(2) Definición de la operación unión

Intersección de conjuntos

Definición: Sean A y B dos conjuntos: se llama intersecciónde A y B a un nuevo conjunto formado por los elementos que pertenecen a Ay a B.



$$A \cap B = \{x / x \in A \wedge x \in B\}$$

Propiedades de la intersección

1) $A \cap A = A$ (Idempotencia)

2) $A \cap B = B \cap A$ (Conmutativa)

3) $A \cap \left(B \cap C\right) = \left(A \cap B\right) \cap C$ (Asociativa)

4) $A \cap \varnothing = \varnothing$ (Absorción)

5) $A \cap B \subseteq A$

6) $B \subseteq A \Leftrightarrow A \cap B = B$

Demostración la propiedad 2)

$A \cap B = B \cap A$

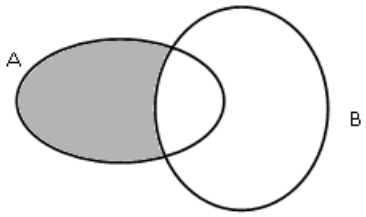
Para demostrar la igualdad demostraremos $(1)\ A \cap B \subseteq B \cap A$ y $(2)\ B \cap A \subseteq A \cap B$

(1) $\forall x,\ si\ \ x \in A \cap B \Rightarrow x \in A \wedge x \in B \Rightarrow x \in B \wedge x \in A \Rightarrow x \in B \cap A$

(2) $\forall x,\ si\ \ x \in B \cap A \Rightarrow x \in B \wedge x \in A \Rightarrow x \in A \wedge x \in B \Rightarrow x \in A \cap B$

Diferencia de conjuntos

Definición: Sean A y B dos conjuntos. Se llama diferencia de de A y B y se escribe A-B, a un nuevo conjunto que tiene por elementos los que pertenecen a A y que no pertenecen a B.



$$A - B = \{x / x \in A \ y \ x \notin B\}$$

Propiedades de la diferencia

1) $A - A = \varnothing$
2) $A - \varnothing = A$
3) $\varnothing - A = \varnothing$
4) $A - (B \cup C) = (A - B) - C$

<u>Diferencia simétrica</u>

Definición: Dados dos conjuntos A y B , llamamos "diferencia simétrica de A menos B " al conjunto que anotamos $A\Delta B$ formado por los elementos que pertenecen a uno de los dos conjuntos, y no pertenecen al otro.

$$A\Delta B = \{x / (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\}$$

<u>Complemento</u>

Definición: Sean los conjuntos A y B tales que $A \subseteq B$. Se llama complemento de A con respecto a B y se escribe $A_B^c,\ A_B^{'}\ o\ \overline{A_B}$ al conjunto formado por los elementos que pertenecen a B y que no pertenecen a A.

$$A\ {}_B^c = \{x / x \in B \wedge x \notin A\}$$

Propiedades del complemento:

1) $A \cup \overline{A} = U$
2) $\overline{U} = \varnothing$
3) $A \cap \overline{A} = \varnothing$
4) $\overline{\varnothing} = U$

Pares ordenados

Si a y b designan dos objetos matemáticos cualesquiera, el par ordenado asociado con a y b se representa mediante el símbolo (a, b) donde a y b se denominan respectivamente primer componente y segundo componente del par.

Par ordenado

Definición: $(a,b)=\{\{a,b\},a\}$

Es un conjunto que tiene dos elementos, uno de ellos es a su vez otro conjunto con dos elementos (las componentes del par), y el segundo elemento nos indica cuál de ellas es la primera componente. Los pares ordenados $(a,b)\ y\ (b,a)$ son diferentes, ya que $\{\{a,b\},a\}\neq\{\{a,b\},b\}$

Producto cartesiano

Definición: Sean A y B dos conjuntos. Se llama producto cartesiano de A por B y se escribe AxB, a un nuevo conjunto formado por todos los pares ordenados tales que el primer componente del par pertenece al conjunto A y el segundo al conjunto B.

$$A\times B=\{(a,b)/a\in A\wedge b\in B\}$$

Propiedad: Sean A y B dos conjuntos cualesquiera. Entonces $A\times B\neq\varnothing\Leftrightarrow(A\neq\varnothing\ \wedge B\neq\varnothing)$

Demostración:

$$(\Rightarrow)\ A\times B\neq\varnothing\Rightarrow\exists un\ par\ ordenado\ (x,y)tal\ que\ x\in A\wedge y\in B\Rightarrow A\neq\varnothing\wedge B\neq\varnothing$$

$$(\Leftarrow)\ A\neq\varnothing\wedge B\neq\varnothing\Rightarrow existen\ elementos\ a\ y\ b\ tales\ que\ a\in A\wedge b\in B\Rightarrow$$

$$(a,b)\in A\times B\Rightarrow A\times B\neq\varnothing$$

Todo teorema es equivalente a su contrarrecíproco:

$$(A=\varnothing\ \vee\ B=\varnothing)\Leftrightarrow A\times B=\varnothing$$

Algunas propiedades de las operaciones entre conjuntos y sus demostraciones:

(1) $A\cup A=A$ (Idempotencia)

Demostración:

$$x\in A\cup A\Leftrightarrow x\in A\vee\ x\in A(def\ \ de\ \cup)\Leftrightarrow x\in A$$

(2) $A\cap A=A$

(3) Conmutativa de la intersección

$$A \cap B = B \cap A$$

Demostraremos que: a) $A \cap B \subseteq B \cap A$

b) $B \cap A \subseteq A \cap B$

a) Demostración:

$$\forall x,\ si\ x \in (A \cap B) \underset{def\ de \cap}{\Rightarrow} x \in A \wedge x \in B \underset{conmutativa\ de \wedge}{\Rightarrow} x \in B \wedge x \in A \underset{def\ de \cap}{\Rightarrow} x \in (B \cap A)$$

b) Análogamente.

(4) Conmutativa de la unión

$$A \cup B = B \cup A$$

Demostración:

$$\forall x,\ si\ x \in (A \cup B) \underset{def\ de \cup}{\Leftrightarrow} x \in A \vee x \in B \underset{conmutativa\ de \vee}{\Leftrightarrow} x \in B \vee x \in A \underset{def\ de \cup}{\Leftrightarrow} x \in (B \cup A)$$

(5) Asociativas

i) $A \cup (B \cup C) = (A \cup B) \cup C$

Demostración:

$$\forall x,\ si\ x \in A \cup (B \cup C) \underset{def\ de \cup}{\Leftrightarrow} x \in A \vee x \in (B \cup C) \underset{def\ de \cup}{\Leftrightarrow}$$

$$x \in A \vee (x \in B \vee x \in C) \underset{asociativa\ de \vee}{\Leftrightarrow} (x \in A \vee x \in B) \vee x \in C \underset{def \cup}{\Leftrightarrow} x \in (A \cup B) \vee x \in C$$

$$\underset{def \cup}{\Leftrightarrow} x \in (A \cup B) \cup C$$

ii) $A \cap (B \cap C) = (A \cap B) \cap C$

(6) Distributivas

i) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Demostraremos (a) $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$

(b) $(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$

Demostración:

(a) Dado cualquier $x, x \in A \cap (B \cup C)\ \ es\ \ decir\ \ \ x \in A\ y\ \ x \in B \cup C \Rightarrow x \in A\ y\ \ x \in B \vee x \in C$

Entonces tenemos: $\left.\begin{cases} x \in A \ y \ \ x \in B \Rightarrow x \in A \cap B \\ \vee \\ x \in A \ \ y \ \ x \in C \Rightarrow x \in A \cap C \end{cases}\right\} \Rightarrow x \in (A \cap B) \cup (A \cap C)$

(b) $x \in (A \cap B) \cup (A \cap C) \Rightarrow \begin{cases} x \in A \cap B \Rightarrow x \in A \, y \, x \in B \Rightarrow x \in A \ y \ \ x \in B \cup C \\ \vee \\ x \in A \cap C \Rightarrow x \in A \wedge x \in C \Rightarrow x \in A \ \ y \ \ x \in B \cup C \end{cases}$

En resumen tenemos $x \in A \ y \ \ x \in B \cup C \ \Rightarrow x \in A \cap (B \cup C)$

ii) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

(7) Propiedad involutiva

$$\left(\overline{A}\right) = A$$

Demostración:

$$\forall x, x \in \left(\overline{\overline{A}}\right) \underset{\text{def de complemento}}{\Leftrightarrow} x \notin \overline{A} \underset{\text{negación}}{\Leftrightarrow} \neg\left(x \in \overline{A}\right) \underset{\text{def de complemento}}{\Leftrightarrow} \neg(x \notin A)$$

$$\underset{\text{negación}}{\Leftrightarrow} \neg\neg(x \in A) \underset{\text{doble negación}}{\Leftrightarrow} x \in A$$

(8) Leyes de De Morgan

Dados dos conjuntos A y B en un universal U, se verifica:

i) $\left(\overline{A \cup B}\right) = \overline{A} \cap \overline{B}$

Demostración:

$$\forall x, x \in \left(\overline{A \cup B}\right) \underset{\text{def complemento}}{\Leftrightarrow} x \in U \ \wedge x \notin (A \cup B) \underset{\text{negación}}{\Leftrightarrow} x \in U \ \wedge \neg\left(x \in (A \cup B)\right)$$

$$\underset{\text{def de}\cup}{\Leftrightarrow} x \in U \wedge \left(\neg(x \in A \vee x \in B)\right) \underset{\substack{\text{De Morgan para} \vee \\ (\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q)}}{\Leftrightarrow} x \in U \wedge \left(\neg(x \in A) \wedge \neg(x \in B)\right)$$

$$\underset{\text{def complemento}}{\Leftrightarrow} \left(x \in U \wedge (x \notin A \wedge x \notin B)\right) \Leftrightarrow (x \in U \wedge x \notin A) \wedge (x \in U \wedge x \notin B)$$

$$\underset{\text{def complemento}}{\Leftrightarrow} \ x \in \overline{A} \ \wedge \ x \in \overline{B} \underset{\text{def}\cap}{\Leftrightarrow} x \in \overline{A} \cap \overline{B}$$

ii) $\left(\overline{A \cap B}\right) = \overline{A} \cup \overline{B}$ Se demuestra en forma análoga a i)

Más demostraciones

(9) $A - B \subseteq A$

Demostración:

$$\forall x, x \in (A - B) \underset{def -}{\Leftrightarrow} x \in A \wedge x \notin B \underset{simplificación}{\Rightarrow} x \in A$$

(10) $Si \;\; A \subseteq B \;\; y \;\; C \subseteq D \Rightarrow (A \cup C) \subseteq (B \cap D)$

Demostración:

$$\forall x, x \in (A \cup C) \underset{def \cup}{\Leftrightarrow} x \in A \vee x \in C \underset{hipótesis}{\Rightarrow} x \in B \vee x \in D \underset{def \cup}{\Leftrightarrow} x \in (B \cup D)$$

(11) $Si \;\; A \subseteq B \;\; y \;\; C \subseteq D \Rightarrow (A \cap C) \subseteq (B \cap D)$

Análogamente a la parte anterior

(12) $A \subseteq (A \cup B)$

Demostración:

$$\forall x, x \in A \underset{adición}{\Rightarrow} x \in A \vee x \in B \underset{def \cup}{\Leftrightarrow} x \in (A \cup B)$$

(13) $(A \cap B) \subseteq A$

Demostración:

$$\forall x, x \in (A \cap B) \underset{def \cap}{\Leftrightarrow} x \in A \wedge x \in B \underset{simplificación}{\Rightarrow} x \in A$$

(14) $Si \;\; A \subseteq B \;\; \Rightarrow (A \cup B) = B$

Demostración:

$$\forall x, x \in (A \cup B) \underset{def \cup}{\Leftrightarrow} x \in A \vee x \in B \underset{hipótesis}{\Rightarrow} x \in B \vee x \in B \Leftrightarrow x \in B$$

(15) $Si \;\; A \subseteq B \Rightarrow (A \cap B) = A$

Demostración:

$Ya \;\; se \;\; demostró \;\; que \;\; A \cap B \subseteq A \qquad (1)$

$$\forall x, x \in A, como \;\; A \subseteq B \underset{hipótesis}{\Leftrightarrow} x \in A \wedge x \in B \underset{def \cap}{\Rightarrow} x \in A \cap B$$

$ahora \;\; demostramos \;\; que \;\; A \subseteq A \cap B \qquad (2)$

$De \;\; (1) \;\; y \;\; (2) \Rightarrow \;\; A \cap B = A$

(16) $A - B = A \cap \overline{B}$

Demostración:

$$\forall x, x \in (A - B) \underset{def\ diferencia}{\Leftrightarrow} x \in A \wedge x \notin B \underset{def\ complemento}{\Leftrightarrow} x \in A \wedge x \in \overline{B} \underset{def\ \cap}{\Leftrightarrow} x \in \left(A \cap \overline{B}\right)$$

(17) $A \cap (B - A) = \varnothing$

Demostración:

$$A \cap (B - A) \underset{prop\ anterior}{=} A \cap \left(B \cap \overline{A}\right) \underset{conmutativa}{=} A \cap \left(\overline{A} \cap B\right) \underset{asociativa}{=}$$

$$\left(A \cap \overline{A}\right) \cap B \underset{prop\ complemento}{=} \varnothing \cap B = \varnothing$$

(18) $A \cup (B - A) = A \cup B$

Demostración:

$$A \cup (B - A) = A \cup \left(B \cap \overline{A}\right) \underset{distributiva}{=} (A \cup B) \cap \left(A \cup \overline{A}\right) = (A \cup B) \cap U = (A \cup B)$$

(19) $A - (B \cup C) = (A - B) \cap (A - C)$

Demostración:

$$A - (B \cup C) = A \cap \overline{(B \cup C)} \underset{De\ Morgan}{=} A \cap \left(\overline{B} \cap \overline{C}\right) = A \cap A \cap \overline{B} \cap \overline{C} =$$

$$= A \cap \overline{B} \cap A \cap \overline{C} = \left(A \cap \overline{B}\right) \cap \left(A \cap \overline{C}\right) = (A - B) \cap (A - C)$$

(20) Ejercicio del parcial de julio 2010:

A, B y C tres conjuntos. Demuestra que $\overline{A \cup (B \cap C)} = \left(\overline{C} \cup \overline{B}\right) \cap \overline{A}$

Demostración:

$$\overline{A \cup (B \cap C)} \underset{De\ Morgan}{=} \overline{A} \cap \overline{(B \cap C)} \underset{De\ Morgan}{=} \overline{A} \cap \left(\overline{B} \cup \overline{C}\right) \underset{conmutativa}{=}$$

$$\left(\overline{B} \cup \overline{C}\right) \cap \overline{A} \underset{conmutativa}{=} \left(\overline{C} \cup \overline{B}\right) \cap \overline{A}$$

Lógica y matemática: un ejemplo extraído de "Lógica simbólica" de Lewis Carroll.

Examina atentamente las siguientes cinco afirmaciones:

1) Ningún gato al que le gusta el pescado es indomesticable.
2) Ningún gato sin cola jugará con un gorila.
3) A los gatos con bigotes les gusta el pescado.
4) Ningún gato domesticable tiene ojos verdes.
5) Ningún gato tiene cola a menos que tenga bigotes.

Demuestra que estas cinco afirmaciones permiten deducir que: *"Ningún gato de ojos verdes jugará con un gorila"*

Llamemos U al conjunto de todos los gatos.

Consideremos los siguientes subconjuntos de U:

A: "con ojos verdes"

B: "les gusta el pescado"

C: "con cola"

D: "indomesticable"

E: "con bigotes"

F: "deseosos de jugar con un gorila"

Tratemos de traducir a lenguaje simbólico las cinco afirmaciones anteriores:

1) Ningún gato al que le gusta el pescado es indomesticable.

$$\neg(\exists x)(x \in B \wedge x \in D) \Leftrightarrow (\forall x)(x \in B \Rightarrow x \notin D) \Rightarrow B \cap D = \varnothing$$

2) Ningún gato sin cola jugará con un gorila.

$$\neg(\exists x)(x \in \overline{C} \wedge x \in F) \Leftrightarrow (\forall x)(x \in \overline{C} \Rightarrow x \notin F) \Leftrightarrow (\forall x)(x \in \overline{C} \Rightarrow x \in \overline{F})$$

$$\Leftrightarrow \overline{C} \subseteq \overline{F} \Leftrightarrow F \subseteq C$$

3) A los gatos con bigotes les gusta el pescado.

$$(\forall x)(x \in E \Rightarrow x \in B) \Leftrightarrow E \subseteq B$$

4) Ningún gato domesticable tiene ojos verdes.

$$\neg(\exists x)\left(x \in \overline{D} \wedge x \in A\right) \Leftrightarrow (\forall x)\left(x \in \overline{D} \Rightarrow x \notin A\right)$$

$$\Leftrightarrow (\forall x)\left(x \in \overline{D} \Rightarrow x \in \overline{A}\right) \Leftrightarrow \overline{D} \subseteq \overline{A} \Leftrightarrow A \subseteq D$$

5) Ningún gato tiene cola a menos que tenga bigotes.

$$(\forall x)(x \in C \Leftrightarrow x \in E) \Leftrightarrow C = E$$

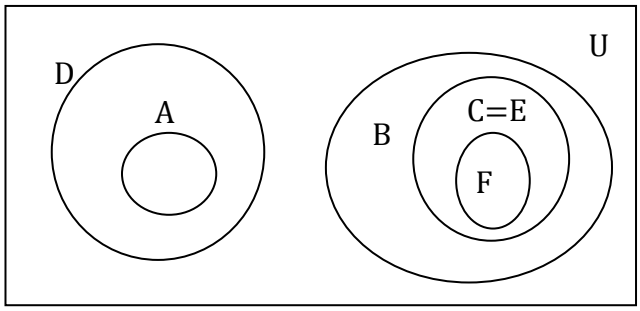
Así que tenemos:

$$(F \subseteq C \wedge C = E) \Rightarrow F \subseteq E$$

$$(F \subseteq E \wedge E \subseteq B) \Rightarrow F \subseteq B$$

Además

$$(A \subseteq D \wedge F \subseteq B) \wedge B \cap D = \varnothing \Rightarrow A \cap F = \varnothing$$



Vamos a escribir entonces las afirmaciones de otra forma, equivalente a la anterior.

1') Los gatos indomesticables no gustan del pescado. $D \cap B = \varnothing \ (si \ \ x \in D \Rightarrow x \notin B)$

2') Los gatos que no tienen cola no jugarás con un gorila. $\left(\forall x, x \in \overline{C} \Rightarrow x \in \overline{F}\right)$

3') Los gatos a los que no les gusta el pescado no tienen bigote. $(si \ x \notin B \Rightarrow x \notin E \ \ ya \ que \ \ E \subseteq B)$

4') Los gatos con ojos verdes son indomesticables. $(si \ \ x \in A \Rightarrow x \in D \ \ \ porque \ \ A \subseteq D)$

5') Los gatos que no tienen bigote no tienen cola (y recíprocamente) $(x \notin E \Leftrightarrow x \notin C \ \ ya \ \ que \ \ C = E)$

Ordenamos en forma conveniente…

(4') (1') (3') (5') (2')

Principales leyes lógicas.

En la elaboración de las siguientes leyes, se ha supuesto que p, q y r son proposiciones que pueden asumir cualquier valor de verdad; mientras que V es una proposición verdadera (tautología) y F es una proposición falsa (contradicción).

- ✓ Doble negación: $\neg\neg p \Leftrightarrow p$
- ✓ Idempotencia: $p \vee p \Leftrightarrow p$; $p \wedge p \Leftrightarrow p$
- ✓ Elemento neutro: $p \vee F \Leftrightarrow p \quad de\ la\ disyunción$; $p \wedge V \Leftrightarrow p \quad de\ la\ conjunción$
- ✓ Condiciones de negación: $p \vee \neg p \Leftrightarrow V \quad de\ la\ disyunción$; $p \wedge \neg p \Leftrightarrow F \quad de\ la\ conjunción$
- ✓ Leyes conmutativas: $p \vee q \Leftrightarrow q \vee p$; $p \wedge q \Leftrightarrow q \wedge p$
- ✓ Leyes asociativas: $p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$; $p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$
- ✓ Leyes distributivas: $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$; $p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$
- ✓ Leyes de absorción: De la conjunción respecto a la disyunción: $p \wedge (p \vee q) \Leftrightarrow p$

  De la disyunción respecto a la conjunción: $p \vee (p \wedge q) \Leftrightarrow p$
- ✓ Leyes de De Morgan: $\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$; $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$

Relaciones

Recordemos la definición de producto cartesiano: $A \times B = \{(a,b) / a \in A \wedge b \in B\}$

Definición: Sean A y B conjuntos cualesquiera.
Una relación de A en B es un subconjunto de AxB.

$\Re$ *es una relación de* A *en* $B \Leftrightarrow \Re \subseteq A \times B$

o

$\Re$ *es una relación* $A \to B \Leftrightarrow \Re \subseteq A \times B$

Observaciones:

- A recibe el nombre de DOMINIO de la relación y B CODOMINIO.
- Si $\Re : A \to B \quad y \quad (a,b) \in \Re$ diremos que a está relacionado con b y lo representaremos $a \Re b \quad o \quad \Re(a) = b$
- Habitualmente trabajaremos con relaciones de A en A (binarias) a las que llamaremos, en forma más breve, relaciones definidas en A o relaciones en A. Usualmente, en vez de escribir $\Re : A \to A$ se escribe $\Re : A$

Relación inversa:

Definición: Sea $\Re : A \to B$. Llamamos relación inversa (su notación es $\Re^{-1}$) a la relación de B en A que cumple: $\Re^{-1} : B \to A / \Re^{-1} = \{(b,a) / (a,b) \in \Re\}$

Relación complementaria:

Definición: Sea $\Re : A \to B$. Llamamos relación complementaria (su notación es $\overline{\Re}$) a la relación de A en B que cumple: $\overline{\Re} : A \to B / \overline{\Re} = \{(a,b) / (a,b) \notin \Re\}$

Veamos un ejemplo. Sea el conjunto $A = \{ 3, 4, 7 \}$ y se define la siguiente relación:

$\Re = \{ (3,4), (3,7), (4,4), (4,7), (7,7) \}$

Entonces la relación inversa es $\Re^{-1} = \{ (4,3), (7,3), (4,4), (7,4), (7,7) \}$

Y la relación complementaria es: $\overline{\Re} = \{ (3,3), (4,3), (7,3), (7,4) \}$

Observación: $\Re \cup \overline{\Re} = A \text{ x } A$

La unión de una relación y su complementaria es todo el producto cartesiano de A por A

Propiedades de relaciones de A en A

- **Propiedad reflexiva (o idéntica)**

  Una relación $\Re$ sobre un conjunto A es reflexiva si para todo $x \in A$ entonces $(x,x) \in \Re$.

  $$\forall x, x \in A, \quad (x,x) \in \Re$$

- **Propiedad antirreflexiva (o irreflexiva)**

  Una relación $\Re$ sobre un conjunto A es irreflexiva si para todo $x \in A$ se cumple $(x,x) \notin \Re$.

  $$\forall x, x \in A, (x,x) \notin \Re$$

- **Propiedad simétrica**

  Una relación $\Re$ sobre un conjunto A es simétrica si para todo $x \in A$, $y \in A$ si $(x,y) \in \Re$ entonces $(y,x) \in \Re$.

  $$\forall x, \forall y, x \in A, y \in A \quad si \quad (x,y) \in \Re \Rightarrow (y,x) \in \Re$$

- **Propiedad antisimétrica**

  Def 1: Una relación $\Re$ sobre un conjunto A es antisimétrica si para todo $x \in A$, $y \in A$ si $(x,y) \in \Re$ e $(y,x) \in \Re$ entonces x=y.

  $$\Re \subseteq A \times A \ \ es \ \ antisimétrica \ \ si \ \big((x,y) \in \Re \wedge (y,x) \in \Re\big) \ \Rightarrow x = y$$

  Otra definición de propiedad antisimétrica. Def 2:

  $$\Re \subseteq A \times A \ \ es \ \ antisimétrica \ \ si \ \big((x,y) \in \Re \wedge x \neq y\big) \ \Rightarrow \ (y,x) \notin \Re$$

- **Propiedad asimétrica**

  $$\Re \subseteq A \times A \ \ es \ \ asimétrica \ \ si \ \ (x,y) \in \Re \ \Rightarrow \ (y,x) \notin \Re$$

  Observación: las relaciones asimétricas son antirreflexivas.

- **Propiedad transitiva**

  $$\Re \subseteq A \times A \ \ es \ \ transitiva \ \ si \ \big((x,y) \in \Re \wedge (y,z) \in \Re\big) \Rightarrow (x,z) \in \Re \ \ donde \ \ x,y,z \in A$$

Ejemplos: Sean las Relaciones binarias R sobre el conjunto A= {1, 2, 4,7}

R2 = {(1,1), (2,2), (4,4), (2,7), (7,1)}

R3 = { (4,1), (1,4), (2,7), (7,2)}

R4 = { (4,1), (1,4), (2,7), (7,1)}

R5 = { (4,1), (1,7), (2,7), (7,4), (2,2) }

R6 = { (4,1), (1,7), (2,7), (7,4)}

R7 = { (4,1), (1,7), (4,7)}

- La relación R1 es reflexiva pues para todos los elementos "a" del conjunto A, esto es, el 1, 2, 4 y 7, se encuentran presentes todos los pares (a,a). Esto es, se encuentran el (1,1), (2,2), (4,4) y (7,7). Hay más pares ordenados, (2,7) y (7,1) pero eso no interesa para la propiedad reflexiva.
- La relación R2 no es reflexiva pues le falta el par (7,7).
- La relación R3 tampoco es reflexiva pues le faltan todos los pares ordenados. Es más, esta relación es entonces antirreflexiva.
- La relación R2, que tiene algunos pares, pero no los tiene todos, no es reflexiva ni irreflexiva.

Atención entonces: las propiedades reflexiva e irreflexiva no son "opuestas". Si una relación no es reflexiva, no tiene porqué ser antirreflexiva.

Las propiedades 1 y 2, reflexiva y antirreflexiva, tienen un "para todo" en el principio. Se deben cumplir **para todos** los elementos del conjunto A.

Sin embargo, las propiedad 3, 4, 5 y 6 tienen un "SI" adelante. Su definición dice entonces que **si** pasa algo, entonces **deberá suceder** otra cosa. Son definiciones condicionales.

Por ejemplo, la propiedad simétrica dice que si está el par (1,4), entonces tiene que estar el (4,1).

Vemos que la relación R3 es simétrica, pero la R4 no es simétrica.

- La relación R4 no es antisimétrica ni asimétrica. Intenta verificarlo.
- Las relaciones R5 y R6 son ambas antisimétricas.
- La relación R6 es además asimétrica.
- La relación R7 es transitiva, y además es antirreflexiva, antisimétrica y asimétrica.

Relaciones de equivalencia

Definición: Se dice que una relación $\Re$ en un conjunto A es una relación de equivalencia si es REFLEXIVA, SIMÉTRICA y TRANSITIVA.

Si dos elementos de un conjunto están relacionados por una relación de equivalencia, se dice que son *equivalentes*.

Ejemplos:

(1) Supongamos que $\Re$ es una relación en el conjunto de cadenas del alfabeto tal que $a\Re b \Leftrightarrow l(a) = l(b)$, siendo $l(x)$ la longitud de la cadena x. ¿Es $\Re$ una relación de equivalencia?

- ✓ $l(a) = l(a)$, entonces $a\Re a$ para cualquier cadena. Entonces $\Re$ es reflexiva.

- ✓ Supongamos que $a\Re b$, entonces $l(a) = l(b) \Rightarrow l(b) = l(a) \Rightarrow b\Re a \Rightarrow \Re \; es \; simétrica$
- ✓ Supongamos $a\Re b \;\; y \;\; b\Re c$ esto significa que $l(a) = l(b) \;\; y \;\; l(b) = l(c)$. Entonces $l(a) = l(c) \Rightarrow a\Re c$. Por lo tanto $\Re$ es transitiva.

  Como $\Re$ es reflexiva, simétrica y transitiva, es una relación de equivalencia.

(2) Analizar si la relación $\Re$ definida en el conjunto de los reales tal que $a\Re b \Leftrightarrow a - b \; es \; un \; número \; entero$ es una relación de equivalencia.

- ✓ $\forall a, a \in R, (a, a) \in \Re \Leftrightarrow a - a \; es \; un \; número \; entero \; (a - a = 0; 0 \in Z)$
- ✓ $a\Re b \Leftrightarrow (a-b) \in Z \Rightarrow su \; opuesto \; (-(a-b)) \in Z \Rightarrow (-(a-b)) = (-a+b) = (b-a) \in Z$
- ✓ Si $a\Re b$ y $b\Re c$

  $a\Re b \Leftrightarrow (a-b) \in Z \;\; b\Re c \Leftrightarrow (b-c) \in Z$

  $La \; suma \; de \; dos \; enteros \; es \; un \; entero \Rightarrow (a-b)+(b-c) = (a-c) \in Z \Rightarrow a\Re c$

Clases de equivalencia

Definición: Dado un conjunto A y una relación $\Re$ de equivalencia definida en A, sea $a \in A$; definimos la CLASE de a y lo anotaremos C_a o $[a]$ como el conjunto formado por todos los elementos de A que están relacionados con a.

$$C_a = [a] = \{x / x \in A, \;\; x \; \Re \; a\}$$

Conjunto cociente

Definición: Dado un conjunto A, $\Re$ una relación de equivalencia definida en A. Definimos conjunto cociente ($A/\Re$) como el conjunto formado por todas las clases de equivalencia.

$$A/\Re = \{C_x \in \wp(A),\ x \in A\}$$

Ejemplo: Sea A = { 2, 3, 4, 7 }

En dicho conjunto definimos la relación R = { (2,2), (3,3), (4,4), (7,7), (3,4), (4,7), (3,7), (4,3), (7,4), (7,3) }

Podemos ver que la relación R es reflexiva, simétrica y transitiva. Entonces R es una relación de equivalencia. El elemento 3 está relacionado con el 4 y con el 7.

[3] = {3, 4, 7 }

El elemento 2 sólo está relacionado consigo mismo.

[2] = {2 }

Entonces tenemos 2 clases de equivalencia. La clase del 2 y la clase del 3. Esta última es la misma que la clase del 4 y que la clase del 7. [3] = [4] = [7]

El conjunto cociente tiene sólo 2 clases de equivalencia. A/R = {[2], [3]}

Partición de un conjunto

Si tenemos un conjunto A no vacío, podemos formar otros conjuntos con los elementos de A de forma que ninguno de esos conjuntos sea vacío, que no tengamos elementos repetidos y que la unión de todos sea el conjunto A.

Ejemplo: Sea $A = \{1,2,3,4\}$

El conjunto $P = \{\ \{1,2\},\ \{3\},\ \{4\}\ \}$ es una partición de A.

El conjunto $Q = \{\ \{1,2,3\},\ \{4\}\ \}$ es otra partición de A.

Definición: Dado un conjunto A (no vacío) definimos PARTICIÓN de A como el conjunto $P = \{X_i\}$ tal que se cumple:

1) $P \subseteq \wp(A)$
2) $\forall i, X_i \neq \varnothing$
3) $\bigcup X_i = A$
4) Si $X_i \neq X_j \ \Rightarrow\ X_i \cap X_j = \varnothing$

Propiedad: Dado un conjunto A no vacío, $\Re$ una relación de equivalencia definida en el conjunto A. Se cumple que el conjunto cociente $A/\Re$ es una partición de A.

Vamos a repasarlo con el ejemplo que vimos en la página anterior.

A = {2, 3, 4, 7 } A/R = { [2], [3] } ¿Es éste conjunto cociente una partición de A?

Sí, porque la clase de equivalencia del 3 contiene al 3, al 4 y al 7. Estos son los elementos de A que están relacionados con el 3.

[3] = {3, 4, 7 } [2] = { 2 }

Entonces, A/R = { [2] , [3] } = { { 2 } , { 3, 4, 7 } } Esto es, una partición del conjunto A.

Relaciones de orden

Definición: Se dice que una relación $\Re$ en un conjunto A es una relación de orden (amplio) si es REFLEXIVA, ANTISIMÉTRICA y TRANSITIVA.

Definición: Se dice que una relación $\Re$ en un conjunto A es una relación de orden (estricto) si es ANTISIMÉTRICA, TRANSITIVA y NO cumple la propiedad REFLEXIVA.

Relación de orden parcial: Una relación de orden $\Re$ definida en un conjunto A se dice parcial si y solo si al menos un par de elementos de A no se relacionan entre sí (es decir: $\exists x, y \in A / (x, y) \notin \Re \wedge (y, x) \notin \Re$)

Relación de orden total: Una relación de orden $\Re$ definida en un conjunto A se dice total si y solo si todos los elementos de A se relacionan entre sí (es decir: $\forall x, y \in A / x \Re y \vee y \Re x$)

Ejemplos:

1) En el conjunto de los números enteros definimos la relación R de la siguiente forma: un número entero **a** está relacionado con otro número entero **b** si y sólo si **a** es menor o igual que **b**.

Con símbolos, a R b $\Leftrightarrow$ a ≤ b

Hay que probar que es una relación de orden. Queda a cargo del lector.

Luego de eso, es fácil ver que si tomamos 2 números cualquiera, siembre se va a cumplir que el primero es menor o igual que el segundo o al revés, que el segundo es menor o igual que el primero. Para cualquier par de números enteros uno es menor o igual que el otro. Entonces ésta es una relación de orden total. Dos elementos cualquiera siempre están relacionados.

2) En el conjunto de los números enteros definimos la relación R de la siguiente forma: un número entero **a** está relacionado con otro número entero **b** si y sólo si **a** es múltiplo de **b**.

Con símbolos, a R b $\Leftrightarrow$ $\boldsymbol{a = \dot{b}}$

Primero hay que probar que es una relación de orden. Queda otra vez a cargo del lector.

Luego, vamos a investigar si 2 números cualquiera están relacionados. Si tomamos el 5 y el 30, ¿5 es múltiplo de 30 o 30 es múltiplo de 5? Sí, vemos que 30 es múltiplo de 5. Entonces el par (30, 5) pertenece a la relación. Vamos a probar con otros pares de números.

Si tomamos el 8 y el 27, ¿8 es múltiplo de 27 o 27 es múltiplo de 8?

La respuesta es NO. Ninguna de las dos condiciones se cumple. El 8 no está relacionado con el 27 y tampoco el 27 está relacionado con el 8.

Entonces ésta es una relación de orden parcial.

Funciones

Definición: Sean A y B conjuntos cualesquiera. Se dice que f es una función de A en B y se anota $f : A \to B$ si y solo si f es una relación de A en B que cumple: para cada elemento x de A existe un único elemento y de B tal que (x, y) es elemento de f .

$$f : A \to B \begin{cases} * f \text{ es una relación de} A \text{ en } B \text{ o sea } f \subseteq A \times B \\ * \text{Existencia} : (\forall x)(x \in A \Rightarrow \exists y)(y \in B \wedge (x, y) \in f) \\ * \text{Unicidad} : (\forall x)(\forall y)(\forall z)(((x, y) \in f \wedge (x, z) \in f) \Rightarrow y = z) \end{cases}$$

Entonces decimos que una función asocia a cada elemento de un conjunto, llamado dominio, uno y sólo un elemento de un segundo conjunto, llamado codominio.

Sea $f : A \to B$ una función de A en B. Se dice que:

- A es el dominio de f y se simboliza $\mathcal{D}(f)$
- B es el codominio de f y se simboliza $\mathcal{C}(f)$
- Si $(x, y) \in f$ diremos que: $\begin{cases} x \text{ es la PREIMAGEN de } y \text{ al aplicar } f \\ y \text{ es la IMAGEN de } x \text{ al aplicar } f \end{cases}$
- Recorrido de f : es el conjunto de todas las imágenes al aplicar f . ($\mathcal{R}(f)$)

Ejemplo:

Sea el dominio A = {1, 2, 4} y como codominio el conjunto B = {3, 6, 10, 12, 99}

Definamos la función **f** de modo tal que **f(x) = 3x**

Entonces a cada elemento le corresponde su triple.

Al 1 le corresponde el 3, al 2 el 6 y al 4 el 12.

f (1) = 3 la imagen de 1 es 3; la preimagen del 3 es el 1.

f (2) = 6

f (4) = 12

El recorrido de f es el conjunto de las imágenes. Rec (f)= {3, 6, 12}

El recorrido no coincide con el codominio, porque hay elementos del codominio que no tienen preimagen. En este caso, el 10 y el 99.

Clasificación de funciones

Función inyectiva: Una función $f / f : A \to B$, es inyectiva si y solo si a todo par de elementos distintos del dominio les corresponden elementos distintos del codominio.

$$f : A \to B \ \ inyectiva \ \Leftrightarrow \ \forall x_1, x_1 \in A, \forall x_2, x_2 \in A, (x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2))$$

Función sobreyectiva: Una función $f / f : A \to B$, es sobreyectiva si el recorrido de la función es igual al codominio; o dicho de otro modo, si todo elemento del codominio tiene preimagen.

$$f / f : A \to B \ \ \ sobreyectiva \ \Leftrightarrow \ \Re(f) = \mathcal{C}(f)$$

$$f / f : A \to B \ \ \ sobreyectiva \ \Leftrightarrow \forall y, y \in B, \exists x, x \in A, f(x) = y$$

Función biyectiva: Una función es biyectiva si y solo si es inyectiva y sobreyectiva.

Función parcial: Una función parcial es una relación que asocia algunos elementos de un conjunto (llamado dominio) con un único elemento de otro conjunto (llamado codominio). No es necesario que todos los elementos del dominio estén asociados con algún elemento del codominio.

Si todos los elementos de un conjunto X se asocian con algún elemento de Y mediante una función parcial f: X→Y, entonces se dice que f es una **función total**, o simplemente una función, como se entiende tradicionalmente este concepto en matemáticas. No todas las funciones parciales son funciones totales. En matemática, una función se dice que es total si *está definida* para todo el conjunto de partida.

Función inversa

Previamente veamos un ejemplo:

Consideremos las siguientes relaciones definidas en el conjunto $A = \{1,2,3,4\}$

$$R = \{(2,3),(1,4),(3,2),(4,4)\}$$

$$S = \{(2,3),(1,4),(3,1),(4,2)\}$$

Analiza si $R \ y \ S$ son funciones. Halla $R^{-1} \ y \ S^{-1}$ (la relaciones inversas). Analiza si son funciones. ¿Qué será necesario para que la inversa de una función también sea una función?

Función inversa: Sea $f / f : A \to B$ una función BIYECTIVA. Se llama función inversa de f, (f^{-1}) a la función $f^{-1} : B \to A$ tal que $f^{-1}(y) = x \Leftrightarrow f(x) = y$

Observaciones:

- Si f es biyectiva, f^{-1} es biyectiva
- Ampliaremos la definición de función inversa. Hemos definido las funciones parciales, entonces solo es necesario que la función f sea inyectiva para que su inversa también sea una función (parcial o total)

Función compuesta: Dadas las funciones $f / f : A \to B$ y $g / g : C \to D$, tales que el recorrido de f ($\Re(f)$) esté incluido en el dominio de g, se define la relación $(g \circ f) : A \to D / (g \circ f)(x) = g(f(x))$. Estamos definiendo una nueva función $h : A \to D$, tal que para cada $x \in A$, $h(x) = g(f(x))$.

Currificación

Haskell es un lenguaje de programación puramente funcional. El modelo funcional tiene como objetivo la utilización de funciones matemáticas puras sin efectos laterales. El esquema del modelo funcional es similar al de una calculadora. El usuario introduce una expresión inicial y el sistema la evalúa mediante un proceso de reducción. En este proceso se utilizan las definiciones de función realizadas por el programador hasta obtener un valor no reducible.
Los orígenes teóricos del modelo funcional se remontan a los años 30. A comienzos de los ochenta surgieron una gran cantidad de lenguajes funcionales debido a los avances en las técnicas de implementación. Esta gran cantidad de lenguajes perjudicaba el desarrollo del paradigma funcional. En setiembre de 1978 se decide formar un comité internacional que diseñe un nuevo lenguaje puramente funcional de propósito general denominado Haskell. Con este lenguaje se pretendía unificar las características más importantes de los lenguajes funcionales, como las funciones de orden superior, evaluación perezosa, inferencia de tipos, tipos de datos definidos por el usuario, encaje de patrones y listas. En 1998 se proporciona una versión estable del lenguaje, que se denominará Haskell98 a la vez que se continúa la investigación de nuevas características.

Haskell Brooks Curry ha dado el nombre al lenguaje haskell y a una técnica conocida como curryficación. Haskell Brooks Curry era un matemático estadounidense y lógico. Nació el 12 de septiembre de 1900, en Millis, Massachusetts. Es conocido por su trabajo en lógica combinatoria. El enfoque del trabajo de Curry eran intentos de mostrar que la lógica combinatoria podría proporcionar una base para las matemáticas.

Haskell es un lenguaje funcional fuertemente tipificado. Esto indica que los elementos del lenguaje utilizables están clasificados en distintas categorías o tipos. Cada objeto del lenguaje (valores, funciones, operadores, etc.) tiene tipo. Esta información permite comprobar que se hace un uso consistente de los distintos elementos del lenguaje.

Hay distintas estrategias para evaluar una expresión matemática.
Por ejemplo, (5*3)+6.
Los lenguajes tradicionales evalúan todos los argumentos de una función antes de conocer si éstos serán utilizados. Haskell utiliza la evaluación perezosa que consiste en no evaluar un argumento hasta que no se necesita.

El sistema de tipos de haskell es estático, esto significa que las comprobaciones de tipo se realizan durante la compilación del programa, antes de la ejecución del mismo.

Tipos simples predefinidos en haskell

Principales tipos de datos

- El tipo Bool. En haskell solo hay dos constantes para este tipo {True, False}. Algunos ejemplos de funciones y operadores del tipo Bool.

  &&: Conjunción lógica

  &&::BoolxBool->Bool

  &&:: Bool ->Bool ->Bool (notación currificada)

  ||: Disyunción lógica

  || :: BoolxBool->Bool

  ||:: Bool -> Bool -> Bool

  not: Negación lógica

  not:: Bool-> Bool

- El tipo Int. Son números enteros de precisión limitada. El intervalo de representación depende de la arquitectura de la máquina y de la implementación de Hugs.

  4::Int

  8<4:: Bool

- El tipo Integer. Son los números enteros de precisión ilimitada. Representa el conjunto de los enteros matemáticos. Los cálculos realizados con Integer son menos eficientes que con Int, son más lentos.

- El tipo Char. Un valos de tipo Char representa un carácter (una letra, un dígito, un signo de puntuación, etc.) Un valor constante de tipo Char se escribe siempre entre comillas simples: ‘a’, ‘?’, ‘1’, ‘H’.

- El tipo String. Es el tipo de las cadenas de caracteres. Un texto entre comillas dobles es un elemento de tipo String: “parque”, “aHoRa”.

Los tipos:

Consideremos la función f de dominio AXB y codominio C, que en forma resumida se escribe f : AXB –> C, y ahora consideremos la función g que aplicada al conjunto A y luego al conjunto B se obtiene un elemento de C, g : A -> B -> C

Hay un isomorfismo entre ambos dominios, el de f y el de g. Se puede pasar de uno a otro, en Haskell, con las funciones curry y uncurry.

Un ejemplo concreto: consideremos la función **suma** que dados dos enteros devuelve la suma de ambos.

suma: (ZxZ)-> Z

Esto se interpreta: **suma** es una función que recibe un par de números enteros y devuelve un número entero.

suma: Z-> Z -> Z

Esto se interpreta: **suma** es una función que recibe un número entero seguido de otro entero y devuelve la suma de ambos.

Esta forma de denotar los tipos de las funciones se denomina **currificada**.

El símbolo -> (“implica”) asocia hacia la derecha. Entonces la expresión **suma: Z-> Z -> Z** y la que sigue **suma: Z-> (Z -> Z)** son equivalentes. Existe un isomorfismo.

La función **suma: Z-> (Z -> Z)** recibe un entero y devuelve una función que a su vez recibe un entero y devuelve un entero.

- ✓ (suma 5 9) se interpreta como la aplicación de la función suma a 5, luego la aplicación de la función resultante a 9.
- ✓ El tipo de la expresión (suma 5 9) es Z, es decir (suma 5 9) es un entero y en particular su valor es 14.
- ✓ (suma 8) se interpreta como la aplicación de la función suma a 8.
- ✓ El tipo de la expresión (suma 8) es Z->Z, (suma 8) es una función que recibe un entero y devuelve otro entero que es el resultado de sumar el entero dado a 8.

En la siguiente expresión f: A-> B decimos que f es una función que recibe un parámetro de tipo A y devuelve un elemento de tipo B. Es decir, f es de tipo A-> B.

La función **producto** es de tipo N-> N-> N ¿Cómo se interpreta?

Haskell es un lenguaje de programación fuertemente tipado, eso significa que todas las expresiones tienen tipo. Si una expresión en Haskell no tiene tipo quiere decir que es una expresión incorrecta. Si la expresión es incorrecta será imposible definir cuál es su tipo. Esta particularidad reduce considerablemente los errores de programación.

Indicar si las siguientes expresiones son correctas, es decir, indicar si tienen tipo. En caso de ser correctas indicar su tipo.

x, y, z :: Int

- sum x y z
- sum 3
- sum 3 5
- sum 8 x
- div 5
- div 6 4
- div 6 4 1
- prod x
- mod 7 3 2
- mod 9 5
- mod 8
- abs 6
- abs 7 7

En la notación currificada así como podemos escribir funciones que devuelven funciones, podemos escribir funciones que reciban funciones como parámetros.

Veamos un ejemplo: **`f: (Z -> Z)-> Z,`** **f**es una función que recibe una función de Z en Z como parámetro y devuelve un entero.

Observación importante: **`(Z -> Z)-> Z`** no es lo mismo que **`Z-> Z -> Z`** ¿Podrías explicar porqué?

Veamos otras funciones: **`g: (Z -> Z)-> Z->Z`**

`h: Z->(Z -> Z)-> Z -> Z`

¿Cuál es el tipo de **g**? ¿Cuántos parámetros recibe? ¿En qué orden? ¿Cuál es el tipo de **h**?

Notación matemática	Notación currificada
g(x)	g x
f(x, y)	f x y
g (f (x, y))	g (f x y)
f (x, g (y))	f x (g y)
h (x + y)	h (x + y)
j (x) + y	j x + y

Consideremos las funciones mencionadas anteriormente:

f:: (Z -> Z)-> Z

g:: (Z -> Z)-> Z->Z

h:: Z->(Z -> Z)-> Z -> Z

x, y :: Z

- **(g f)::Z;** la aplicación de g a f es un entero.
- **(h x f)::Z;** la aplicación de h a x a f es un entero.
- **(h y)::(Z->Z)->Z->Z;** es una función que recibe una función de Z->Z, luego un entero y devuelve un entero.

Veamos otros ejemplos.

Sean

f:: (Z -> Z)-> Z

p:: Z->Z

h:: Z->(Z -> Z)-> Z -> Z

x::Z

Analicemos cuáles de las siguientes expresiones son correctas, es decir, tienen tipo, y cuáles no tienen tipo. Explicar las respuestas e indicar el tipo en el caso de las expresiones correctas:

f x – incorrecta

f p – tiene tipo (es correcta)

f p x- no tiene tipo

f (p x)- no tiene tipo

h (f p)- correcta

h f p- incorrecta

h x f- correcta

h x p (f p)- correcta

h (f p) (h x p)- correcta

h (f p) h x p- incorrecta

h (f p) (h x p) (f p)- correcta

f (p x) (h x p x)- incorrecta

Ejercicio:

Sean

f :: N-> N

g:: N->N

h :: (N -> N)-> N -> N

t :: ((N -> N) -> N -> N) -> N -> N

p :: N -> (N -> N) -> N -> N

x, y :: N

Indicar si las siguientes expresiones tienen tipo, en caso afirmativo indicarlo. Justificar en todos los casos la respuesta.

1. h x y
2. h f
3. g (h f)
4. g (h f 3)
5. (p x) g
6. t h
7. t (h f)
8. h (p x g)
9. g (t h x)
10. t (p x) x
11. f (p 3 f 2) 2
12. h (t h)
13. f (p x g y) (h g)
14. p (g y) (h f) (t h y)
15. g (h g x)

Introducción a Haskell

Instalación de Haskell.

Descargue el instalador desde www.haskell.org

Preste atención al sistema operativo que utiliza su computadora.

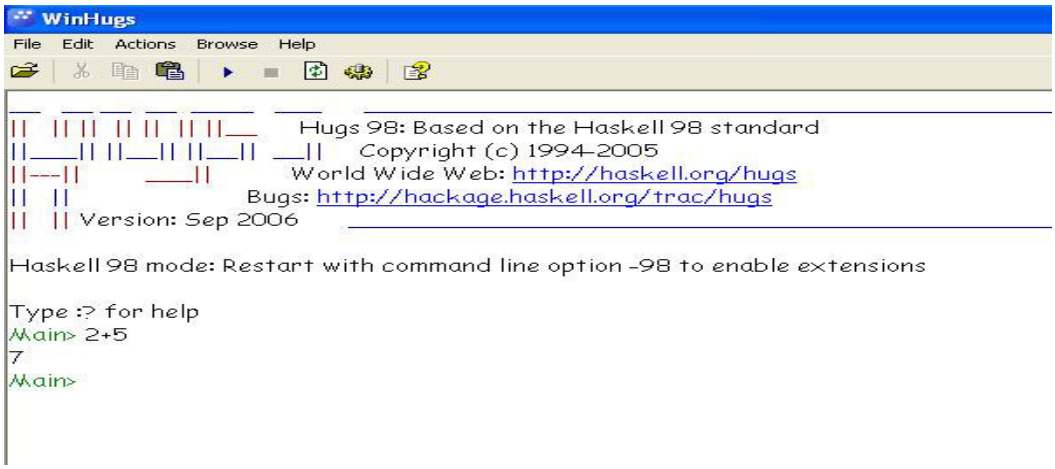
Luego de instalado, ¡pruébelo! Si puede digitar 2 + 5 y le da 7, ¡¡¡ felicitaciones!!!

Primeros pasos con Haskell.

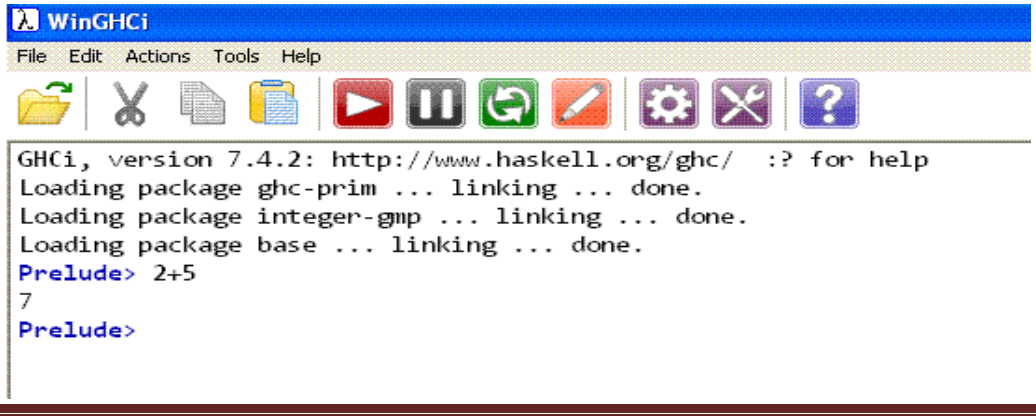
Este software vamos a usarlo a lo largo de todo el año. Necesitamos empezar a familiarizarnos con él. Como práctica individual, se puede empezar utilizando Haskell como calculadora, haciendo las operaciones del práctico correspondiente. Las operaciones comunes funcionan bien.

Luego de instalado el programa, en la primer pantalla que aparece, llamada "línea de comandos", si escribimos 2+5 y digitamos enter, debería aparecer un "7".

En la versión de Haskell 98, utilizando Hugs como intérprete por defecto, se verá así:



En la versión 2012 se ve así:



Funciones predefinidas.

Existen funciones especiales que ya vienen predefinidas en Haskell. Algunas son completamente nuevas y seguramente no las ha utilizado el estudiante antes.

33	7
5	4

div a b : el cociente de la división entera entre a y b.

El cociente de la división entera entre 33 y 7 es 4. Aquí se llama **div.**

div 33 7 = 4 Observe que no lleva paréntesis. Sólo un espacio después de la palabra div y otro espacio entre el 33 y el 7.

La función **div** espera 2 números enteros y su respuesta es otro número entero.

¿Siempre se puede hacer la división entre a y b, para cualquier a y b enteros?

$$div: Z \times Z^* \rightarrow Z$$

Esto significa que la función **div** tiene por dominio un par ordenado de números enteros, en el cual el primer elemento es un número entero, cualquiera, y el segundo elemento es un número entero diferente de cero. Además el codominio es un número entero.

Si digita div 33 4 6 el programa le dará un mensaje de error. Hay demasiados parámetros en este caso.

Si digita div 33 el programa le dará otro mensaje de error. Hay pocos parámetros en este caso.

mod a b : resto de dividir a entre b

mod 33 7 = 5

mod 7 33 = 7

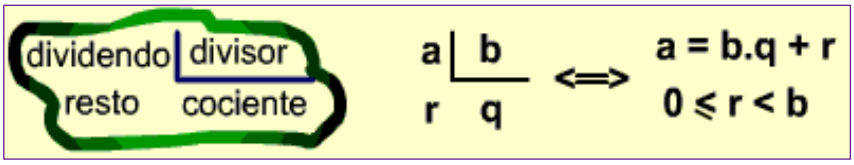
Ejercicio: ¿Que espera obtener si digitamos mod 19 3? ¿Y si digitamos mod (-19) 3?

(Observación: los números negativos necesitan paréntesis en Haskell)

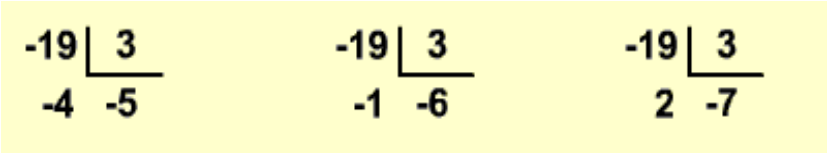
Recuerde que en la definición de división entera es fundamental que el resto de la división entera siempre sea menor que el divisor y además positivo.

División entera entre a y b

Definición: Sean a y b números enteros, con b positivo. Dividir a entre b significa encontrar dos números enteros q y r de modo que se verifique:



Vamos a dividir (-19) entre 3. ¿Cuál de las siguientes opciones será la correcta?



En los tres casos se cumple que (-19) = 3.q + r, pero sólo en uno el resto es positivo y menor que 3.

¿Quieres probarlo con tu computadora? Recuerda escribir los números negativos entre paréntesis. mod (-19) 3 div (-19) 3

Funciones lógicas: || (or) && (and) == (igual)

Las funciones lógicas devuelven un elemento del conjunto **Bool.** El conjunto **Bool** es un conjunto que tiene solamente 2 elementos: Verdadero y Falso.

Bool = {True, False}

Ejemplos:

- Nos preguntamos si 8 será mayor que 5. La respuesta es sí, por supuesto.

En haskell se escribe así: 8>5. Al dar enter la respuesta es True.

- (8 > 5) && (3 > 7) En este caso la primera expresión es verdadera pero la segunda no. Entonces la respuesta será False.
- También nos podremos preguntar si el resto de dividir 9 entre 2 será 0. **mod 9 2 == 0** (Observe que hay 2 signos de = pegados. No se puede dejar espacio entre ellos). La respuesta es False.

En resumen:

mod 9 2 tiene como respuesta 1 (Calcula el resto de la división entera de 9 entre 2)

mod 9 2 ==1 tiene respuesta True. (pregunta ¿El resto de la división entera de 9 entre 2 es 1?)

mod 9 2 ==0 tiene respuesta False. (pregunta ¿El resto de la división entera de 9 entre 2 es 0?)

Otras funciones lógicas predefinidas en Hugs son por ejemplo: /= (distinto), not (negación).

Funciones y su implementación en Haskell.

Recordemos que una función es una correspondencia entre 2 conjuntos, llamados dominio y codominio, de forma tal que cada elemento del primer conjunto tiene uno y sólo un correspondiente en el segundo conjunto.
Si todos los elementos del dominio tienen imagen, decimos que la función es total.
Si algún elemento del dominio no tiene correspondiente, no tiene imagen, decimos que la función es parcial

Veamos ejemplos de funciones sencillas.
Llamaremos "f" a una función que cada número entero le asigne su doble. Mediante la función f al 5 le corresponde el 10, al 7 le corresponde el 14, al (-3) le asigna el (-6), etc.

La expresión matemática de la función es f(x) = 2x. Esto significa que mediante la función f a cada número x le corresponde el 2x. Sin embargo, no se indica que estamos trabajando con números enteros.
Para indicar, entonces, en forma correcta, que la función f es de dominio entero y codominio entero y que a cada número le corresponde su doble, tenemos que escribir:

$f : Z \rightarrow Z \ / \ f(x) = 2x$

En forma general, si A es el dominio y B el codominio, $f : A \rightarrow B / f(x) = 2x$
Por otra parte, la "fórmula" no tiene porqué ser la misma para todos los elementos del dominio.
Por ejemplo, queremos escribir la expresión analítica, esto es, "la fórmula", de la función "gato" que le asigna a cada número entero par su doble, y a cada entero impar su triple.

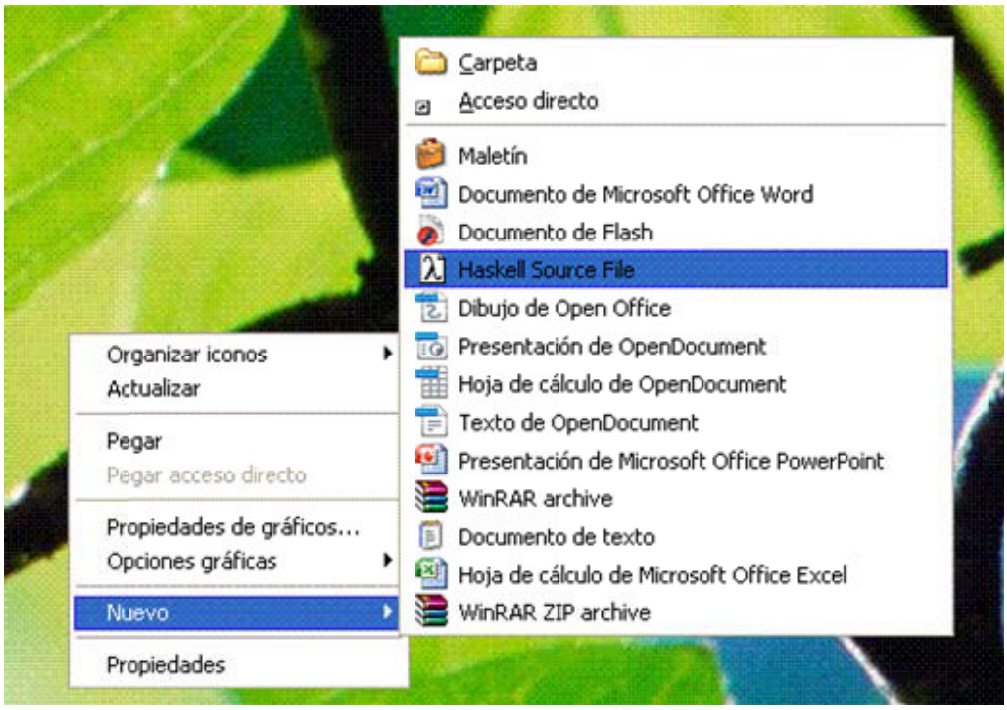
$$gato : Z \rightarrow Z \ / \ gato(x) = \begin{cases} 2x & si \quad x \ es \ par \\ 3x & si \quad x \ es \ impar \end{cases}$$

Por ejemplo, gato(4)=8, gato(5)=15, gato(6)=12, gato(-7)=-21, gato(0) = 0, etc.

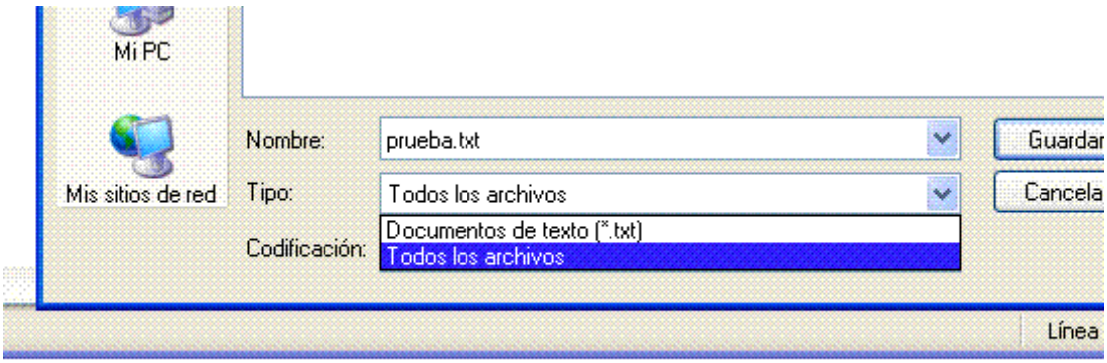
Definición de funciones en Haskell:

En Haskell se pueden definir nuevas funciones.
Para definir una función primero hay que crear un archivo de texto nuevo, con extensión **.hs**

Por ejemplo, en el "escritorio" creamos el archivo prueba.hs (suponiendo que estamos utilizando Windows. Para los demás sistemas operativos ver en haskell.org)



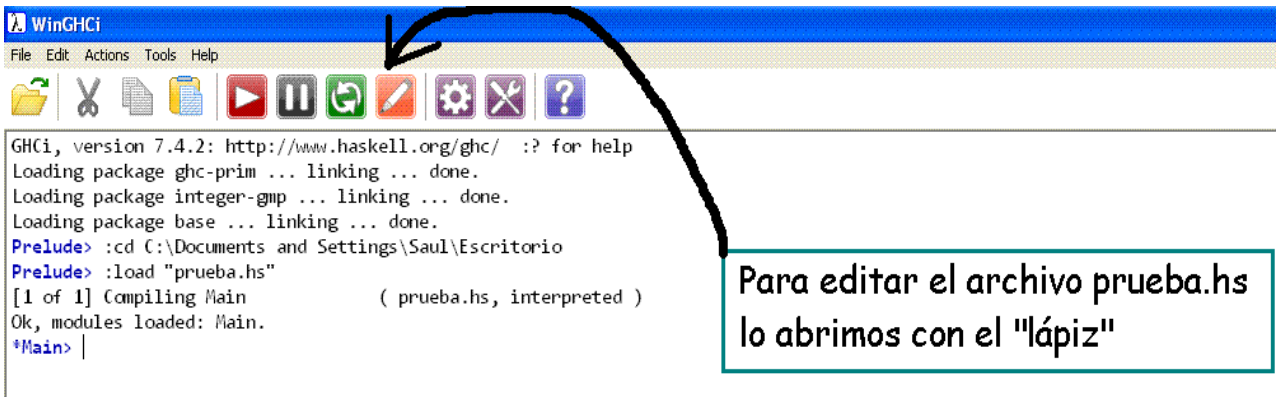
También podemos crear un archivo nuevo de texto prueba.txt y luego renombrarlo como prueba.hs, teniendo mucho cuidado que el nombre no quede prueba.hs.txt, lo cual haría que no funcione.



Primeros pasos en Haskell:

Ahora estamos en condiciones de empezar.

Abrimos el programa Haskell, WinGHCi. Luego cargamos el archivo prueba.hs, mediante "file" , "load" o utilizando el icono de "abrir carpeta". (En Ubuntu para cargar el archivo escribimos :load\seguido de la ruta a nuestro archivo)



Una vez editado el archivo "prueba.hs", vamos a escribir algo sencillo, solo para verificar su buen funcionamiento. Por ejemplo, escribimos x = 45. Luego guardamos y cerramos. Para que el programa Haskell lea de nuevo el archivo después de haber sido guardado, hay que teclear el ícono verde. Esto se llama recargar el archivo. (En Ubuntu escribimos :reload)

Cada vez que modificamos el archivo prueba, hay que "cargarlo" de nuevo.
Si ahora digitamos la letra "z" y luego tecleamos enter, la respuesta será un mensaje de error, porque nosotros no le dimos ningún valor a "z". Si digitamos "x" y luego enter, la respuesta debería ser 45, si hicimos las cosas bien. (Hay que digitar la x sola, sin comillas).

1) En una carpeta nueva, o en el escritorio, crear un archivo nuevo, que se llamará nuevo.hs
2) Abrir el programa Haskell, cuyo ícono más común es 

3) Abrir el archivo nuevo.hs
4) Escribir x = 5
5) Guardar y cerrar este archivo.
6) Recargar el archivo.
7) Digitar x y luego enter.

Observaciones:

- ✓ Los nombres de las funciones comienzan con letras minúsculas.
- ✓ Los tipos (nombres del dominio y codominio, Integer en este caso) comienzan con mayúscula (la primera mayúscula y todas las demás minúsculas).

Para mayor información se sugiere leer alguna introducción a Haskell, directamente descargada desde el sitio web. Buscar " haskell.org en español"

Definición de funciones en Haskell:

Ejemplo 1: Empecemos con la función $f : Z \rightarrow Z$ / $f(x) = 2x$

En una carpeta, creamos un archivo prueba.hs (puede ser el mismo archivo que usamos antes)

Lo abrimos con Haskell y escribimos en una línea, el dominio y codominio. En otra línea escribimos la fórmula de la función. La notación es la siguiente:

```
f :: Integer ->  Integer
f  x  =  2 * x
```

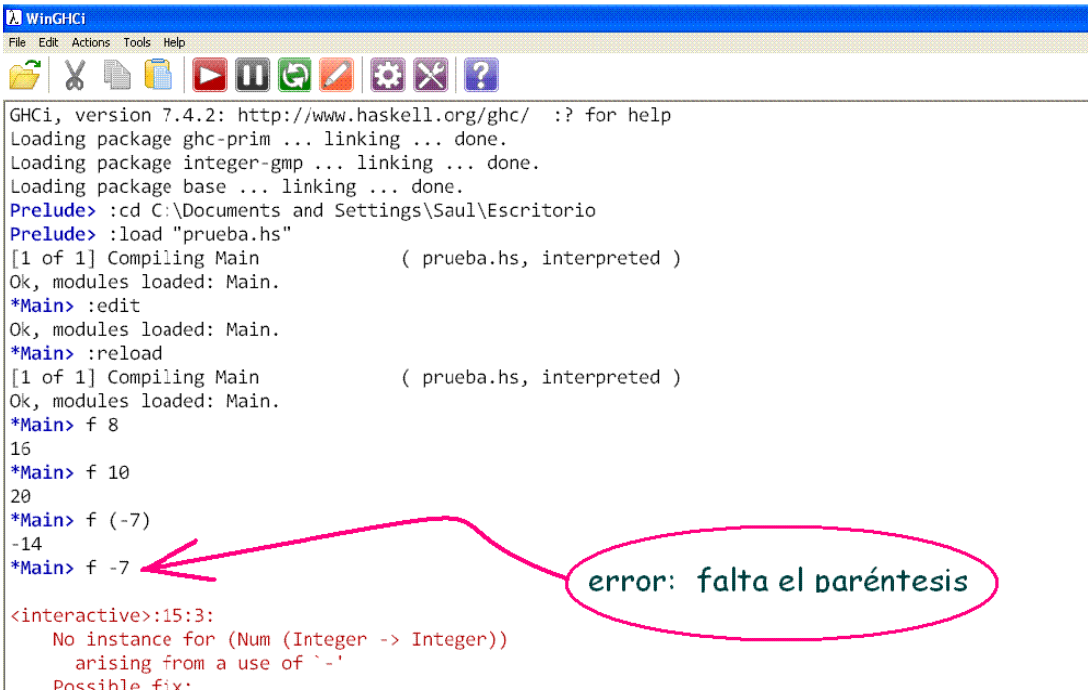
En la primera línea estamos indicando que la función se llamará "f" y que su dominio es el conjunto de los enteros, llamados Integer en Haskell, y el codominio es también Integer.

Los cuatro puntos forman parte de la notación. La flecha, que separa el dominio del codominio, se hace con el signo de menos y el signo mayor juntos y sin espacio.

No dejar espacio en el medio, porque dará error. **->**correcto **- >**error

La separación entre la letra f y los cuatro puntos, así como los demás espacios se han puesto sólo para facilitar la lectura. El programa no les adjudica valor a los espacios de más.

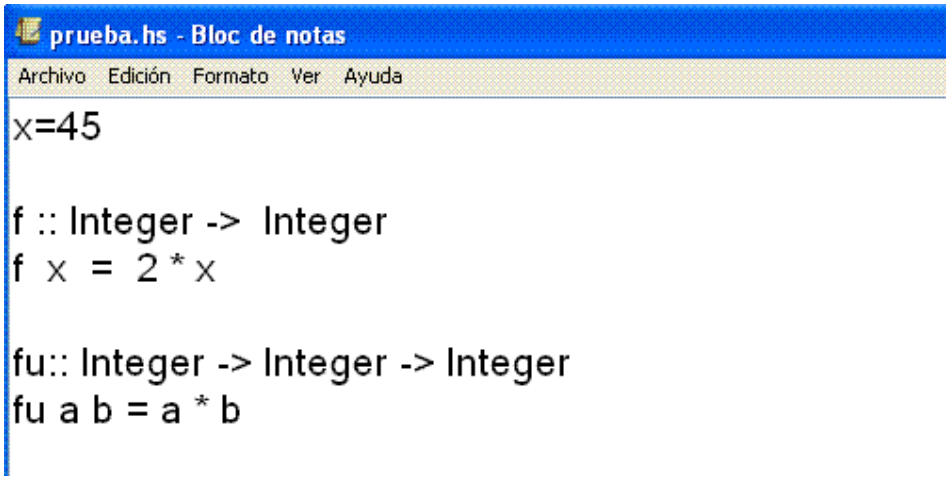
f(x)=2x se escribe en Haskell f x = 2*x. Ahora, habiendo escrito esto en el archivo prueba.hs, lo guardamos, cerramos y recargamos. Si ahora, en la pantalla de comandos, escribimos f 8 y luego enter, debería aparecer el 16.



Ejemplo 2:

Se pueden definir funciones con más de una variable. Por ejemplo, recordemos que una fórmula muy utilizada en física es que F=m.a Fuerza = masa * aceleración.

Podemos también definirla con Haskell. Pero el inconveniente es que en el mismo archivo no pueden coexistir dos funciones con el mismo nombre. En nuestro archivo prueba.hs ya tenemos una función llamada "f". Hacemos un archivo nuevo o podemos llamarla de otra forma.
Por ejemplo, podemos llamarle a la fuerza, fu.
Entonces, el archivo prueba.hs, editado, quedará así:



La función fu tiene como dominio un entero y otro entero, y su codominio es otro entero.
Con la notación matemática habitual, escribiríamos fu (4,5)=20
En haskell, escribimos fu 4 5 = 20

Ejercicio: Evaluar ("a mano") las siguientes expresiones, suponiendo que seguimos con nuestro archivo prueba.hs y justificar las respuestas.

a) f (f 5)
b) f (f (f 3))
c) f 3 4
d) fu 3 4
e) fu (fu 2 3) (fu 5 2)
f) fu (f 5) (f 2)
g) f x
h) f a

Ahora lo puedes verificar con tu computadora. Es una forma de obtener la respuesta correcta.
¿Algunos dieron "error"? ¿Por qué?

Ejemplo 3:
Intentaremos ahora definir con Haskell la función **gato**. Para los números pares hay que aplicar una fórmula y para los números impares otra fórmula. Previamente hay que hacer una función

que nos indique si un número dado es par o no. Hay funciones predefinidas que ya lo hacen, pero no las usaremos.

Entonces antes de definir la función **gato** vamos a definir una función que determine si un número es par o impar (puede ser una función o dos).

Pero antes tenemos todavía que aprender algo más.

Definición de funciones por casos.

En Haskell, cuando la función definida tiene varias líneas, se trabaja de la siguiente manera: Primero lee la primera línea. Si el argumento de la función coincide con el argumento de entrada, se obtiene la respuesta = salida y ahí termina. Si el argumento no coincide, pasa a leer el segundo, y así sucesivamente.

Veamos un ejemplo:

```
pato::Integer->Integer
pato 4 = 17
pato 5 = 81
pato n = n+1
```

¿Qué es lo que hace la función pato? Si en la pantalla de comandos digitamos pato 4 la respuesta será 17; si digitamos pato 5, la respuesta es 81; si digitamos pato 10, la respuesta es 11, porque ahora estamos en la tercera línea.

Observación: si hacemos algo mal puede suceder que Haskell entre en un bucle, esto es, un proceso que no termina. Se queda "pensando", parpadeando, y no puede terminar el proceso.

En ese caso hay que interrumpir los cálculos con el botón de interrupción.

Ejercicio: Implementar una función por casos, llamada **casidoble** que a todos los números enteros los multiplique por 2 menos al 3 y al 4, que los deja iguales.

Definición de funciones en Haskell: El conjunto Bool.

El conjunto de los números enteros tiene infinitos elementos y es el que usamos hasta ahora.
El conjunto Bool tiene sólo 2 elementos: True y False (verdadero y falso).

Ejemplo: queremos implementar una función **contraseña** que servirá para evaluar si un número ingresado por el operador es el correcto en una contraseña. Por ejemplo, supongamos que la contraseña es 3214. La idea entonces es que nuestra función recibe un número entero y la respuesta es correcto o incorrecto. Dicho de otra forma, la respuesta es True o False.

```
contraseña :: Integer ->Bool
contraseña 3214   =  True
contraseña    a      =  False
```

Esta función, sin embargo, que parece estar bien escrita, da un mensaje de error.
Ayuda1: Tiene que ver con nuestro idioma español
Ayuda2: Si la hubiéramos llamado password, zapato o pepe, hubiera funcionado bien.
Ayuda3: Si su nombre fuera **ñ**andú, **ñ**oqui o ca**ñ**o, tampoco habría funcionado.

Funciones recursivas en Haskell

Una función recursiva es una **función** que se llama a sí misma en algún paso previo.

Observación: Tiene que existir alguna condición de terminación, también llamada condición de parada.
Observación 2: si no es función, no puede ser función recursiva.

Ejemplo:

```
tio::Integer->Integer
tio n =  9 + tio (n-1)
```

Intentaremos evaluar ahora tio 5
Aplicando esta función, vemos que, si sustituimos n por 5 queda:

tio 5 = 9 + tio (5-1) O sea, haciendo la resta,

tio 5 = 9 + tio 4

Ahora volvemos a aplicar la función para calcular tio 4 y ya hacemos las operaciones:

tio 4 = 9 + tio 3. Y aplicando de nuevo la función

tio 3 = 9 + tio 2

tio 2 = 9 + tio 1

tio 1 = 9 + tio 0

tio 0 = 3 + tio (-1)

tio (-1) = 3 + tio (-2)

Y así seguimos en forma infinita..... Aparecerá un mensaje de error. Falta la condición de terminación. Esta "función" está, entonces, mal definida. “No es una función”. Vamos a arreglarla.

```
tio::Integer->Integer
tio 2 = 100
tio n =  9 + tio (n-1)
```

Ahora vamos de nuevo a intentar evaluar tio 5.

tio 5 = 9 + tio 4

tio 4 = 9 + tio 3.

tio 3 = 9 + tio 2

tio 2 = 100

Entonces, ahora que sabemos que tio 2 = 100 podemos calcular, de abajo hacia arriba:

tio 3 = 9 + tio 2 y como tio 2 = 100 entonces tio 3 = 109

tio 4 = 9 + tio 3. y como tio 3 = 109 entonces tio 4 = 118

tio 5 = 9 + tio 4 y como tio 4 = 118 entonces tio 5 = 127 Es la respuesta final.

Lo que hicimos anteriormente se llama secuencia de cómputos.

En resumen, con esta función se puede calcular tio 5, tio 994, tio 3000.

¿Se podrá calcular tio 1 ? ¿Y se podrá calcular tio (-4)?

Esta función, tio, ¿es una función total o parcial, en los enteros?

Definición de funciones recursivas en Haskell: otro ejemplo

Queremos definir una función que dado un número me indique si este es par.

En este ejemplo estamos suponiendo que cuando hablamos de "número" nos referimos a números enteros no negativos.

Observemos que si restamos un número par a otro número par, obtenemos un número también par. Si a un número par le restamos 2 muchas, muchas veces, al final obtendremos el 0. Si partimos de un número que no sabemos si es par o no, y le restamos 2 muchísimas veces, al final obtendremos un 0 o un 1.
Si el número original era par, obtendremos el 0. Si el número original era impar, obtendremos el 1. Y para hacerlo en forma automática, empezamos comparando nuestro número con el 0 y con el 1. Si nuestro número no es ni 0 ni 1, le restamos 2. Si el resultado es 0 o 1, terminamos. Si no, restamos de nuevo y así sucesivamente.

Ejercicio: Implementar una función por casos, llamada **par**, que nos indique si un número entero positivo dado es par o no.

Otro ejemplo y una nueva herramienta.
Se quiere definir la función h, cuyo dominio y codominio sea el conjunto de los números enteros positivos, y que trabaja de la siguiente forma: a los números pares le asigna el 10 y a los números impares el 11.

$$h: Z^+ \to Z^+ / h(x) = \begin{cases} 10 & si \quad x\ es\ par \\ 11 & si \quad x\ es\ impar \end{cases}$$

Otra forma de escribirla, absolutamente equivalente, es:

$$h: Z^+ \to Z^+ / h(x) = \begin{cases} 10 & si \quad x = \dot{2} \\ 11 & si \quad x \neq \dot{2} \end{cases}$$

El 2 con el puntito arriba significa múltiplo de 2.

Entonces, h(5)=11, h(6)=10, h(7)=11, h(8004)=10.

EXPRESIONES CONDICIONALES: la sentencia condicional if.

La cláusula if significa "si", es un si condicional.

Primero vamos a escribirlo en español:

si (.....) entonces (.....) y sino entonces (.....)

si (llueve) entonces (vamos al cine) y sino entonces (vamos al tablado)

si (la entrada cuesta menos de $200) entonces (vamos al cine) y sino entonces (vamos a caminar y te invito un helado)

En resumen:

si (condición lógica) entonces (acción 1) y sino entonces (acción 2)

```
if ( condición lógica) then ( acción 1) else ( acción 2 )
```

Dentro del primer paréntesis no se puede escribir sólo un número, una palabra o una frase. Tiene que ser un valor **booleano**. Es una condición lógica, por tanto sus valores son True o False.

Ejemplo:

Implementaremos la función **edad**. Si el número de entrada es menor que 18, le suma 3 unidades. Si es mayor o igual que 18, le resta 2 unidades. Entonces, si todo marcha bien, debería quedar así:

edad (14) = 17

edad (17) = 20

edad (18) = 16

edad (25) = 23

Una forma de definir dicha función es:

```
edad:: Integer-> Integer
edad a = if ( a<18 ) then ( a+3 ) else ( a-2 )
```

Si el valor de a es menor de 18, el paréntesis (a<18) tiene el valor True, entonces lo que se ejecuta es (a+3). Cuando a no es menor de 18, el paréntesis (a<18) tiene el valor False, entonces lo que se ejecuta es (a-2).

El signo de igual = ==

Asignación: Cuando escribimos j=3, estamos diciendo que el valor de j es 3. Antes j no tenía ningún valor previo, y a partir de ahora sabemos que el valor de j es 3. Le estamos asignando

un valor a la variable j: el valor 3. Más aún. Aunque j antes tuviera otro valor, a partir de la expresión j=3 el valor de j será 3.

Comparación: Otra situación diferente es comparar el valor de una variable con otro. La variable m tiene un valor, el cual no conocemos. Quizás sea 17. Quisiéramos comparar el valor de m, que no sabemos cuál es, con el número 17.¿Será m igual a 17 o no? Cuando nos hacemos una pregunta lógica, el signo de igual es ==.Esto es, son 2 signos de igual juntos, sin espacio entre ellos.

Si m valía 17, la expresión m==17 tiene valor True.

Si m no valía 17, la expresión m==17 tiene valor False.

Sea cual sea el valor de m la expresión m==17 no cambia el valor original de m. No le estamos asignando ningún valor a m. Sólo se lo está comparando.

Ahora estamos en condiciones de poder hacer la función h.

$$h : Z^+ \to Z^+ / h(x) = \begin{cases} 10 & si \quad x\ es\ par \\ 11 & si \quad x\ es\ impar \end{cases}$$

Lo nuevo que vamos a ver ahora es que se puede usar en haskell una función como insumo, como una parte integrante de la misma. En esta función **h** tenemos que discriminar según el valor de entrada, según sea par o no.. Pero como ya tenemos hecha la función par, la podemos utilizar.

Para poder utilizar varias funciones simultáneamente, una forma de hacerlo es escribirla en el mismo archivo. En nuestro caso, todo está escrito en el archivo **prueba.hs**

Ya hicimos la función par. Ahora llegó el momento de utilizarla.

Si la entrada de la función **h** es un número que lo llamamos **x**, el resultado dependerá de saber si x es par o impar.

Recordemos que si x es par, entonces par x tiene el valor True.

```
h:: Integer-> Integer
h x = if  (par x  ) then ( 10 ) else ( 11 )
```

Entonces, si x es par, (par x) tiene valor True y la respuesta es 10.

Si x es impar, (par x) tiene valor False y la respuesta es 11.

Intenta escribirlo en tu computadora.

Ejercicio: Escribir la función gato para ser ejecutada en Haskell, con dominio y codominio los enteros positivos.

$$gato : Z^+ \to Z^+ \,/\, gato(x) = \begin{cases} 2x & si \quad x\ es\ par \\ 3x & si \quad x\ es\ impar \end{cases}$$

(respuestas al final del capítulo)

Más funciones recursivas.

Ejemplo:

Intentemos descubrir que es lo que hace la función **dado**.

```
1) dado :: Integer -> Integer
2) dado 0 = 5
3) dado n = 4 + dado (n-1)
```

La numeración de los renglones es sólo para poder referirnos a ellos, pero NO debe ponerse al escribir en Haskell, porque daría error.

Vamos a evaluar dado 3.

dado 3 = 4 + dado 2 usando el renglón 3

Como "dado 2" no sabemos aún cuál es su valor, empezamos de nuevo.

dado 2 = 4 + dado 1 usando el renglón 3

dado 1 = 4 + dado 0 usando el renglón 3

dado 0 = 5 usando el renglón 2

Entonces ahora podemos ir de abajo hacia arriba para obtener todos los resultados parciales:

dado 0 = 5

dado 1 = 4 + dado 0 entonces dado 1 = 4 + 5 = 9

dado 2 = 4 + dado 1 entonces dado 2 = 4 + 9 = 13

dado 3 = 4 + dado 2 entonces dado 3 = 4 + 13 = 17

La función dado empieza en 5 y le va sumando 4 en cada paso.

Ejercicio: ¿cuánto es dado 8000?

Veamos: dado 0 no tiene ningún 4.

dado 1 tiene un 4 y el 5.

dado 2 tiene un 4, otro 4 y el 5.

dado 3 tiene un 4, 4, 4 y 5.

Vamos a ordenar un poco esto:

dado 0 = 5

dado 1 = 5 + 4

dado 2 = 5 + 4*2

dado 3 = 5 + 4*3

dado 4 = 5 + 4*4

dado 8000 = 5 + 4*8000 = 32005.

Ejercicio: Calcular “a mano” sapo 5 y deducir la finalidad de la función sapo. ¿Qué es lo que hace?

1) sapo :: Integer -> Integer

2) sapo 0 = 0

3) sapo n = if (mod n 3 > 0) then (1 + sapo (n-1)) else (sapo (n-1))

Más funciones recursivas.

Recordemos: mod 17 2 = 1 El resto de la división entera de 17 entre 2 es 1.Dicho de otra manera, 17 no es divisible entre 2 porque su resto no es 0. El resto de dividir 33 entre 7 es 5. Entonces mod 33 7 = 5

Un número x es divisible entre 2 si el resto de la división entre ese número y 2 es 0. Escrito de otra manera; mod x 2 debe valer 0.

Utilizando esta propiedad, realiza el siguiente ejercicio.

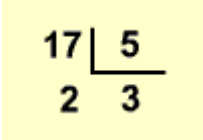
Ejercicio: Implementa la función **parmod** que nos indique si un número dado es o no es par.

Observación: no le podemos llamar "par" porque ya existe en nuestro archivo una función con este nombre. Hay que cambiarle el nombre porque no puede haber 2 funciones diferentes con el mismo nombre porque se produce conflicto. Se puede llamar par2, otropar, o paralelepipedo, etc.

Más funciones recursivas con Haskell.

Queremos implementar la función resto. Esta función lo que hace es proporcionar el resto de la división entera entre 2 números enteros positivos.

Empecemos investigando algo.



¿Qué quiere decir exactamente que el resto de dividir 17 entre 5 es 2? ¿Cómo obtendríamos el mismo resultado con una calculadora si no tuviéramos la tecla de división?
¿Qué operación hemos realizado muchas veces? Vamos a hacer muchas restas, pero primero hacemos una pregunta.
¿17 es mayor que 5? Respuesta: Si. Entonces restamos 17 - 5 = 12.
¿12 es mayor que 5? Respuesta: Si. Entonces restamos 12 - 5 = 7.
¿7 es mayor que 5? Respuesta: Si. Entonces restamos 7 - 5 = 2.
¿2 es mayor que 5? Respuesta: No. Entonces ya terminamos. El resto es 2.

Para tratar de entenderlo un poco más, vamos a escribir el primer renglón de otra forma. Si 17 es mayor que 5, entonces restamos (17-5) y volvemos a comparar, y si no es mayor la respuesta es 17. Vamos ahora a repetir este último renglón, pero en vez de usar los números 17 y 5, usaremos los números a y b.
Si **a** es mayor que **b**, entonces (**a-b)** y **volvemos a comparar** (¿(a-b) es mayor que b?) y si no, la respuesta es **a**. Parece que es una función recursiva.

Ejercicio: Implementar la función resto, en forma recursiva. (Sin utilizar la función "mod").
Ayuda: No olvidarse, además, de considerar además que resto 15 3 = 0
Ejercicio: Implementar la función cociente, en forma recursiva. (No utilizar la función "div").
Ejercicio: Implementar la función "sumacifras" que sume los dígitos de un número entero positivo de 3 cifras.
Por ejemplo, sumacifras 321 = 6 sumacifras 705 = 12

Mensajes de error.
A veces es conveniente que aparezca un mensaje. Una forma de hacerlo con Haskell es con la función predefinida error. La sintaxis es error "texto explicativo"
Ejemplo:

```
resta:: Integer -> Integer-> Integer
resta a  b = if a>=b then (a-b) else error  "no se puede restar"
```

Probar: resta 8 5 y resta 8 51

Uso de guardas:
Cuando hay muchas sentencias con **if**, es más práctico usar guardas. Son muchas sentencias if anidadas. Se expresa de la siguiente forma:

ejemplo:: Dominio -> Codominio

ejemplo a | pregunta lógica 1 = acción 1
           | pregunta lógica 2 = acción 2
           | pregunta lógica 3 = acción 3
           | otherwise   = acción 4

El símbolo **" | "** se puede hacer apretando las teclas AltGr y el número 1 simultáneamente.
El otherwise significa "en todos los demás casos". Su valor de verdad es 1.
Veamos cómo funciona:
Si la pregunta lógica 1 es cierta, se ejecuta la acción 1. Si es falsa, pasamos al otro renglón.
Si la pregunta lógica 2 es cierta, se ejecuta la acción 2. Si es falsa, pasamos al otro renglón.
Si la pregunta lógica 3 es cierta, se ejecuta la acción 3. Si es falsa, pasamos al otro renglón.
Si se llega a este último renglón, siempre se ejecuta la acción 4.
Vamos a utilizar lo que recién aprendimos en la definición de la función “divisible”:

```
divisible  :: Integer -> Integer
divisible a | mod a 2==0    = error "es divisible entre 2"
divisible a | mod a 3==0    = error "es divisible entre 3"
divisible a | mod a 7==0    = error "es divisible entre 7"
divisible a | otherwise     =  100
```

Al ingresar un número, si es divisible entre 2, 3 o 7, aparecen mensajes de error, indicándolo. Si el número ingresado no es divisible entre 2,3 ni 7, el resultado es 100. ¡Ganaste 100 puntos!
Probar con divisible 55.

Ejercicio: Nuevamente implementar la función "sumacifra" que sume los dígitos de un número entero positivo de 3 cifras. Ahora hay que controlar que si el número ingresado no tiene 3 dígitos, aparezca un mensaje de error, del tipo "número incorrecto de cifras"

Ejercicio: Implementar la función "sumatuti" que sume los dígitos de un número entero positivo. En este caso no se indica la cantidad de cifras del número ingresado.
Sumatuti 123456789 = 45
Sumatuti 11111111111111111111 = 20

Respuesta a algunos ejercicios del capítulo:

Funciones y su implementación en Haskell.

Ejercicio: Implementar una función por casos, llamada casi doble que a todos los números enteros los multiplique por 2 menos al 3 y al 4, que los deja iguales.

```
casidoble::Integer->Integer
casidoble 3 = 3
casidoble 4 = 4
casidoble n = 2*n
```

Ejercicio: Implementar una función por casos, llamada par, que nos indique si un número entero positivo es par o no. Si la función tiene que indicar si algo "es" o "no es", su codominio es el conjunto Bool.

```
par::Integer->Bool
par 0 = True
par  1 = False
par n = par (n-2)
```

Para comprobar o testear si la función par hace lo que quisiéramos, vamos a efectuar la secuencia de cómputos de algún número, por ejemplo, el 6

```
par 6 = par 4
par 4 = par 2
par 2 = par 0
par 0 = True   Entonces sí, el 6 es par !!!
```

¿Ahora, que pasará con el 7?

```
par 7 = par 5
par 5 = par 3
par 3 = par 1
par 1 = False        Entonces el 7 no es par.
```

Intenta definirla en tu computadora, utilizando Haskell, y luego probarla con esto ejemplos:

```
par 1000
par 4896651
par (34*67+12-3455+77*13*5-512)
par (- 4)
```

Ejercicio: Escribir la función gato para ser ejecutada en Haskell, con dominio y codominio los enteros positivos.

$$gato : Z^+ \rightarrow Z^+ / gato(x) = \begin{cases} 2x & si \quad x \ es \ par \\ 3x & si \quad x \ es \ impar \end{cases}$$

```
gato :: Integer -> Integer
gato x = if ( par x ) then (2*x) else (3*x)
```

Ejercicio: Calcular sapo 5 y deducir la finalidad de la función sapo. ¿Qué es lo que hace?

1) sapo ::Integer ->Integer

2) sapo 0 = 0

3) sapo n = if (mod n 3 > 0) then (1 + sapo (n-1)) else (sapo (n-1))

Una resolución:

sapo 5 = if (mod 5 3 > 0) then (1 + sapo (5-1)) else (sapo (5-1))

Como mod 5 3 es 2, es mayor que cero y el primer paréntesis es cierto. Entonces se ejecuta 1+sapo4

Hagamos un resumen.

sapo 5 = 1 + sapo 4

sapo 4 = 1 + sapo 3

sapo 3 = sapo 2 porque el mod 3 3 =0 y no es mayor que cero

sapo 2 = 1 + sapo 1

sapo 1 = 1 + sapo 0

sapo 0 = 0

Y ahora de abajo hacia arriba, nos queda que sapo 5 = 4

Hay que observar que esta función va sumando 1 en todos los términos menos en aquello en los que mod n 3 no es mayor que cero. O sea que mod n 3 = 0, o sea en los múltiplos de 3.

Esta función suma 1 unidad de todos los números desde 0 hasta n excepto los múltiplos de 3.

Entonces está contando todos los números naturales que no son múltiplos de 3.

Ejercicio: Escribir la función parmod utilizando la función auxiliar mod.

Si el resto de la división entera entre x y 2 es 0, es par.

Si el resto de la división entera entre x y 2 no es 0, no es par.

Estos "si" nos indican que puede servir utilizar una expresión condicional, una cláusula con if.

```
parmod:: Integer->Bool
parmod x = if  ( mod x 2 == 0   ) then ( True ) else ( False )
```

Otra resolución:

¿Se podrá escribir esta función parmod de otra forma, mucho más sintética, con menos palabras? Vemos que en la sentencia condicional, las acciones 1 y 2, esto es, lo que está después del then y del else son también expresiones booleanas, que justamente coinciden con la expresión lógica.

Esto es, cuando mod x 2 == 0 es True, la respuesta es True.

Y cuando mod x 2 == 0 es False, la respuesta es False.

Entonces mod x 2 == 0 **ya nos proporciona** la salida que estamos buscando.

En resumen,

```
parmod2:: Integer-> Bool
parmod2  x =   (mod x 2 == 0)
```

Ejercicio: Implementar la función resto, en forma recursiva. (Sin utilizar la función "mod").

```
resto :: Integer -> Integer -> Integer
resto a b = if a >= b then resto (a-b) b else a
```

Ejercicio: Implementar la función cociente, en forma recursiva. (No utilizar la función "div").

```
cociente :: Integer -> Integer -> Integer
cociente a b = if a >= b then  (1 + cociente (a-b) b) else 0
```

Ejercicio: Implementar la función "sumacifra" que sume los dígitos de un número entero positivo de 3 cifras. En este caso hay que controlar que si el número ingresado no tiene 3 dígitos, aparezca un mensaje de error, del tipo "número incorrecto de cifras"

Una respuesta posible puede ser:

```
sumacifra :: Integer -> Integer
sumacifra  a  |  a < 0      = error "el numero ingresado es negativo"
              | a> 999   = error  "el numero ingresado tiene mas de 3 cifras"
              | otherwise = mod a 10 + div a 100 + div (mod a 100) 10
```

Hay que tener mucho cuidado porque si escribimos " el número ....." puede dar error en algunas versiones de Haskell por el tilde. Haskell 98 lo soporta pero Haskell 2012 no.

Ejercicio: Implementar la función "sumatuti" que sume los dígitos de un número entero positivo, con cualquier cantidad de cifras.

Una resolución se ha podido rescatar dentro de una vieja botella que estaba flotando en el mar,

```
sumatuti :: Integer -> Integer
sumatuti 0
sumatuti n = mod
```

pero parte de ella se ha borrado. Intenta reconstruirla.

Conjuntos definidos en forma inductiva.

Definición inductiva de conjuntos.

Un conjunto está definido inductivamente, o se dice también que se ha definido un conjunto por inducción, si se dan 3 reglas para su construcción:

Paso base: se especifica una colección inicial de elementos. No puede ser vacía. Tiene que haber por lo menos un elemento.

Paso recursivo (también llamado paso inductivo): se proporcionan reglas para la formación de nuevos elementos del conjunto a partir de los que ya se conocen.

Regla de exclusión (también llamada cláusula de clausura): no puede haber más elementos que aquellos nombrados en el paso base o generados por la aplicación del paso recursivo un número finito de veces.

Veamos otro ejemplo: vamos a definir al conjunto J.

1) Cláusula base: $3 \in J$

2) Cláusula inductiva: $x \in J \rightarrow (x+2) \in J$

3) Cláusula de clausura: los únicos elementos del conjunto J son los indicados en la cláusula base y los que son generados por la aplicación de la cláusula inductiva un número finito de veces.

Cómo la cláusula de clausura es muy similar en todas las definiciones, se acostumbra abreviar CC. ¿Cuáles son los elementos que están en el conjunto J?

El 3 pertenece al conjunto J por aplicación de la cláusula 1.

Por aplicación de la cláusula 2, como el 3 pertenece a J, el 5 pertenece a J.

Por aplicación de la cláusula 2, como el 5 pertenece a J, el 7 pertenece a J.

Por aplicación de la cláusula 2, como el 7 pertenece a J, el 9 pertenece a J.

Y así sucesivamente.

Entonces J es un conjunto que tiene infinitos elementos. Los primeros son 3, 5, 7, 9, 11, 13, 15.

Por supuesto que el conjunto J puede definirse de otras maneras.

Por ejemplo, J es el conjunto de los números naturales impares menos el 1.

Esta es otra definición de J, aunque esta última no es una definición inductiva.

El conjunto J tiene infinitos elementos. Si suponemos (demostración por absurdo) que tiene una cantidad finita de elementos, siempre se puede encontrar un elemento más, por aplicación de la cláusula 2.
Pero algo muy importante, es que cada elemento del conjunto J se puede obtener mediante la aplicación de las cláusulas 1 y 2 un número finito de veces.

En resumen: El conjunto **J** tiene **infinitos** elementos. **Cada uno** de sus elementos se obtiene de aplicar la cláusula 1 o 2 un número **finito** de veces.

Los conjuntos definidos por inducción **no siempre** tienen que tener infinitos elementos.

Vamos a definir al conjunto P.

1) Cláusula base: $2 \in P,\ 17 \in P,$

2) Cláusula inductiva: $x \in P \rightarrow x \in P$

3) Cláusula de clausura: CC

En este caso, el paso base indica que los números 2 y 17 están en el conjunto P. Y el paso inductivo dice que si x está en P, entonces x está en P. Dicho de otra forma, el paso inductivo no agrega nuevos elementos al conjunto P. Entonces $P = \{2, 17\}$

Un conjunto definido en forma inductiva, con sólo 2 elementos, muy utilizado en Ciencias de la Computación, es el conjunto **Bool**. Vamos a definir al conjunto Bool en forma inductiva.

1) Cláusula base: $True \in Bool,\ False \in Bool,$

2) Cláusula inductiva: (no hay)

3) Cláusula de clausura: CC

El conjunto es entonces: $Bool = \{True,\ False\}$

Otra definición del mismo conjunto Bool puede ser:

1) Cláusula base: $True \in Bool$

2) Cláusula inductiva: $x \in P \rightarrow (-x) \in P$

3) Cláusula de clausura: CC

(-x) significa (No x).

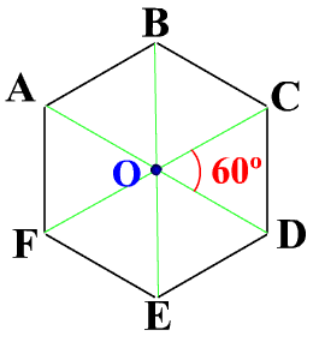
Otro ejemplo:

Sea un hexágono regular de centro O cuyos vértices los llamaremos A, B, C, D, E, F. Una definición inductiva del conjunto de sus vértices V es:

1) Cláusula base:	$D \in V$
2) Cláusula inductiva:	$x \in P \rightarrow R_{O,60^o}(x) \in P$
3) Cláusula de clausura:	CC

En esta definición, $R_{O,60^o}(x)$ significa la rotación de centro O y ángulo 60º aplicada al vértice x.

La rotación de centro O y ángulo 60º aplicada al vértice D nos proporciona el vértice C (la rotación de 60º es anti horaria si es positivo). Luego aplicada a C da B, luego A, luego F, E y al final, de nuevo, D. Aunque el proceso tenga la cantidad de pasos que se quiera, miles de pasos, la cantidad de elementos del conjunto V es 6. Entonces, V = {A, B, C, D, E, F}
Y otra vez tenemos un conjunto finito definido en forma inductiva.



Ejercicio 1:

1) Cláusula base:	$4 \in W$
2) Cláusula inductiva:	$x \in W \rightarrow (10x+4) \in W$
3) Cláusula de clausura:	CC

Escribir los primeros 6 elementos del conjunto W.

Ejercicio 2:

1) Cláusula base:	$3 \in H$
2) Cláusula inductiva:	$x \in H \rightarrow (2x+5) \in H$
3) Cláusula de clausura:	CC

Escribir los primeros 6 elementos del conjunto H. ¿$262139 \in H$?

Ejercicio 3:

1) Cláusula base:	$3 \in H,\ 7 \in H$
2) Cláusula inductiva:	$(x \in H)$ y $(x' \in H) \rightarrow (x + x') \in H$
3) Cláusula de clausura:	CC

Escribir los primeros elementos del conjunto H. ¿$2010 \in H$? ¿$2011 \in H$? ¿$2012 \in H$? ¿Es el conjunto H el conjunto de todos los números enteros positivos? Justifique.

Otro ejemplo:

1) Cláusula base:	$A \in P,\ G \in P,\ T \in P,\ C \in P$
2) Cláusula inductiva:	$(x \in P)\ y\ (x' \in P) \rightarrow (xx') \in P$
3) Cláusula de clausura:	CC

La Adenina, Guanina, Timina y Citosina son las bases nitrogenadas que forma el ADN. Su secuencia es lo que determina el ADN de los diferentes seres vivos. El genoma de algunas pocas proteínas fue descubierto gracias a un estudio basado en una estrecha relación entre biología e informática.
La síntesis de proteínas es un proceso biológicamente inductivo.

Para más información, buscar: PEDECIBA, BIOINFORMATICA.

NATURALES

Definición inductiva del conjunto de los números naturales.

1) Cláusula base:	$0 \in N$
2) Cláusula inductiva:	$n \in N \leftrightarrow (S\ n) \in N$
3) Cláusula de clausura:	CC

¿Qué significa esta definición?

- ✓ La cláusula base establece que 0 es un elemento del conjunto de los números naturales (N)
- ✓ La cláusula inductiva establece que siempre que x sea un elemento del conjunto N, (S x) es un elemento del conjunto N.
- ✓ S es el constructor utilizado en haskell y se lee “siguiente” o “sucesor”

El 0 pertenece al conjunto N por aplicación de la cláusula 1. Por aplicación de la cláusula 2, como el 0 pertenece a N, el (S 0) pertenece a N. Por aplicación de la cláusula 2, como el (S 0) pertenece a N, el (S (S 0)) pertenece a N. Por aplicación de la cláusula 2, como el (S (S 0)) pertenece a N, el (S (S (S 0))) pertenece a N. Entonces, en N tenemos los elementos 0, (S 0), (S (S 0)), (S (S (S 0))), ...
Otra vez tenemos un conjunto infinito, pero cada elemento se obtiene de aplicar las cláusulas 1 y 2 un número finito de veces.

De ahora en adelante, para hablar un poco más rápido, al (S 0) vamos a darle otro nombre. Por ejemplo, podemos llamarle "1", al (S (S 0)) podemos llamarle "2", (S(S(S 0))) será el 3, ...

Ejercicio 4: Definir en forma inductiva el conjunto de los números enteros que no son múltiplos de 3.

Ejercicio 5: i) Definir en forma inductiva el conjunto de los números naturales pares.
ii) Lo mismo con los impares.

Ejercicio 6: i) Definir en forma inductiva el conjunto de los números enteros pares.
ii) Lo mismo con los impares.

Aclaración: en general, cuando hablamos de "enteros" es porque expresamos los números en la forma habitual, esto es: 1, 2, -4, 27, -12, etc.
Cuando hablamos de "Naturales" estamos considerando la definición inductiva de los mismos, por lo que cuando queremos referirnos al 1, escribimos (S Z), si estamos pensando en el 2, (S (S Z)).

Otros conjuntos definidos en forma inductiva: Lenguajes.

Definición: Un **alfabeto** es un conjunto finito (o por lo menos numerable) de símbolos, números o caracteres. Por ejemplo, M= {a,b,c,d,e} es un alfabeto con 5 letras.
Una **palabra** sobre un alfabeto es una secuencia, o cadena, de cero o más elementos del alfabeto, concatenados. A la palabra vacía la llamaremos ε.

Con nuestro alfabeto M definido más arriba, y sin reglas definidas por ahora, podemos decir que podemos formar las palabras "ba", "cae", "baca", "ababa", " ". Esta última palabra, la palabra que no tiene letras, recordemos que se llama ε. La palabra "matemática" no se puede formar porque nos faltan muchas letras.

Lenguajes definidos en forma inductiva.

Sea S= { a, b, c }.
Con los 3 elementos de S podemos hacer un número infinito de palabras. A este conjunto de las palabras que podemos hacer con las letras de S vamos a llamarlo S*.
S*= {p / p es una palabra con letras de S}
S es un conjunto con 3 letras.
S* es el conjunto de <u>todas</u> las palabras que se pueden formar con las letras de S

De las infinitas palabras, vamos a elegir un subconjunto L que tenga determinadas características.

Sea S= { a, b, c } y $L \in S^*$
1) Cláusula base: $a \in L$
2) Cláusula inductiva: $x \in L \rightarrow bxb \in L$
3) Cláusula de clausura: CC

Este conjunto L es un lenguaje sobre S. ¿Qué palabras contiene L?

Por la cláusula 1, "a" es una palabra que está en L. Como "a" está en L, entonces, aplicando la cláusula 2, obtenemos que "bab" está en L. Como "bab" está en L, entonces, aplicando la cláusula 2, obtenemos que "bbabb" está en L. Como "bbabb" está en L, entonces, aplicando la cláusula 2, obtenemos que "bbbabbb" está en L.

¿Qué podemos decir del largo de todas las palabras en este ejemplo? ¿En L está la palabra vacía?

Respuestas a algunos ejercicios del capítulo:

Conjuntos definidos por inducción.

Ejercicio 1: Los primeros elementos del conjunto W son: 4, 44, 444, 4444, 44444,....

Ejercicio 2: Si. ¿Tienes una calculadora a mano? Utiliza la tecla "Ans" varias veces.....

¿Con que cifra(s) terminan todos los elementos del conjunto?

Ejercicio 3: El conjunto H es igual que los Naturales menos algunos pocos, al principio, que faltan. Hay que demostrarlo por inducción completa, separando en varios casos.....

Por ejemplo, en el conjunto H están el 3 y el 7 seguros, porque son los elementos bases.

Luego estarán todos los que se deducen de la regla número 2.

Como están en H el 3 y el 3, estará el 6.

Como están en H el 7 y el 7, estará el 14.

Con el 3 y el 7, se "forma" el 10.

Con el 6 y el 3 se forma el 9;

Vamos a intentar seguir un poco mas ordenados:

9 = 3+6

10= 3+7

11 = me parece que no se puede formar...

12= 6+6

13=6+7

14=7+7

15=12+3

16=13+3

17=14+3

18=15+3

y así podemos seguir. Parece entonces que de aquí en adelante se podrán formar todos los demás números naturales. ¿Cómo se puede DEMOSTRAR esto?

No es sólo mostrarlo, hay que "**demostrarlo**". Lo haremos aplicando "inducción completa". Intentaremos demostrar que cualquier número natural, mayor o igual que 12, se puede escribir como una suma de varios tres y varios sietes.

Paso Base: 3 + 3 + 3 + 3 =12 se cumple.
Paso Inductivo: si un número cualquiera "n" se puede escribir como suma de varios 3 y varios 7, entonces el número siguiente, "n+1", también.

Por ejemplo (no sirve de demostración, sólo es un ejemplo para fijar ideas):
El número 838 se puede escribir como: 279*3 + 1 = 838
Pero esto no nos sirve porque NO es una suma de números 3 y 7.
Hay 279 veces el 3, pero el "1" no nos sirve.
Hagamos aparecer un 7, empleando algunos "trucos de magia".

838 = 279*3 + 1 = 278*3 + 3 + 1 = 277*3 + 3 + 3 + 1 = 277*3 + 7
Esta descomposición no es única. Se puede hacer de muchas formas.
¿Y cómo haremos con el 839? Este es una unidad mayor que el 838.
839 = 277 * 3 + 7 + 1 = 275*3 + 3 + 3 + 7 + 1 = 275*3 + 7*2 y quedó probado.

Volvamos ahora a la prueba formal para un número n cualquiera.

Hipótesis: a * 3 + b * 7 = n esta igualdad suponemos que se cumple, es un dato, es cierta.
Tesis: a' * 3 + b' * 7 = n+1 esta igualdad hay que probar que es cierta.
Demostración: para empezar, no sabemos si "a" o "b" son números grandes o no. Pudiera ser que alguno de ellos sea 0.
Para pasar de "n" a "n+1" hay que sumarle "1". (Como se dice habitualmente, chocolate por la noticia). Esa unidad extra se puede obtener de varias formas. Si el número "a" es por lo menos 2, le podemos quitar 2 unidades, con lo que habremos disminuido en 6 al sacarle "dos 3", y al agregar un 7 obtenemos una unidad más.

(Hipótesis) a * 3 + b * 7 = n $\Rightarrow$ (a-2) * 3 + (b+1) * 7 = n+1 lqqd (lo que queda demostrado). Hemos llegado a la tesis.
Pero si el número "a" no es mayor que 2, hay que hacerlo al revés, con "b".

(Hipótesis) $a * 3 + b * 7 = n \Rightarrow (a+5) * 3 + (b-2) * 7 = n+1$ lqqd (lo que queda demostrado). Hemos llegado a la tesis.

Si hay dudas, podemos hacer operaciones: vamos a aplicar la propiedad distributiva:

$(a+5) * 3 + (b-2) * 7 = 3a + 15 + 7b - 14 = 3a + 7b + 1 = n + 1$ porque $3a+7b = n$.

Esta vez al sacar dos números 7 hemos disminuido en 14 unidades. Si agregamos 5 veces el 3, aumentamos en 15, con lo cual, en resumen, aumentamos en 1 el total.

Ejercicio 4: Definir en forma inductiva el conjunto de los números enteros que no son múltiplos de 3.

1) Cláusula base: $1 \in H$

$2 \in H$

2) Cláusula inductiva: $x \in H \rightarrow x + 3 \in H$;

$x \in H \rightarrow x - 3 \in H$

3) Cláusula de clausura: C C

Si hay dudas, buscar algunos elementos de H, aplicando las cláusulas 1 y 2.

Ejercicio 5: i) Definir en forma inductiva el conjunto de los números naturales pares.

1) Cláusula base: $0 \in H$

2) Cláusula inductiva: $x \in H \rightarrow S(S(x)) \in H$

3) Cláusula de clausura: C C

ii) Lo mismo con los impares.

1) Cláusula base: $(S\ Z) \in H$

2) Cláusula inductiva: $x \in H \rightarrow S(S(x)) \in H$

3) Cláusula de clausura: C C

Observación: la única diferencia en las definiciones de los pares e impares es la cláusula base.

Ejercicio 6: i) Definir en forma inductiva el conjunto de los números enteros pares.

1) Cláusula base: $0 \in H$

2) Cláusula inductiva: $x \in H \rightarrow x + 2 \in H$

$x \in H \rightarrow x - 2 \in H$

3) Cláusula de clausura: C C

ii) Lo mismo con los impares.

1) Cláusula base:	$1 \in H$
2) Cláusula inductiva:	$x \in H \rightarrow x + 2 \in H$
	$x \in H \rightarrow x - 2 \in H$
3) Cláusula de clausura:	C C

LISTAS

Conjuntos definidos por inducción: Listas. Funciones sobre Listas.

Ahora introduciremos una estructura matemática denominada listas o secuencias, con la cual se pueden modelar muchas realidades. Una lista es una ordenación de elementos de un cierto conjunto, una lista puede ser vacía.

Ejemplos de listas diferentes, con elementos del conjunto {3, 4, 7}

1) [3, 4, 7, 7]
2) [3, 4, 7, 7, 7]
3) [4, 3, 3, 3]
4) [3, 4, 7, 7, 3]
5) [3, 4, 7]
6) [3, 3, 3, 3, 3, 3, 3]

En las listas es importante el orden de los elementos que la integran. Además, los elementos repetidos también importan.

NOTACIÓN: Para indicar listas, lo haremos señalando entre paréntesis rectos sus elementos. Por ejemplo, la lista que contiene el 1, el 1 y el 5 será [1, 1, 5]. Esta lista es diferente de la siguiente: [5, 1, 1]

La lista vacía, esto es, la lista que no tiene elementos, la simbolizaremos [ ].

Mas ejemplos: con los elementos del conjunto A = {3}, ¿cuántas listas se pueden formar?

Respuesta:

1) [ ]
2) [3]
3) [3, 3]
4) [3, 3, 3]
5) [3, 3, 3, 3]

Si el conjunto A no es el conjunto vacío las listas que se pueden formar tomando elementos de A son infinitas. Cada lista tiene una cantidad finita de elementos, pero la cantidad de listas que se pueden formar con los elementos de un conjunto dado A, es un número infinito

Sea A un conjunto cualquiera. Definiremos el conjunto de todas las listas que se pueden construir con elementos de A por inducción. Llamaremos a este conjunto **list A**.

Definición inductiva del conjunto **list A**

1) La lista vacía pertenece a **list A**

2) Si tenemos un elemento de **A** y una lista que pertenece a **list A**, podemos formar una nueva lista agregándole al principio de dicha lista ese nuevo elemento.

Por ejemplo, si tenemos la lista [5,7,2] y el elemento a agregar es el 4, entonces la nueva lista será 4:[5,7,2] = [4,5,7,2]

3) Las únicas maneras de formar listas es aplicar las reglas 1 y/o 2 un número finito de veces.

Con notación matemática:

1. nil $\in$ list A
2. (cons a x) $\in$ list A si a $\in$ A y x $\in$ list A
3. CC

¿Qué significa esta definición?

- ✓ El axioma nil $\in$ list A establece que nil es un elemento del conjunto de listas que se pueden formar con elementos de A, es decir la lista vacía es una lista.
- ✓ La cláusula (cons a x) $\in$ list A si a $\in$ A y x $\in$ list A establece que siempre que a sea una elemento de A y x una lista de elementos de A, (cons a x) es una lista de elementos de A.

Un ejemplo con esta notación: La lista [2,5,1] , ¿cómo se formó?

Primero, le agregamos el 1 a la lista vacía, luego le agregamos el 5 a la lista que sólo contenía el 1, y para terminar le agregamos el 2 a la lista que contiene al 5 y al 1.

Para escribirlo en forma teórica:

- ✓ nil $\in$ list A
- ✓ 1 $\in$ A y nil $\in$ list A $\Rightarrow$ cons 1 nil $\in$ list A
- ✓ 5 $\in$ A y (cons 1 nil) $\in$ list A $\Rightarrow$ cons 5 (cons 1 nil) $\in$ list A
- ✓ 2 $\in$ A y (cons 5 (cons 1 nil)) $\in$ list A $\Rightarrow$ cons 2 (cons 5 (cons 1 nil)) $\in$ list A
- ✓ [2,5,1] es igual que 2:5:1:[ ] y es lo mismo que (cons 2 (cons 5 (cons 1 nil)))

Veamos ahora la misma definición, pero con la notación usada en Haskell.

1) Cláusula base.	[ ] $\in$ **list A**
2) Cláusula inductiva.	Si b $\in$ A y x $\in$ **list A** $\Rightarrow$ b:x $\in$ **list A**
3) Cláusula de clausura.	CC

La cláusula o regla 1 dice que la lista vacía es una lista.
La cláusula o regla 2 dice que si **b** es un elemento de **A** y **x** es una lista, le podemos agregar a la lista **x** un nuevo elemento, el **b**, al comienzo, formándose una nueva lista, **b:x**, que tiene un elemento más que **x**.
La cláusula o regla 3 es la cláusula de clausura, y lo que dice es que solamente podemos formas listas aplicando las cláusulas o reglas 1 y 2 un número finito de veces.

Ejemplo: Supongamos que tenemos el conjunto A = {1, 2, 7}
Por aplicación de la regla número 1, la lista vacía pertenece a **list A.** Como el número 2 pertenece al conjunto A y la lista vacía pertenece a **list A**, entonces aplicando la segunda regla, se le puede agregar a la lista vacía el 2. Queda entonces 2: [ ] = [2]
Como el número 1 pertenece al conjunto A y la lista [2] pertenece a **list A**, entonces aplicando la segunda regla, se le puede agregar a dicha lista el 1. Queda 1: [2] = [1,2]
Como el número 7 pertenece al conjunto A y la lista [1,2] pertenece a **list A**, entonces aplicando la segunda regla, se le puede agregar a dicha lista el 7. Queda 7: [1,2] = [7,1,2]
Como el número 1 pertenece a A y la lista [7,1,2] pertenece a **list A**, entonces aplicando la segunda regla, se le puede agregar a dicha lista el 1. Queda 1: [7,1,2] = [1,7,1,2]
Como el número 1 pertenece a A y la lista vacía pertenece a **list A**, entonces aplicando la segunda regla, se le puede agregar a dicha lista el 1. Queda 1: [ ] = [1]
Y así seguimos construyendo más elementos de list A.

Ejercicio: Escribe las siguientes expresiones en Haskell.

1) [ ]
2) 5: [ ]
3) 3: [5]
4) 3:[3,5]

5) 3:3:5:[ ]

6) ‘a’:[ ]

7) ‘g’:[‘a’,’t’,’o’]

8) ‘g’:”ato”

9) True: [False, False]

Una palabra es una cadena de caracteres, una lista de caracteres.

Notación en Haskell:

- ✓ [a] es una lista con exactamente un elemento
- ✓ [a, b] es una lista con exactamente 2 elementos
- ✓ [a, b, c] es una lista con exactamente 3 elementos
- ✓ (a) es una lista con cualquier cantidad de elementos; podría ser también la lista vacía.
- ✓ (a:x) es una lista donde **a** es el primer elemento de la lista y **x** el resto de la lista; tiene por lo menos un elemento, el a.
- ✓ (a:b:x) es una lista donde **a** es el primer elemento de la lista, **b** es el segundo y **x** el resto de la lista; tiene por lo menos 2 elementos.

Ejemplo: si escribimos en el editor de textos:

(a:x) referido a la lista [7,2,4,1] significa que a es 7 y que x es el resto, esto es, [2,4,1]

(a:x) referido a “manzana” significa que a es la ‘m’ y que “anzana” es x.

Funciones definidas por recursión sobre listas

Ejemplo 1: Queremos definir una función, llamada **largo**, que nos proporcione el largo de una lista de números enteros. Por ejemplo, si digitamos **largo [6, 2, 3, 1]** esperamos obtener un 4, porque esta lista tiene 4 elementos.

largo [True, True, False] = 3; **largo "cuatro"** = 6, **largo [6,2,3,1]**= 4

Observación: largo [6, 2, 3,1] = 1 + largo [2,3,1]

Generalizando**: largo (a:x) = 1 + largo (x)**, donde x es una lista cualquiera.

Ya podemos *"intentar"* hacer la función **largo**. (Esta función, como todas las funciones, para implementarla en Haskell, hay que escribirla en el **editor de texto**).

```
largo :: [Integer]->Integer
largo (a:x) = 1+ largo (x)
```

Antes de probarla con Haskell intentaremos probar la función en el papel. A esto se le llama escribir la "secuencia de cómputos".

Secuencia de cómputos para largo [8,5,0,9]

largo [8,5,0,9] = 1 + largo [5,0,9]

largo [5,0,9] = 1 + largo [0,9]

largo [0,9] = 1 + largo [9]

largo [9] = 1 + largo [ ]

largo [ ] = … No lo podemos completar, ya que no hemos indicado en ningún lado cuanto debería valer el largo de la lista vacía. Si lo probamos en Haskell nos dará un mensaje de error.

Falta definir el valor de la función largo cuando aplicamos la lista vacía.

```
largo :: [Integer]->Integer
largo [ ] = 0
largo (a:x) = 1+ largo (x)
```

Ahora sí, está definida correctamente. Pruébalo.

Ejemplo 2: Ahora vamos a desarrollar la función **sumatodo** que se encargue de sumar todos los elementos de una lista. Por supuesto que tiene que ser una lista de números. En este ejemplo, usaremos números enteros.

Lo que queremos que haga la función es: sumatodo [3, 2, 7] = 12

Comenzamos haciéndonos preguntas……¿sumatodo [3, 2, 7] = sumatodo [2, 7] ?

La respuesta es no; a la derecha falta sumarle 3.

Entonces, sumatodo [3, 2, 7] = 3 + sumatodo [2, 7].

Tratando de generalizar, sumatodo (a:x)= a + sumatodo (x)

¿Podrías escribir la función, prolija, sin errores, para ejecutar en Haskell?

Ejemplo 3: Vamos ahora a investigar un poco cuando trabajamos con listas de caracteres.

'a' es un carácter, 'h' es otro, '4' es otro.

['a', 'b', 'c'] es una lista de caracteres; otra forma de escribirla es: "abc"

Prueba escribir ['g', 'a', 't', 'o'] en Haskell y veremos qué pasa luego de dar "enter".

Vamos a implementar la función **contar** que nos sirva para contar las letras de una palabra o de muchas palabras. Esto es, si funciona bien, **contar** "gato" debería ser 4.

Esta función es casi igual que la función largo. Sólo se diferencia en que el dominio de esta función es una lista de caracteres, [Char], también llamada String.

```
contar :: [Char]->Integer
contar [ ] = 0
contar (a:x) = 1+ contar  (x)
```

Notación: [Char] es lo mismo que String. Prueba hacerlo de las 2 formas.

Vamos ahora a probarlo.

contar "perro" = 5

contar ['g', 'a', 't', 'o'] = 4

contar "dos manos" = …… ¿Qué respuesta esperas?

Ejemplo 4: Ahora vamos a desarrollar la función "**copiar**". Lo que buscamos es cortar en pedacitos una lista y luego volverla a armar en otro sitio, pero que quede igualita. O sea, dicho de otra manera, lo que buscamos es copiar una lista, desarmándola y luego volviéndola a armar.

Por ejemplo, si quisiéramos copiar la lista [1,2,3] lo que hacemos primero es sacarle el 1, y nos queda la lista [2,3]. Luego le sacamos el 2, y nos queda la lista [3]. Finalmente le sacamos el 3 y nos queda la lista vacía, [ ].

Ahora que ya la cortamos en pedacitos, vamos a armarla de nuevo. A la lista vacía le agregamos el 3, y obtenemos la lista [3]. Ahora le agregamos el 2, y nos queda la lista 2: [3] , que es la lista [2,3]. Y ahora le agregamos el 1 y nos queda: 1: [2,3] que es lo mismo que [1,2,3]

Para pensarlo de una forma general, para copiar la lista [1,2,3], le sacamos el primer elemento, el 1, y luego se lo agregamos a la lista que quedó. Esto es, [2,3]

Para copiar (a:x) habrá que sacarle el primer elemento, esto es, el “a” , con lo que queda la lista (x). Luego, agregarle a la lista (x) el elemento “a”. Y hay que hacerlo recursivo, sino sólo va a ocurrir un paso, y queremos que suceda en todos los pasos, en toda la lista.

Entonces el paso recursivo será: copiar (a:x) = a: copiar (x)

Vamos ahora a armar la función, en forma prolija:

```
copiar:: [Integer] -> [Integer]
copiar [ ] = [ ]
copiar (a:x) = a: copiar (x)
```

Para probar la función con Haskell, escribimos:

copiar [4,7,1] y la respuesta será [4,7,1]

Tal **parece** que la función “no hizo nada”. En realidad, hizo mucho !!

Veamos la secuencia de cómputos, para entenderlo un poco más.

i) copiar [4,7,1] = 4: copiar [7,1] aplicando la última cláusula de la función.

ii) copiar [7,1] = 7: copiar [1] aplicando la última cláusula de la función.

iii) copiar [1] = 1: copiar [ ] aplicando la última cláusula de la función.

iv) copiar [ ] = [ ] aplicando la segunda cláusula de la función.

Entonces, ahora vamos de abajo hacia arriba, sustituyendo.

iii) copiar [1] = 1: copiar [ ] = 1: [ ] = [1]

ii) copiar [7,1] = 7: copiar [1] = 7: [1] = [7,1]

i) copiar [4,7,1] = 4: copiar [7,1] = 4: [7,1] = [4,7,1]

Y ya quedó copiada.

Ejemplo 5: Tenemos ahora una función, que aparentemente es parecida a la anterior. Por ahora la llamaremos equis, porque es una incógnita, en el sentido que no sabemos qué es lo que hace.

```
equis:: [Integer] -> [Integer]
equis [ ] = [ ]
equis (a:x) = equis (x)
```

Para investigar y deducir que es lo hará esta función, podemos aplicarla a una lista pequeña, digamos [4,7,1]. Con la secuencia de cómputos veremos que sucede.

i) equis [4,7,1] = equis [7,1] aplicando la última cláusula de la función.

ii) equis [7,1] = equis [1] aplicando la última cláusula de la función.

iii) equis [1] = equis [ ] aplicando la última cláusula de la función.

iv) equis [ ] = [ ] aplicando la segunda cláusula de la función.

Entonces, ahora vamos de abajo hacia arriba, sustituyendo.

iii) equis [1] = equis [ ] = [ ]

ii) equis [7,1] = equis [1] = [ ]

i) equis [4,7,1] = equis [7,1] = [ ]

Conclusión: esta función borra la lista. Claro, si en la cláusula inductiva se elimina el primer elemento de la lista, lo está haciendo es eliminar el primer elemento **en cada paso**. Si siempre elimina el primer elemento, y esta orden la repetimos, al final nos quedamos sin elementos. Es decir, obtenemos la lista vacía, [ ].

Ahora que ya sabemos lo que hace esta función, vamos a llamarla "borrar".

```
borrar:: [Integer] -> [Integer]
borrar [ ] = [ ]
borrar (a:x) =  borrar (x)
```

Repaso.

copiar:: [Integer] -> [Integer]	borrar:: [Integer] -> [Integer]
copiar [ ] = [ ]	borrar [ ] = [ ]
copiar (a:x) = a: copiar (x)	borrar (a:x) = borrar (x)

Aparte del nombre, que podría ser cualquiera, estas dos funciones sólo difieren en 1 letra y 2 puntos. Entre copiar toda una lista y borrarla toda la diferencia es, aparentemente, muy muy pequeña. Hay que cuidar mucho, muchísimo, los detalles.

Observación : el nombre de la función no es importante.

Si en algún ejercicio estamos utilizando la función llamada "verde" y queremos que en algún momento copie una lista, lo haremos escribiendo verde (n:p) = n:verde (p). Aquí le llamamos "n" al primer elemento de la lista y "p" al resto de ella.

Y si queremos borrar todo, escribiremos verde (n:p) = verde (p).

Esto, que es muy importante, lo usaremos en el futuro. (futuro = ejemplo 6)

Ejemplo 6: Ahora que ya sabemos cómo copiar y cómo borrar una lista, podemos intentar funciones más interesantes.

Tenemos una lista de números enteros positivos y queremos eliminar todos los números 3 que aparezcan. Llamaremos a esta función sacatres.

Esto es, debería suceder que: sacatres [1,2,3,4,5,6] = [1,2,4,5,6]

Además, sacatres [1, 3, 2, 3, 4, 3, 3, 3, 3] = [1, 2, 4]

En este caso, partimos de una lista y obtenemos otra lista.

¿Cómo trabajará esta función?

Para cada uno de los elementos de la lista, hay que fijarse si es el número 3 o no. Si es el número 3, lo eliminamos, lo borramos. Si no es el 3, lo dejamos, lo copiamos. O sea, a veces copiamos y a veces borramos.

Ya sabemos cómo fabricar las funciones borrar y copiar, pero ahora tenemos que seleccionar cuando aplicar una y cuando aplicar la otra.

Veamos: si es 3, se elimina; sino, se queda.

Vamos a redactarlo de nuevo: si es 3, se borra; sino, se copia.

Ejercicio: Defina la función sacatres mencionada en el ejemplo anterior.

Ejemplo 7: En este último ejemplo, intentaremos hacer una función que nos permita dejar las primeras 5 letras de un texto. Vamos a llamar a esta función "cinco".

Quisiéramos que haga esto:

cinco "otorrinolaringologo" = "otorr"

cinco "Pepe" = "Pepe"

Si la palabra tiene 5 letras o menos, se mostrará toda la palabra.

Aquí tenemos que "copiar" y simultáneamente ir contando las letras; cuando lleguemos a 5 letras dejamos de copiar.

Se puede diseñar la función de muchísimas formas. Veamos una.

```
cinco :: [Char]->[Char]
cinco (x) = pepe (x) 5
pepe :: [Char] ->Integer -> [Char]
pepe (x) 0 =  [ ]
pepe (a:x) n = a:pepe (x) (n-1)
```

Utilizamos una función auxiliar llamada pepe. Hay que usar una función auxiliar para poder introducir una nueva variable, n, que posibilita contar. Hay que probarla. Investiga que sucederá si la probamos así:

cinco "otorrinolaringologo"

cinco "gato"

cinco "perro"

Si nuestra función trabaja bien con el otorrinolaringologo y con el perro, pero no le gusta el gato, habrá que hacerle una pequeña corrección. Con los gatos funciona bien, pero con un gato no. Con los peces también funciona bien, pero con un pez tampoco……

Ejercicio: Corrija la función del ejemplo anterior para que sea correcta en todos los casos.

En resumen, hay un error……

Respuesta a algunos ejercicios del capítulo: Listas

Ejemplo 2:

```
sumatodo :: [Integer]->Integer
sumatodo [ ] = 0
sumatodo (a:x) = a + sumatodo x
```

Ejemplo 3: contar "dos manos" = 9, porque el espacio es un carácter y se cuenta.

Ejemplo 6:

Primero, un repaso:

```
copiar:: [Integer] -> [Integer]
copiar  [ ] = [ ]
copiar (a:x) = a: copiar (x)
```

```
borrar:: [Integer] -> [Integer]
borrar  [ ] = [ ]
borrar (a:x) =  borrar (x)
```

Función sacatres: a veces copia y a veces borra.

```
sacatres:: [Integer] -> [Integer]
sacatres [ ] = [ ]
sacatres (a:x) =  if  (a==3) then (sacatres (x)) else (a:sacatres (x))
```

Ejemplo 7:

"Peces" tiene 5 letras y funcionaba bien. "Pez" con 3 letras no funcionaba bien.

La corrección es indicarle a la función auxiliar lo que tiene que hacer si la lista se queda sin elementos. Esto es, que sucede con la lista vacía.

```
cinco :: [Char]->[Char]
cinco (x) = pepe (x) 5
pepe :: [Char] ->Integer -> [Char]
pepe (x) 0 =  [ ]
pepe [ ] n = [ ]                    ------ esta es la línea que faltaba
pepe (a:x) n = a:pepe (x) (n-1)
```

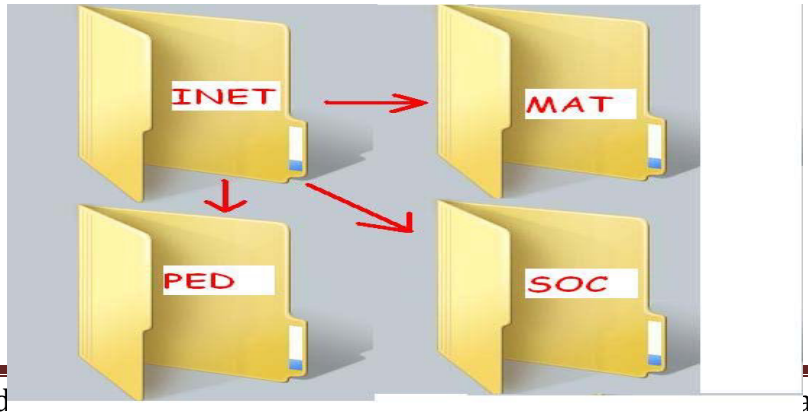
Árboles

Una de las estructuras de datos más importantes en programación es el árbol. Pueden usarse los árboles para representar la información en una estructura jerárquica. Los árboles pueden procesarse en forma recursiva y son muy adaptables a pruebas matemáticas. El estudio de árboles ilustra las conexiones entre varios temas de la matemática discreta y ofrece oportunidades para aprovechar la matemática formal en la programación práctica. La idea de estructura jerárquica es muy usada en la práctica. Por ejemplo, los libros son a menudo organizados como una sucesión de capítulos cada uno de los cuales son una sucesión de secciones que puede tener subdivisiones, y así sucesivamente. Una empresa puede organizarse como las colecciones de unidades comerciales cada uno de las cuales pueden tener varias secciones. Las secciones, a su vez, pueden tener secciones múltiples, y así sucesivamente.

El software es organizado como una colección de módulos cualquiera que pueden constituirse de varios submódulos, con el nivel de refinamiento que los diseñadores encuentren apropiado. En cierto nivel, los módulos se expresan en unidades básicas como los objetos, los métodos, o procedimientos.

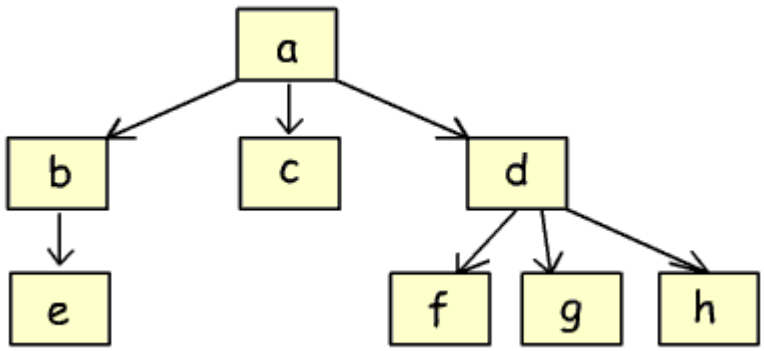
Antes de ver una definición formal, vamos a mirar ejemplos, con su nomenclatura habitual, que seguramente ya conoces.

Lucía se inscribió en el INET para cursar algunas asignaturas de primero: matemática, pedagogía y sociología. Antes de empezar el año, para organizarse, abre una carpeta nueva, llamada INET y dentro de ella, tres carpetas más, una para cada asignatura.



Esto, que seguramente es conocido para ti, es un árbol. Aunque no sabemos si dentro de alguna carpeta hay materiales o no, tenemos un árbol. Además, la carpeta INET, con todo su contenido, forma parte de alguna otra carpeta, quizás llamada “estudio” o “profesorado” o quizás esté “colgada” directamente de la raíz del disco duro “C”.
Entonces, en este capítulo, mantendremos muchas de estas palabras y cambiaremos otras, pero el significado ya lo conocemos.Sólo falta darle formalidad.

Veamos ahora, con un ejemplo, algunos nombres que usaremos.
El siguiente diagrama representa un árbol de caracteres, letras.



- ➔ En este árbol hay 8 nodos, uno por cada letra que vemos.
- ➔ El tamaño de este árbol es, entonces, 8.
- ➔ La raíz es el nodo 'a'.
- ➔ Este árbol tiene altura 3, porque tiene 3 niveles.
- ➔ En el nivel 0, tenemos sólo la 'a'.
- ➔ En el nivel 1 tenemos 'b', 'c' y 'd'.
- ➔ En el nivel 2 tenemos 'e', 'f', 'g' y 'h'.
- ➔ Las hojas son los nodos que no tiene hijos (ramificaciones hacia abajo).
- ➔ Las hojas son 'e', 'c', 'f ', 'g' y 'h'
- ➔ La raíz tiene 3 hijos, el nodo 'b' tiene un sólo hijo, el nodo 'c' no tiene hijos, etc.

Veamos, entonces, algunas definiciones:

- **Nodo**: son los elementos del árbol.
- **Raíz del árbol**: todos los árboles que no está vacíos tiene un único nodo raíz. Todos los demás elementos o nodos derivan o descienden de la raíz.
- **Nodo hoja**: es aquel que no contiene ningún subárbol. A cada nodo que no es hoja se le asocia uno o varios subárboles llamados descendientes o hijos.
- Cada nodo tiene asociado un sólo antecesor o **ascendiente** llamado padre (excepto la raíz que no tiene padre).
- **Tamaño** de un árbol: es el número de nodos.
- Cada nodo tiene asociado un **número de nivel** que se determina por la longitud del camino desde la raíz al nodo específico.
- La **altura** o profundidad de un árbol es el nivel más profundo más uno.

Ahora, quizás, estemos en condiciones de empezar a entender la siguiente definición de árbol. Recordemos que nil es el nombre que le damos a una lista vacía. Por extensión, también llamaremos nil al árbol vacío.
En la primera definición, le llamaremos "e" al elemento que se encuentra en el nodo y serán a_1, a_2, a_n, los subárboles.

Definición:

Hemos visto varias definiciones inductivas: conjuntos y listas. Ahora vamos a ver la definición inductiva de árboles.

➔ Un árbol o bien es un árbol vacío o es un nodo junto con una sucesión de árboles.

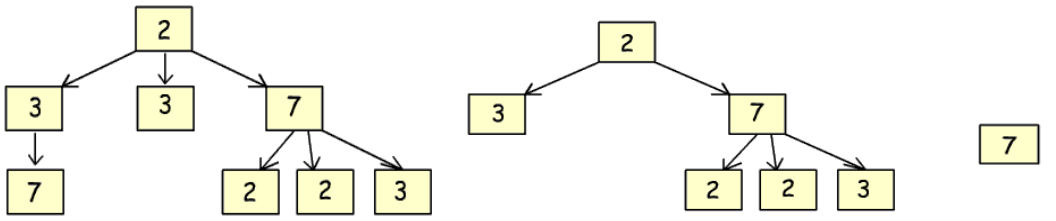
Sea A un conjunto cualquiera:

1. nil $\in$(Árbol A)
2. (cons e a1 a2… an) $\in$ (Árbol A) sii (e $\in$A) y (a1, a2,… ,an $\in$ (Árbol A))
3. C C

La cláusula base nos indica que nil (árbol vacío) es un elemento del conjunto (Árbol A). La existencia del árbol vacío se acepta como un axioma. El término "nodo" no se define, y la existencia de nodos para construir árboles también se toma como un axioma.

Lo importante entonces es que "e", el elemento del nodo, pertenezca al conjunto A y que los subárboles, a1, a2, …, an, pertenezcan al conjunto (Árbol A) , que es el conjunto de todos los árboles que se pueden hacer a partir del conjunto A.

El conjunto **A** puede ser finito, por ejemplo, A = {2,3,7} , pero el conjunto **Árbol A,** conjunto de todos los árboles que se pueden formar a partir de los elementos del conjunto A, es infinito. Veamos ahora 3 elemento del conjunto Árbol A para este caso; van entonces 3 ejemplos de árboles construidos a partir del mismo conjunto A.

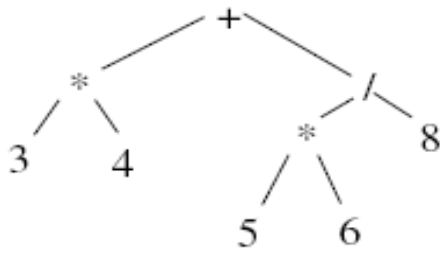


Árbol 1 Árbol 2 Árbol 3

Más ejemplos: se considera el árbol que representa la expresión aritmética:

(3 x4) + ((5 x 6)/8).

La raíz del árbol es el +. Cada sub árbol representa expresiones que describen los argumentos a ser agregados. Las hojas del árbol representan los números que aparecen en la expresión. El valor de la expresión puede calcularse trabajando desde las hojas a la raíz, mientras se van calculando los valores intermedios que corresponden a cada operador.



Muchos intérpretes de lenguajes de programación y compiladores se acostumbran a representar con árboles la estructura del programa entero.

Un pequeño paréntesis: Representación de árboles en Haskell

Para representar árboles con Haskell, necesitamos introducir un comando de Haskell que servirá para ello.

En Haskell se pueden fabricar nuevas clases de datos, con el constructor “data”.

Si buscamos en la librería de Haskell, vemos que la forma de definir al conjunto Bool es:

data Bool = False | True

Asimismo, podemos nosotros inventar alguna clase de datos que sea útil.

Por ejemplo, quisiéramos definir la clase de datos “Dedos”

data Dedos = Pulgar | Índice | Medio | Anular | Menique deriving Show

Primera aclaración: el nombre de la clase de datos siempre se escriben con la primera letra en mayúscula, como Bool, Integer, Natural. Los datos también.

Segundo: la frase **"deriving Show"** es para que se puedan mostrar.

Ahora que tenemos una nueva clase de datos, podemos hacer alguna función con ellos. Definamos una función que le asigne a cada número del 1 al 5 su dedo correspondiente.

Hay que escribir la función y la definición de la clase de datos en el **editor de texto.**

```
data Dedos = Pulgar | Índice | Medio | Anular | Menique deriving Show
nombre :: Integer -> Dedos
nombre 1 = Pulgar
nombre 2 = Indice
nombre 3 = Medio
nombre 4 = Anular
nombre 5 = Menique
```

Ahora, hay que guardar y probarlo.

Observación: la palabra Pulgar no lleva doble comillas, porque no es una cadena de caracteres. En este caso, Pulgar ya es el tipo de datos. Del mismo modo, cuando escribimos True no ponemos comillas.

Y ahora que ya sabemos fabricar una nueva clase de datos, podemos intentar hacer alguna clase de datos que sea más útil que Dedos.

En realidad, Dedos no fue inútil. ¡¡¡¡ Sirvió para aprender!!!!!

Árboles binarios

El caso particular de árboles dónde cada nodo debe tener exactamente dos hijos se llama árbol binario.

Como se dijo antes un nodo de un árbol puede tener cualquier cantidad de hijos. Los árboles binarios normalmente se usan en las aplicaciones prácticas de computación.

Definición inductiva de árboles binarios:

1. nil $\in$ (AB A)
2. (cons e izq der) $\in$ (AB A) sii e $\in$ A y izq $\in$ (AB A) y der $\in$ (AB A)
3. CC

Representación de árboles binarios en Haskell

1) ¿Recuerdan la forma de definir al conjunto Bool?
Los elementos de Bool son True o False. Son sólo 2 opciones.
Se escribe así:

```
data Bool = False | True deriving Show
```

2) ¿Recuerdan la forma de definir Dedos?
Dedos sólo tiene 5 opciones.

```
data Dedos = Pulgar | Índice | Medio | Anular | Menique deriving Show
```

3) Si quisiéramos definir una nuevo tipo de datos, llamado Color, que pudiera tener sólo 3 opciones, digamos blanco, negro o verde, se podría hacer así:

```
data Color = Blanco | Negro | Verde deriving Show
```

4) Estamos en condiciones de definir a los Árboles de la misma forma. Son sólo 2 posibilidades: o es un árbol vacío o es un nodo con otros árboles. Y como estamos definiendo ahora árboles binarios, tiene que haber en esta segunda opción exactamente 2 árboles. Queda así:

```
Data Arbol a = Vacio | Nodo a (Arbol a) (Arbol a) deriving Show
```

La variable “a” al principio, en data Arbol a, es para colocar allí el tipo de datos que contendrá el árbol. Por ejemplo, un árbol de caracteres, empezará así:

```
Data Arbol Char = …….
```

La variable “a” a la derecha de Nodo es para colocara el valor del elemento ubicado en el

Nodo. Por ejemplo, podrá ser Nodo 4, Nodo 17, etc, si se tratara de números enteros.

Ejemplo 1:

```
Data Arbol a = Vacio | Nodo a  (Arbol a) (Arbol a)   deriving Show
verde :: Arbol Integer
verde = Nodo 10 (Vacio) (Vacio)
```

Los 3 renglones anteriores hay que escribirlos en el editor de texto de Haskell. Luego, en la sesión de comandos, al digitar verde veremos la estructura del árbol verde. Esto es, aparecerá Nodo 10 (Vacio) (Vacio).

Ejemplo 2:

Para los demás ejemplos supondremos que estamos trabajando en el mismo archivo de Haskell, por lo que no se tiene que repetir la definición del tipo:

```
Data Arbol a = Vacio | Nodo a  (Arbol a) (Arbol a)   deriving Show
```

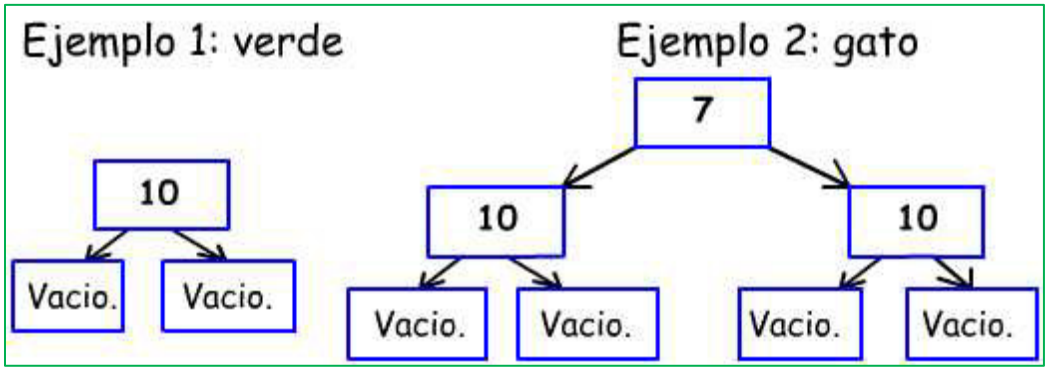
Lo que sí vamos a escribir, en el editor de texto es lo siguiente:

```
gato :: Arbol Integer
gato = Nodo 7 (verde) (verde)
```

¿Qué deberíamos esperar al digitar gato?

Vamos a pensar un poco: gato es un árbol que tiene como Nodo, el número 7. Luego tiene 2 subárboles, iguales (verde) que a su vez tiene, cada uno, un número 10 en su Nodo y luego, cada subárbol, Vacio.

Un dibujo, un esquema de cada uno, sería así.



Escribe en tu computadora gato y trata de razonar lo que responde.

Ejercicio 1:

Escribe uno o varios árboles de modo que al digitar “tres” la respuesta sea:

Nodo 3 (Nodo 1 Vacio Vacio) (Nodo 2 Vacio Vacio)

(Una posible solución, porque hay muchas formas de hacerlo, está al final del capítulo)

Ejemplo 3:

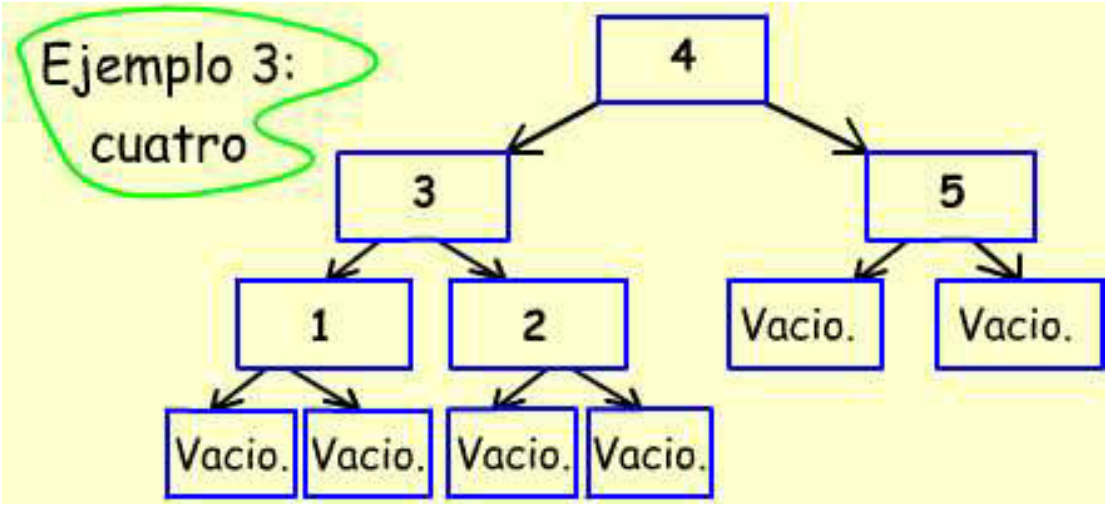
Escribamos lo siguiente en el editor de texto:

```
uno :: Arbol Integer
uno = Nodo 1 (Vacio) (Vacio)

dos :: Arbol Integer
dos = Nodo 2 (Vacio) (Vacio)
tres :: Arbol Integer
tres = Nodo 3 (uno) (dos)

cuatro :: Arbol Integer
cuatro = Nodo 4 (tres) (cinco)

cinco :: Arbol Integer
cinco = Nodo 5 (Vacio) (Vacio)
```

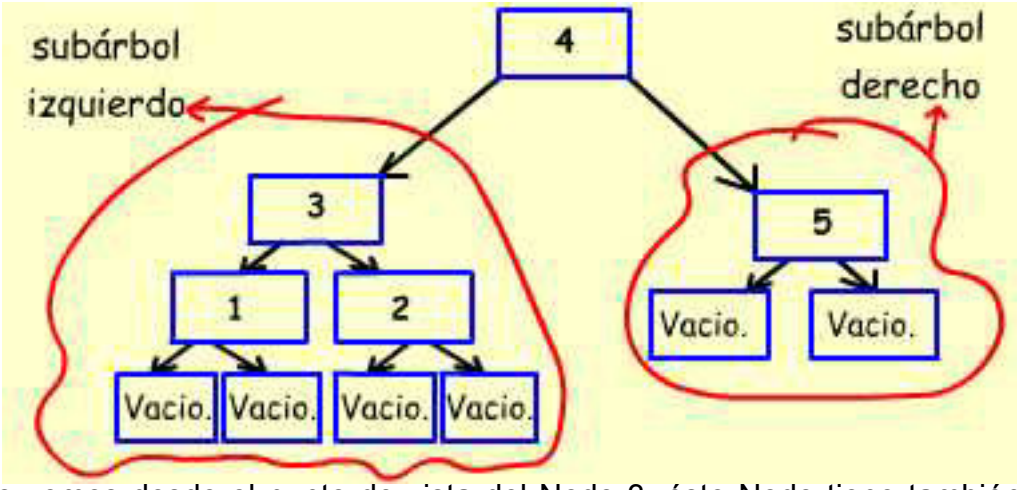
Digita ahora los 5 nombres de árboles, del uno al cinco.

La respuesta obtenida, ¿es lo que esperabas?

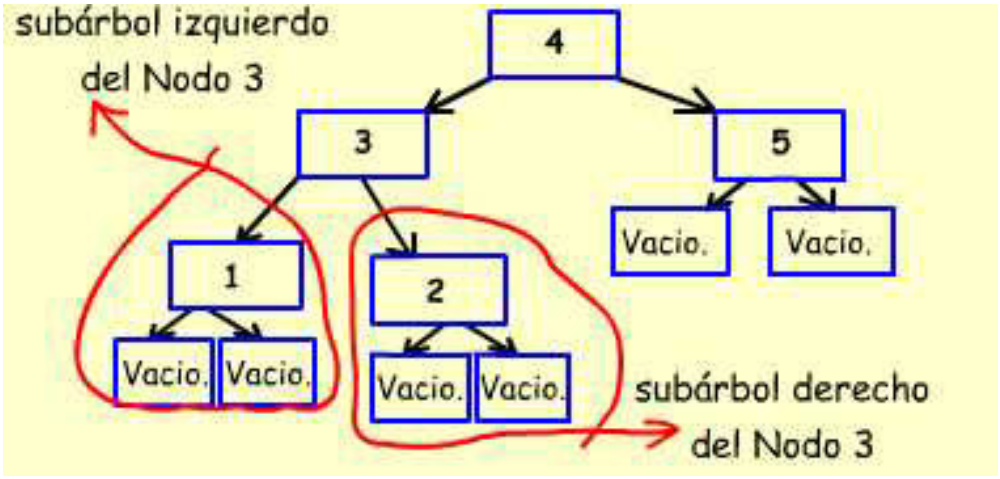
Algunas observaciones: “cuatro” es un árbol que tiene en su Nodo el número 4, a su izquierda el árbol “tres” y a su derecha el árbol “cinco”. Un esquema del mismo es el siguiente:



Si lo vemos desde el punto de vista del Nodo 4, tiene un árbol a la izquierda, el árbol tres, y otro árbol a su derecha, el árbol 5.



Pero si lo vemos desde el punto de vista del Nodo 3, éste Nodo tiene también un árbol a su izquierda y otro subárbol a su derecha.



A su vez, los Nodos 1, 2 y 5 también tienen subárboles izquierdo y derecho; son todos iguales y son el árbol Vacío.

Representación en Haskell de árboles.

Al digitar uno, estando el ejemplo anterior activo en la computadora, vemos:

```
Nodo 1 Vacio Vacio
```

Bueno, es sencillo de entender, pues tenemos el Nodo 1 y los subárboles Vacío.

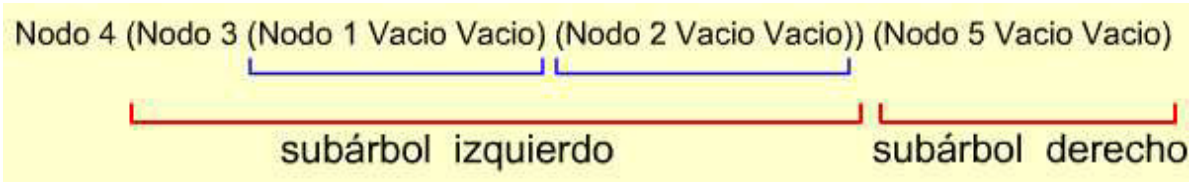
¿Qué pasará al digitar cuatro?

```
Nodo 4 (Nodo 3 (Nodo 1 Vacio Vacio) (Nodo 2 Vacio Vacio)) (Nodo 5 Vacio Vacio)
```

Si, ya no es tan sencillo. En realidad sólo hay que mirar bien los paréntesis.

Como es un árbol binario, el Nodo 4 tendrá 2 subárboles, uno a la izquierda y otro a la derecha.

Vamos a observarlo detenidamente:



En resumen, todos los árboles binarios siempre se representan del mismo modo.

```
Nodo a  (árbol izquierdo)  (árbol derecho)
```

Si dentro de la representación de cada subárbol hay más paréntesis, se va a dificultar "un poco" la visualización.

Vamos a escribir la representación del árbol cuatro de otra forma, más sencilla.

```
Nodo 4
        (Nodo 3
                (Nodo 1 Vacio Vacio)
                (Nodo 2 Vacio Vacio))
        (Nodo 5 Vacio Vacio)
```

De esta forma la visualización es mejor.

En el primer nivel, a la izquierda, el Nodo 4.

En el segundo nivel, un poco más a la derecha, los 2 subárboles del Nodo 4.

En el tercer nivel, todavía más a la derecha, los subárboles de cada subárbol del Nodo 4, y así seguiría el esquema.

Ejercicio 2:

Hacer el diagrama, la representación, del árbol definido de esta forma:

```
Nodo 4
        Nodo 2
                Nodo 1
                        Vacio
                        Vacio
                Nodo 3
                        Vacio
                        Vacio
        Nodo 7
                Nodo 5
                        Vacio
```

Nodo 6

Vacio

Vacio

Nodo 8

Vacio

Vacio

En una forma un poco más simplificada, pueden omitirse todos los “Vacio”.

Representación de árboles binarios en Haskell: otra mirada.

En la definición del tipo de datos Arbol, las palabras o comandos de Haskell predefinidos son data y deriving Show.

```
Data Arbol a = Vacio | Nodo a  (Arbol a) (Arbol a)   deriving Show
```

En idioma Inglés, en vez de Árbol decimos Tree , en vez de Nodo, Node y en lugar de Vacío, Leaf que significa hoja.

```
Data BinTree Int = Leaf |Node Integer BinTree Int BinTree Int  deriving Show
```

Dar los diagramas de los siguientes árboles implementados en Haskell:

```
tree1:: BinTree Int
tree1 = Leaf

tree2:: BinTree Int
tree2= Node 23 Leaf Leaf

tree3 :: BinTree Int
tree3=Node 4
        (Node 2
                (Node 1 Leaf Leaf)
                (Node 3 Leaf Leaf)) (Node 7
```

```
(Node 5
   Leaf
      (Node 6 Leaf Leaf)) (Node 8 Leaf
Leaf))
```

Haskell también permite definir árboles polimórficos, dónde los datos de los nodos son de algún tipo “a”. El árbol resultado tiene tipo BinTree a, que significa "árbol binario con valores de tipo a".
Árbol binario de un tipo a:

```
data BinTree a = BinLeaf
                 |BinNode a (BinTree a) (BinTree a) deriving Show

    tree4:: BinTree String
    tree4= BinNode "cat" BinLeaf (BinNode "dog" BinLeaf BinLeaf)

    tree5:: BinTree (Integer, Bool)
    tree5= BinNode (23, False) BinLeaf (BinNode (49,True) BinLeaf BinLeaf )

    tree6:: BinTree Int
    tree6 =BinNode 4 (BinNode2 (BinNode1 BinLeaf BinLeaf) (BinNode 3 BinLeaf
    BinLeaf)) (BinNode6 (BinNode5 BinLeaf BinLeaf) (BinNode7 BinLeaf BinLeaf))
```

En polaco:

Las definiciones no tienen que ser escritas en español o en inglés. Podemos usar, por ejemplo, el idioma polaco!!!!
En polaco, árbol se dice “drzewo”, hoja se dice “arkusz” y vacío se dice “pusty”.

```
Data Drzewo a = Pusty | Arkusz a  (Drzewo a) (Drzewo a)   deriving Show

uno :: Drzewo Integer
uno = Arkusz 53 (Pusty) (Pusty)
```

Este árbol **funciona** bien. Se puede probar copiarlo en el editor de texto y luego probar

digitando “uno”.

Funciones sobre árboles binarios.

Una función sobre árboles binarios es una función cuyo dominio es un árbol binario. Para hacer el primer ejemplo, trataremos de implementar una función sobre un árbol binario de enteros que aplicada sobre él, sume todos los elementos.

¿Cómo podremos hacer esto? Podemos basarnos en algún conocimiento previo.

Recordemos las listas. ¿Cómo se puede implementar la función sumalist, que se encarga de sumar todos los elementos de una lista? ¿Se acuerdan?

Si la respuesta es no, hay que leer de nuevo el capítulo de LISTAS.

La función sumalist es recursiva. La forma de hacerla es: sumamos al primer elemento de la lista la misma función pero aplicada en el resto de la lista.

```
sumalist:: [Integer] -> Integer
sumalist [ ] = 0
sumalist (a:xs) = a + sumalist (xs)
```

Una lista puede pensarse como una cabeza, llamada “a”, y una cola, llamada (xs).

Un árbol binario “podría pensarse” como una cabeza, también llamada “a”, y 2 colas, llamadas (izquierda) y (derecha). Para abreviar, podemos llamarlas (izq) y (der).

Esto es sólo una imagen mental para poder razonar los ejercicios. La definición formal de árbol binario ya lo vimos.

Entonces, si para listas le sumamos al primer elemento la misma función aplicada a la cola de la lista, en arboles podemos hacer algo similar. Esto es, al primer elemento, que aquí lo llamados Nodo, le sumamos la propia función aplicada a sus 2 “colas”.

Vamos a llamarle a esta función sumarbol.

```
sumarbol:: Arbol Integer ->Integer
sumarbol Vacio = 0
sumarbol (Nodo a (izq) (der)) = a + sumarbol (izq) + sumarbol (der)
```

Fijarse bien que es “exactamente” lo mismo que en listas, sólo que con 2 colas.

Probar dicha función con algún árbol conocido, por ejemplo, “cuatro”.

En resumen, en las funciones para ser utilizadas con árboles, es aconsejable aplicarle dicha

función al nodo y luego aplicársela al subárbol izquierdo y al subárbol derecho.

Recorrida de árboles

Una tarea común es recorrer uno a uno los nodos de un árbol con el fin de procesar los datos en de cada nodo, creando una lista como resultado. Un algoritmo que realiza esta función se denomina recorrida de un árbol.

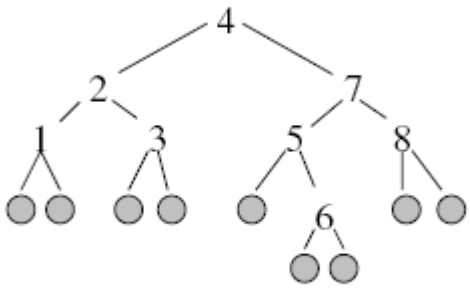
Para árboles binarios se utilizan comúnmente tres algoritmos para recorridas:

Preorden: se visita primero la raíz y a continuación, se recorre en preorden el subárbol izquierdo y luego en preorden el subárbol derecho.

Enorden: se visita el subárbol izquierdo en enorden, a continuación la raíz y por último el subárbol derecho en enorden.

Postorden: se visita el subárbol izquierdo en postorden, luego en postorden el subárbol derecho y por último la raíz.

Ejemplos:



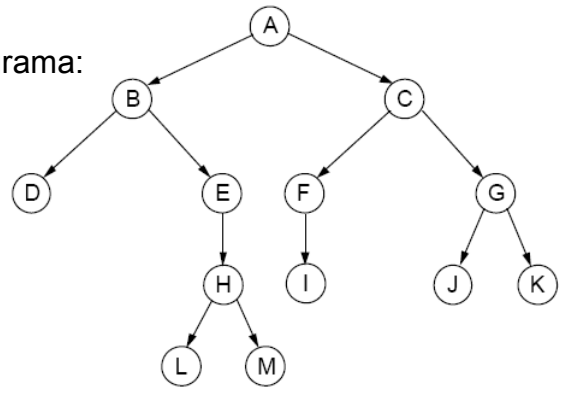
Recorrida en preorden del árbol de la figura: [4,2,1,3,7,5,6,8].

Recorrida en enorden del árbol de la figura: [1,2,3,4,5,6,7,8].

Recorrida en postorden del árbol de la figura: [1,3,2,6,5,8,7,4]

Ejercicios:

Para el árbol binario representado por el siguiente diagrama:



1. Listar los nodos del árbol anterior en:
   a. Preorden
   b. Enorden
   c. Postorden

2. Definir funciones que recorran un árbol binario en:
    a. Preorden
    b. Enorden
    c. Postorden

ÁRBOLES: PRÁCTICO

1. Definir un tipo de datos árbol que contiene un carácter y un entero en cada nodo, y exactamente tres subárboles.
2. Definir un tipo de datos árbol que contiene un entero en cada nodo, y que permite a cada nodo tener cualquier número de subárboles.
3. Defina el tipo de datos Tree que representa árboles binarios de elementos de un tipo genérico que sólo guarda información en los nodos hojas (nodos externos). Los nodos internos no guardan información. El árbol más pequeño es una hoja.
    a. Defina una función mapTree que dado un árbol de tipo (TreeA), para un tipo genérico A, y una función f: A -> B, con B un conjunto dado, retorne un árbol de tipo (TreeB) obtenido por la aplicación de la función f a cada uno de los nodos hojas del árbol parámetro.
    b. Defina una función que cuente la cantidad de nodos hojas que posee un árbol de tipo (TreeA), para un tipo genérico A.
    c. Pruebe que la función MapTree preserva la cantidad de nodos hojas del árbol parámetro.
    d. Defina una función hojas que retorne una lista con las hojas de un árbol de tipo (TreeA), para un tipo genérico A. Pruebe luego que la cantidad de nodos hojas que posee un árbol de tipo (TreeA), para un tipo genérico A, es igual a la longitud de la lista resultante de aplicarle la función hojas al árbol.

4. Defina el tipo de datos (BinTreeA) de árboles binarios con nodos internos de un tipo genérico A y nodos externos (hojas) de un tipo genérico B. El árbol más pequeño es una hoja.
    a. Defina una función que cuente la cantidad de nodos externos de un árbol binario de tipo (BinTreeA).
    b. Defina una función que cuente la cantidad de nodos internos de un árbol binario de tipo BinTree.
    c. Pruebe que la cantidad de nodos externos en un árbol binario de tipo BinTree es igual a la cantidad de nodos internos del árbol más 1.

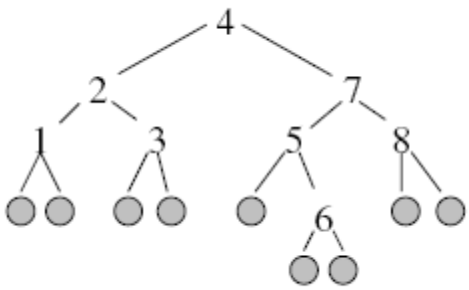
Respuestas a algunos ejercicios del capítulo: Árboles

Ejercicio 1:

```
dataArbol a = Vacio | Nodo a  (Arbol a) (Arbol a)   deriving Show

uno :: ArbolInteger

uno = Nodo 1 (Vacio) (Vacio)


dos :: ArbolInteger

dos = Nodo 2 (Vacio) (Vacio)


tres :: ArbolInteger

tres = Nodo 3 (uno) (dos)
```

Los nombres “uno”, “dos” y “tres” son fantasía y se pueden cambiar, por supuesto.

Ejercicio 2:



Principio de Inducción estructural

Trabajaremos sobre conjuntos que están definidos de forma inductiva. Se quiere probar que una determinada propiedad se cumple para todos los elementos de un conjunto que está definido en forma inductiva.

Sea A un conjunto inductivo cualquiera y P un predicado definido sobre ese conjunto A, o sea que P : A → Bool

Queremos probar el siguiente enunciado: $(\forall x \in A)\ (P\ x)$

1. Se debe probar que P se verifica para cada uno de los elementos que se infieren de las cláusulas base.
2. Para cada cláusula inductiva, se debe probar que si P se verifica para las premisas de la cláusula entonces P se verifica para la conclusión.
3. Habiendo probado lo indicado en 1) y 2), habremos probado $(\forall x \in A)\ (P(x))$

El principio que nos permite afirmar esto último se denomina Principio de Inducción Estructural, que es más conocido, en el caso en que A sea el conjunto de los números naturales, por el nombre Principio de Inducción Completa.

Enunciado del Principio de Inducción Estructural

Sea A un conjunto definido inductivamente y P un predicado definido sobre A, P: A → Bool.

Si para cada regla de la definición inductiva de A, de la forma $a1 \wedge a2 \wedge ... \wedge an \rightarrow a$ se verifica $(P\ a1) \wedge (P\ a2) \wedge ... \wedge (P\ an) \rightarrow P(a)$, entonces se verifica $(P\ x)$ para todo elemento x de A.

Inducción en la estructura de los números naturales:

Veamos cómo se aplica el principio de inducción estructural en el caso de los naturales.

Sea P un predicado unario que recibe un natural, P: N → Bool.

Para probar P(x) para todo natural:

1. Se prueba P(0)
2. Suponiendo P(n) para un cierto natural n, se prueba P(S(n)).

Con los pasos 1 y 2se ha probado P, para todo natural.

Probaremos el siguiente teorema trivial para ver la aplicación del principio: $(\forall n \in N)\ (n+0 = n)$

Prueba:

1. Paso base: Se debe probar que la propiedad se verifica para 0, es decir 0+0=0, trivial.
2. Paso inductivo: Hipótesis inductiva: x+0=x

Tesis inductiva: (S x)+0 = (S x)

Prueba del paso inductivo: (S x)+0 = (S(x+0)) = (S x)

Observaciones:

- ✓ La primera igualdad ( ) es cierta por cómo está definida la suma en el conjunto de los naturales: recordemos la definición: suma a Z=a

suma Z a=a

suma (S a) b = S (suma a b)

- ✓ La segunda igualdad ( ) es cierta por la hipótesis inductiva.

Inducción en la estructura de las listas:

Trabajaremos ahora sobre el conjunto de todas las listas sobre un cierto conjunto A.
El conjunto (tipo) definido inductivamente es list A, por lo tanto la prueba se realizará por inducción en la estructura de las listas.

Se quiere probar una cierta propiedad P: list A → Bool para todo elemento de list A. Se debe probar $(\forall x \in \text{list A})((P\ x))$

Probaremos ahora la siguiente propiedad:
$(\forall x \in \text{list A})(\forall a \in A)$ (largo (cons a x))>0

Prueba:

1. Paso base:

Se debe probar que la propiedad se verifica para nil, es decir (largo (cons a nil))>0, por definición de largo, (largo (cons a nil)) = 1 + (largo nil) = 1 + 0 > 0, entonces está probado.

2. Paso inductivo

Hipótesis inductiva: $(\forall a \in A)$(largo (cons a s)) > 0

Tesis inductiva: (∀a∈A)(∀b∈A)(largo (cons b (cons a s)) > 0

Prueba del paso inductivo:

Por definición de largo, (largo (cons b (cons a s))) = 1 + (largo (cons a s))>0 por HI

Probar las siguientes propiedades:

a. suma_S:(∀n, m∈N) ((suma n (S m)) = (suma (S n) m))
b. suma_conm:(∀n, m∈N) ((suma n m) = (suma m n))
c. P1:(∀ l ∈ list A) ((concat l nil) = l)
d. P2:(∀ l ∈ list A)(∀a∈A) (not((cons a l) = nil))
e. P3:(∀ l, q ∈ list A)(∀n∈A) ((cons n (concat l q))=(concat (cons n l) q))

Bibliografía:

Apostol, T. (1957). *Mathematical analysis* (1st ed.). Reading, Mass.: Addison-Wesley Pub. Co.

Grimaldi, R. (1994). *Discrete and combinatorial mathematics*. Reading, Mass.: Addison-Wesley.

Lewin, R. (2011). *Introducción al álgebra* (1st ed.). Santiago, Chile: Juan Carlos Sáez Editor.

Liu, C. (1977). *Elements of discrete mathematics*. New York: McGraw-Hill.

O'Donnell, J., Hall, C., & Page, R. (2006). *Discrete mathematics using a computer*. London: Springer.

Rojo, A. (1991). Ál*gebra I*. Buenos Aires, ed. El Ateneo.

Rosen, K. (2007). *Discrete mathematics and its applications*. Boston: McGraw-Hill Higher Education.

Ross, K. & Wright, C. (1992). *Discrete mathematics*. Englewood Cliffs, N.J.: Prentice Hall.

Ruiz Jiménez, B., Gutiérrez López, f., Gallardo Ruiz, J., & Guerrero García, P. (2004). *Razonando con Haskell* (1st ed.). Madrid: Thomson-Paraninfo.

Los autores.

Patricia Echenique es Profesora de Matemática egresada del Instituto de Profesores Artigas, es Profesora efectiva en Enseñanza Secundaria y es docente en Formación Docente desde hace varios años.
Cuenta además con estudios de posgrado especializados en la enseñanza de la Ciencia de la Computación: curso de Matematica Discreta utilizando el computador, CFE, y curso de Lógica con el computador: Cok, CFE..

Saúl Tenenbaum de Profesor de Matemática egresado del Instituto de Profesores Artigas.
Es Profesor efectivo en Enseñanza Secundaria y es docente en Formación Docente desde hace varios años.
Tiene varios cursos de posgrado: Matemática Discreta utilizando el computador, CFE, Especialización en Entornos Virtuales de Aprendizaje, Virtual Educa, Argentina, 3 cursos de Geogebra , ITE, España, ha ganado 3 concursos de Informática Educativa realizados por la Presidencia de la República y el MEC, además de haber participado como expositor en Congresos internacionales en el exterior. Es además Bachiller en Química egresado de la Universidad de la Republica. Su sitio web educativo www.x.edu.uy está avalado y recomendado por la Inspección de Matemática del CES y por el CFE.

Paula Echenique es Profesora de Matemática egresada del Instituto de Profesores Artigas, efectiva en Enseñanza Secundaria y es docente en Formación Docente desde hace varios años. Es Ingeniera en Computación egresada de la Universidad de la República. Ha dictado varios cursos de posgrado en el CFE, participado en Congresos internacionales y es una de las responsables de que hoy exista en el país un Profesorado de Informática.


9 789974 857704