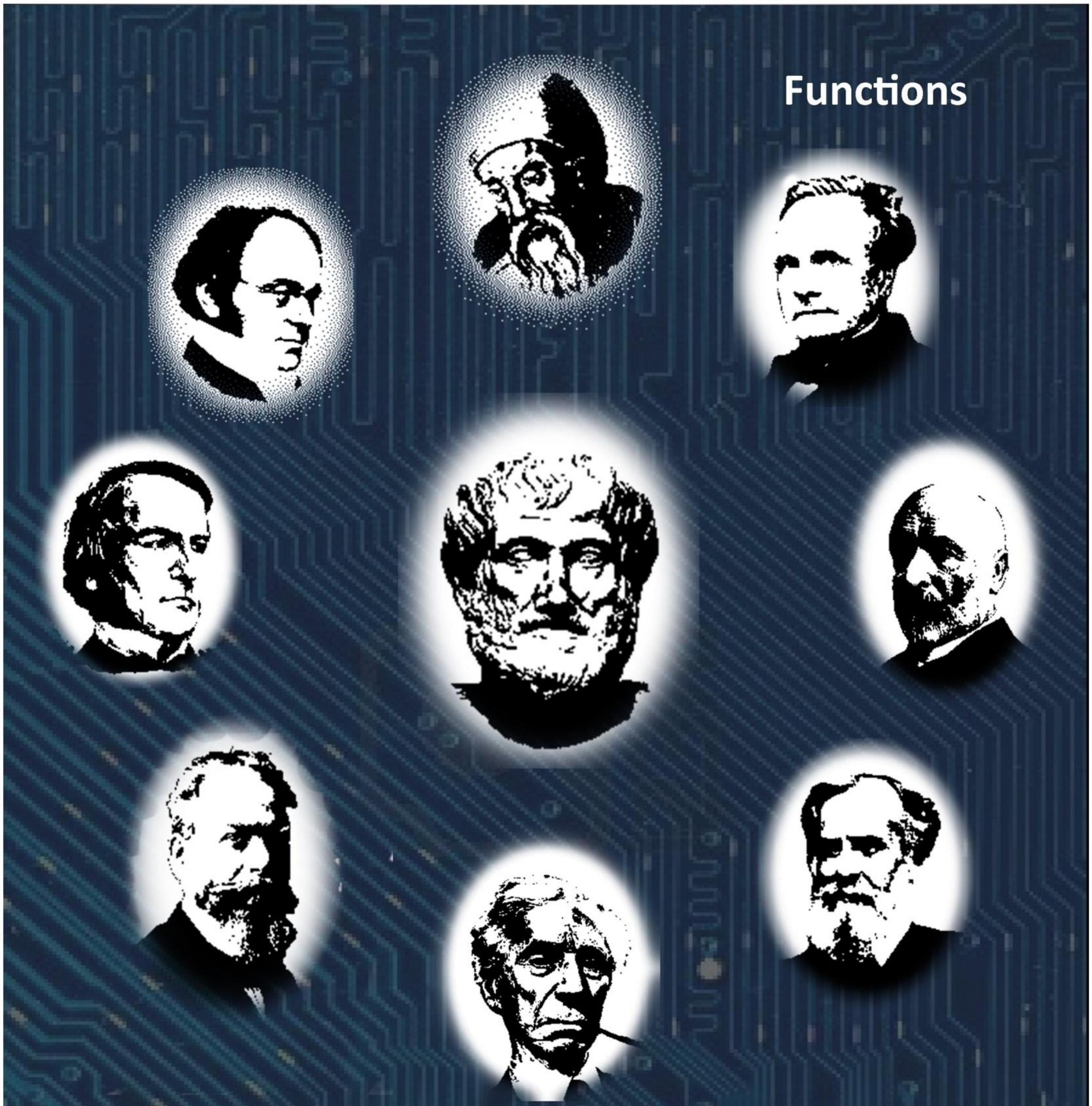
Functions



REASONING

elaine rich

alan kaylor cline

The Logicians on our cover are:

Euclid (? - ?)

Augustus De Morgan (1806 – 1871) Charles Babbage (1791 – 1871)

George Boole (1815 – 1864) Aristotle (384 BCE – 322 BCE) George Cantor (1845 – 1918)

Gottlob Frege (1848 – 1925) John Venn (1834 – 1923)

Bertand Russell (1872 – 1970)

REASONING

AN INTRODUCTION TO LOGIC, SETS, AND FUNCTIONS

CHAPTER 11

FUNCTIONS

Elaine Rich
Alan Kaylor Cline

The University of Texas at Austin

Image credits:

Pigeon holes: © 2014 Lynda Trader

Socks in drawer: © 2014 Lynda Trader

Extended pigeonhole principle: © 2014 Lynda Trader

Clock: http://www.crateandbarrel.com/decorating-and-accessories/clocks/1

http://www.cs.utexas.edu/learnlogic
Library of Congress Cataloging-in-Publication Data

Rich, Elaine, 1950 -
Reasoning—An Introduction to Logic Sets and Functions / Elaine Rich.— 1st ed. p. cm.
ISBN x-xxx-xxxxx-x 1

Table of Contents

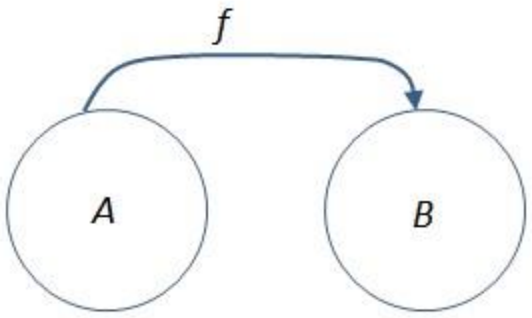
Introduction

What Is a Function?

A ***function*** is a relation that maps each element of its ***domain*** to *exactly one* element of its ***codomain***.

Let f be a function that maps from A to B. We can write a definition of f as:

$f : A \to B$: <what f does>



For example: $succ$: **N** → **N** : $succ(n) = n + 1$

If $f : A \to B$ and $x \in A$, then we can describe the operation of f on x by writing:

$f(x) = y$

Read this as, "f of x equals y," or "y equals f of x".

This table shows common terms for describing the roles of elements x of A and y of B when we evaluate $f(x)$:

	A (**domain**)	→	B (**codomain**)
f :	x	→	y
	preimage of y		***image*** of x
			value of $f(x)$
	argument of f		***result of applying*** f to x

We have that $succ(1) = 2$. So 2 is the image (or the value) of the argument 1 under $succ$.

Note that, given the definition of a function, every element of A has exactly one image (value) under f. But an element of B may have zero, one or more preimages. We'll soon see examples of all of these.

Define the ***range*** of f to be the subset of B (the codomain) that contains just those elements that are the image of *some* element x of A.

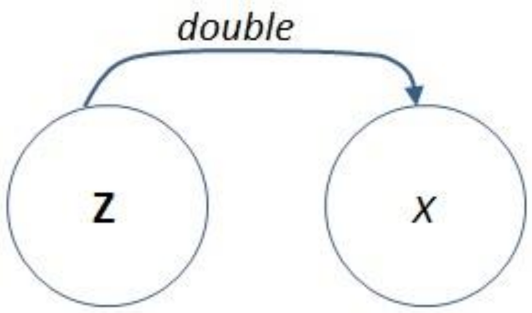
We will also use the notation $f(x)$ to refer to the function f (as opposed to the value of f at a specific point x) whenever we need a way to refer to f's argument. When we do this, we can use any name in place of "x", as long as we use it consistently throughout the definition.

For example, we'll write, as we did above, $succ(n) = n + 1$.

Problems

1. Define a simple doubling function defined from the integers to some as yet unspecified set X:

$$double : \mathbf{Z} \rightarrow X : double(n) = 2 \cdot n.$$



What is *double*(4)?

2. (Part 1) As we've defined it, the domain of *double* is which of the following:

a) The integers
b) The natural numbers
c) The reals
d) The positive integer
e) X

(Part 2) Now let's consider the codomain of *double*. We want *double* to be well-defined (i.e., it will be a *function* on **Z**). Mark each of the following statements as true or false:

double will be well-defined if $X = \mathbf{Z}$.

(Part 3) *double* will be well-defined X = {even integers}.

(Part 4) *double* will be well-defined if $X = \mathbf{N}$ (natural numbers).

(Part 5) Now consider the range of *double*. Recall that the range of any function f is the set that contains exactly those elements that are the image of some element of f's domain. So f's range is a (possibly proper but not necessarily so) subset of its codomain. What is the range of f?

3. Recall that we said that, given some function *f* from domain *A* to codomain *B*, every element of *A* has exactly one image (value) under *f*. But an element of *B* may have zero, one, or more preimages. Define *f* on the integers:

Let $f : \mathbf{Z} \rightarrow \mathbf{Z} : f(n) = n^2$.

(Part 1) Which of the following values has exactly one preimage under *f* (in other words, there is exactly one element of **Z** that maps to it):

a) 5
b) 0
c) 16

(Part 2) Which of the following values has exactly two preimages under *f*:

a) 5
b) 0
c) 16

(Part 3) Which of the following values has no preimages under *f*:

a) 5
b) 0
c) 16

(Part 4) How many preimages under *f* does -4 have:

a) 0
b) 1
c) 2

(Part 5) All of the following values are in the codomain of *f* as we have defined it. For each, indicate whether or not it is in the *range* of *f*:

a) -4
b) 0
c) 9
d) 10

4. Recall that we have been exploiting a variety of logical predicates, including:

Student(x) *KilledBy*(x, y) *Supervises*(x, y) *Scheduled*(c, r, t)

We can now observe that logical predicates are functions (that always return Boolean values).

(Part 1) Let P be an arbitrary predicate of one argument (e.g., *Student*(x)) that is drawn from some set S (e.g., *People*). Which of the following defines P as a function:

a) $P : S \to \{True, False\}$
b) $P : \{True, False\} \to S$
c) $P : \{True, False\} \to \{False, True\}$
d) $P : S \to S$
e) $P : S \to S \times S$

(Part 2) Let P be an arbitrary predicate of two arguments (e.g., *Supervises*(x, y)), the first of which is drawn from some set S and the second of which is drawn from set W. Which of the following defines P as a function:

a) $P : S \to \{True, False\}$
b) $P : \{True, False\} \to S \times W$
c) $P : S \times W \to \{False, True\}$
d) $P : S \times W \to \{False, True\} \times \{True, False\}$
e) $P : W \times S \to \{False, True\}$

Not All Relations Are Functions

Many relations, while useful, are not functions.

Recall the relation:

$Speaks \subseteq People \times Languages = \{(a, b) : a \text{ speaks } b\}$

Speaks is not a function since some people speak multiple languages.

But many useful relations are functions.

Consider one of the relations from a typical corporate Human Relations database:

$SS\# \subseteq EmployeeIDs \times NineDigitNumbers = \{(e, n) : n \text{ is } e\text{'s social security number}\}$

SS# is a function since every employee has exactly one social security number. We can define it, using our function-definition notation as:

$SS\# : EmployeeIDs \rightarrow NineDigitNumbers : SS\#(e) = e\text{'s social security number}$

Often, in a single problem domain, there are relations that are functions and others that are not.

Let *Cities* be the set of cities in the world. (We'll skip over the details of how large something has to be to qualify as a "city". Also, we'll assume, for simplicity, and counter to fact, that a name uniquely identifies a city.) Let *Countries* be the set of countries in the world. Define two relations between *Countries* and *Cities*:

$Capitals \subseteq Countries \times Cities = \{(a, b) : b \text{ is the capital of } a\}$
$Places \subseteq Countries \times Cities = \{(a, b) : b \text{ is in } a\}$

Barring very unusual circumstances, we can say that *Capitals* is a function since every country has exactly one capital. *Places* is not a function since most countries have many cities.

Functions are fundamental in mathematics.

Some important mathematical functions include:

- Polynomial functions on the reals, such as: $f(x) = x^3 + 7x - 12$
- Trigonometric functions on the reals, such as: $f(x) = \sin(x)$
- Useful functions on the natural numbers, such as: $f(x) = n!$
  $f(x) = \sum_{i=1}^{x} i$

English Aside

Assigning a referent (called an antecedent) to a pronoun ought to be a function in clear, situated (i.e., said in some particular context in the world) English. But often it isn't and confusion can result. (Oversimplifying a complex phenomenon) define the relation:

$$AssignReferent \subseteq Pronouns \times Objects = \{(p, o) : \text{pronoun } p \text{ could refer to object } o\}$$

Now consider these examples:

"Jon gave Mary his new book."

("his", Jon") $\in$ *AssignReferent*, which is behaving like a function. No other objects are related to "his". The pronoun "his" refers to Jon.

"Megan gave Mary her new book."

Now ("her", Megan) and ("her", Mary) are both in *AssignReferent*, which is now obviously not a function. We don't know whether the new book belongs to Megan or Mary.

Problems

1. Mark each of the following genealogy relations as a function or not a function:

a) $HasAsBiologicalfather \subseteq People \times People$ = $\{(a, b) : a$ has b as a biological father$\}$
b) $HasAsBiologicalancestor \subseteq People \times People$ = $\{(a, b) : a$ has b as a biological ancestor$\}$
c) $HasAsBiologicalsister \subseteq People \times People$ = $\{(a, b) : a$ has b as a biological sister$\}$

2. Mark each of the following relations as a function or not a function. (Hint: Be careful. Check the boundary cases. A couple of these are subtle.) Recall that **N** is the set of natural numbers (starting with 0). **Z** is the set of integers. $\mathbf{Z}^+$ is the set of positive integers.

a) $Previous \subseteq \mathbf{Z} \times \mathbf{Z} = \{(j, k) : k = j - 1\}$
b) $Previous \subseteq \mathbf{N} \times \mathbf{N} = \{(j, k) : k = j - 1\}$
c) $GreatestCommonDivisor \subseteq (\mathbf{Z}^+ \times \mathbf{Z}^+) \times \mathbf{Z}^+ = \{((n, m), k)$: k is the greatest common divisor of n and $m\}$
d) $CommonDivisor \subseteq (\mathbf{Z}^+ \times \mathbf{Z}^+) \times \mathbf{Z}^+ = \{((n, m), k)$: k is a common divisor of n and $m\}$

3. Consider the following two paragraphs:

[1] Chris was invited to Tracy's birthday party. She$_1$ wondered whether she$_2$ would like a new kite.

[2] Chris was invited to Tracy's birthday party. She$_1$ wondered whether she$_2$ could afford a new kite.

In answering the following questions, you can consider either paragraph [1] or paragraph [2]. You'll get the same answer either way. We've written both just to help you see how the *assignReferent* process could work.

(Part 1) Without using any knowledge about gift-giving conventions at birthday parties, how many ordered pairs with the first element "she$_1$" are in *AssignReferent*? (In other words, to how many different values could "she$_1$" refer?)

(Part 2) Without using any knowledge about gift-giving conventions at birthday parties, how many ordered pairs with the first element "she$_2$" are in *AssignReferent*? (In other words, to how many different values could "she$_2$" refer?)

Unary Functions, Binary Functions

Every function maps from individual elements of its domain to individual elements of its codomain. But when we think about the work that we want functions to perform, it may be useful to think of them as applying to individual objects, to pairs of objects, to triples of objects, or to even larger structures. When we want more than one input in this sense, we can define a function's domain to be the Cartesian product of two or more sets.

Some functions take a single argument. We call such functions ***unary functions***. In this case, we don't need Cartesian products.

Two simple examples of unary functions:

- Defined on the natural numbers: $succ: \mathbf{N} \rightarrow \mathbf{N}$: $succ(n) = n + 1$
- Defined on (B) Boolean expressions: $not: B \rightarrow B$: $not(x) = \neg B$

Some functions take two arguments. We call such functions ***binary functions***. The domain of a binary function must be a set of ordered pairs. However we almost always write $f(x, y)$ instead of the more explicit $f(\,(x, y)\,)$.

Two simple examples of binary functions:

- Defined on pairs of integers: *plus* : $\mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$
- Defined on pairs of people: *#ofcommonfacebookfriends* : $People \times People \rightarrow \mathbf{N}$

The function notation that we've defined so far is called ***prefix notation***. The name of the function comes first, followed by the arguments. To call or invoke a function defined in this way, we write the name of the function followed by a parenthesized list of its arguments.

So we could write *succ*(97) or *#ofcommonfacebookfriends*(Casey, Frankie).

But, in the case of binary functions, it may be easier to use an alternative, ***infix notation***, in which a function symbol, called an *infix operator*, is placed between the two arguments.

So, instead of writing: *plus*(x, *y*)

We may write: x + *y*

Often we use common infix operators to define more special-purpose prefix ones.

Define *howmucholder* : $(Ages \times Ages) \rightarrow Ages$: $howmucholder(a_1, a_2) = a_1 - a_2$

By the way, there is nothing magic about one and two. It is possible (although not as common) to define functions with three or more arguments.

Problems

1. When we introduced the Boolean operators, we called them "operators", rather than "functions", because we hadn't yet given a formal definition of a function. But we now see that the Boolean operators are, in fact, functions. All but one of the ones we've discussed are binary functions. One is unary. Which one?

a) $\vee$
b) $\wedge$
c) $\rightarrow$
d) $\neg$
e) $\equiv$

2. Let's make sure that we're clear about what we have to write to define a function. A person's Body Mass Index (BMI) is a function of his or her height and weight. BMI is a positive rational number (i.e., it can be a fraction). It is generally in the range 12 – 50. It is a measure of whether one has a healthy weight, is underweight, or is overweight. We'll assume that we're working with height in inches and weight in pounds (both whole numbers). Then the BMI formula is:

$$(\text{weight} / \text{height}^2) \cdot 703$$

Let *values* be the set of rational numbers between 12 and 50. Consider each of the following attempts to use our notation to define the function *bmi*:

I.	*bmi*	:	$(\mathbf{Z}^+ \times \mathbf{Z}^+) \rightarrow \textit{values}$	:	$\textit{bmi}(i, j) = (i / j^2) \cdot 703$
II.	*bmi*	:	$(\mathbf{Z}^+ \times \mathbf{Z}^+) \rightarrow \textit{values}$	:	$\textit{bmi}(\textit{height}, \textit{weight}) = (\textit{weight} / \textit{height}^2) \cdot 703$
III.	*bmi*	:	$(\mathbf{Z}^+ \times \mathbf{Z}^+) \rightarrow \textit{values}$	:	$\textit{bmi} = (\textit{weight} / \textit{height}^2) \cdot 703$
IV.	*bmi*	:	$\textit{height} \times \textit{weight} \rightarrow \textit{values}$	:	$\textit{bmi} = (\textit{weight} / \textit{height}^2) \cdot 703$

Which one (or more) of them is/are correct?

The Function Distinction Matters for Programmers

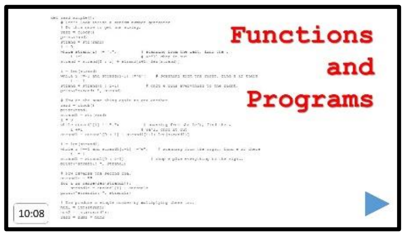
Suppose that I want to write code to compute the average of a set of numbers. I might write (in Python):

```
def average(numbers):    #numbers will be a list of numbers.

    sum = 0                # This loop will add the numbers one at
    for n in numbers:      #        a time into sum.
        sum = sum + n

    if len(numbers) == 0: # --- check how many there are.
        return (0)
    else:
        return(sum/len(numbers))  # Divide unless none.
```

In this code, I've counted on the fact that **+** is a function on the integers. It is guaranteed to return exactly one value, which I can continue to compute with.

On the other hand, **/** (the division operator) is not a function on the integers. It returns no value for *j*/*k* if *k* = 0. So, in my code, I had to treat that as a special case. That's why I tested (in the line marked ---) to see whether I'd been given a list of zero numbers to average.

Whenever I call someone's code, I need to know whether that code implements a function on the domain I'm dealing with. If it doesn't, then I need to worry about what happens if it doesn't return a single value.



https://www.youtube.com/watch?v=6_LsROvbiuE

A program *P* might compute a relation *R* that isn't a function for one or both of the following reasons:

- There are elements of the domain with which *R* associates *no* value (the divide by 0 case).
- There are elements of the domain with which R associates *multiple* values.

We can deal with both of these cases by turning *R* into a function that returns a unique *set* of individual values. The set may be empty. It may contain a unique element. Or it may contain multiple elements.

There are many common cases where we want to describe a function that returns a set of values.

employeesof	: *Divisions* $\rightarrow$ *Sets of Employees*
suppliersfor	: *Products* $\rightarrow$ *Sets of Suppliers*

Alternatively, if the only issue is that *R* returns no value, we can just test for that as a special case (as we did in the average program above).

Problems

1. Assume that we are running a summer camp for kids. We have a database into which we enter information about each child, based on the (possibly badly filled out) forms that the parents have submitted, plus the comments of our counsellors.

(Part 1) Let *Grades* = {K, 1, 2, … , 12}. Consider the relation:

SchoolGrade $\subseteq$ *Kids* $\times$ *Grades*

We want to implement the function *getgrade* : *Kids* $\rightarrow$ ________ that returns a given child's upcoming grade in traditional school. Which of the following statements is true:

a) We can almost certainly get away with implementing *getgrade* as a function *getgrade* : *Kids* $\rightarrow$ *Grades* that returns a unique grade level for each child.
b) We can almost certainly implement *getgrade* as a function *getgrade* : *Kids* $\rightarrow$ *Grades* that returns a unique traditional grade level for each child, but only if we check for the special case of someone who doesn't have a traditional grade assignment. Correct
c) Even if we check for the case of a student with no grade assignment, we've very likely to get in trouble unless we implement *getgrade* as a function *getgrade*: *Kids* $\rightarrow$ *Set of Grades* that returns a set of grade assignments.

(Part 2) It's common for families to send all their children to our camp. Consider the relation:

Siblings $\subseteq$ *Kids* $\times$ *Kids* = {(x, y) : x and y are brothers or sisters}

We want to implement the function *getsibs* : *Kids* $\rightarrow$ ________ that returns a given child's sibling(s). Which of the following statements is true:

a) We can implement *getsibs* as a function *getsibs* : *Kids* $\rightarrow$ *Kids* that returns a unique sibling for each child.
b) We can implement *getsibs* as a function *getsibs* : *Kids* $\rightarrow$ *Kids* that returns a unique sibling for each child but only if we check for the special case of someone who has no siblings.
c) We must implement *getsibs* as a function *getsibs* : *Kids* $\rightarrow$ *Set of Kids* that returns a (possibly empty) set of siblings.

Properties of Functions

Introduction

Recall that a function is a relation that delivers exactly one value for every element of its domain.

Because a function is a relation, we might want to consider whether it possesses one or more of the properties that we've found it useful to associate with relations. In particular, we might ask, of some function *f*, whether it is:

- reflexive,
- symmetric, and/or
- transitive.

These properties are only defined for relations on a *single set* (i.e., not for relations that map from one set to another). And we'll clearly want some functions that do map from one set to a different one. But there are also many useful functions on a single set (for example the reals or the integers). So, in the next few problems, we'll look at whether these three properties will be useful to us as we study functions.

Then we'll define two new properties that some functions possess:

- A function may be one-to-one.
- A function may be onto.

Problems

1. Let's explore what it would mean if a function were reflexive (in other words, under *f*, every element is related to itself). Let *f* be an arbitrary function on some set *A*.

Recall that, since *f* is a function, for every *a* in *A*, there is exactly one $(a, b) \in f$. Suppose that *f* is reflexive. Consider the following claims:

I. *f* must be the empty relation.
II. *f* must equal the identity relation *I*.
III. *f* must equal $A \times A$.

Which of them is/are true?

2. Let's explore what it would mean if a function were transitive. Let *f* be an arbitrary function on some set *A*.

Assume that *f* is transitive and contains some element (*a*, *b*). Consider the following claims:

I. It must be the case that $a = b$.
II. There is no *c* such that *f* contains (*b*, *c*).
III. There is at most one *c* such that *f* contains (*b*, *c*).

Which of them is/are true?

One-to-One and Many-to-One Functions

We've just seen that the properties that were useful in talking about arbitrary relations are much less useful in talking about functions. There are, however, two new properties that are very useful as we analyze functions.

Some functions preserve distinctness.

A function *f* from *A* to *B* is ***one-to-one*** if no two distinct elements of *A* map to the same element of *B*. Put another way:

$f : A \rightarrow B$ is one-to-one if and only if, whenever $a \neq b$, $f(a) \neq f(b)$.

If *f* is one-to-one, we may write:

$$f: A \xrightarrow{1\text{-}1} B$$

A function *f* from *A* to *B* is ***many-to-one*** if, on the other hand, there exist any two distinct elements of *A* that map to the same element of *B*.

An example of a one-to-one function:

Let $succ(n) : \mathbf{N} \rightarrow \mathbf{N}$ be the successor function for the natural numbers. It is one-to-one. Any natural number *k* can be the successor of at most one number. (We say "at most" rather than "exactly" because 0 isn't the successor of anything.)

An example of a many-to-one function:

$$parity : \mathbf{N} \rightarrow \{0, 1\} : \quad parity(n) \; = 0 \text{ if } n \text{ is even} \; = 1 \text{ if } n \text{ is odd}$$

The function *parity* is many-to-one because, for example, $parity(2) = parity(258) = 0$.

Problems

1. Define the function: $\mathit{parity} : \mathbf{N} \rightarrow \{0, 1\}$: $\mathit{parity}(n)$ = 0 if n is even
= 1 if n is odd

Now define the relation: $\mathit{equalparity} \subseteq \mathbf{N} \times \mathbf{N} = \{(j, k) : \mathit{parity}(j) = \mathit{parity}(k)\}$.

True or false:

a) *equalparity* is a function.
b) *equalparity* is an equivalence relation.
c) *parity* is an equivalence relation.

2. Mark each of the following functions true if it is 1-1 and false otherwise:

fingerprintpattern : *People* → *Fingerprints*
numsiblings : *People* → **N**
managerof : *Employees* → *Employees* : *managerof*(*e*) = *e*'s direct supervisor
birthday : *People* → {1 – 365} : *birthday*(*p*) = *p*'s birthday given as a day of the year

Onto Functions

Some functions map the domain to the entire codomain.

A function f from A to B is ***onto*** if every element of B is the value of the function applied to some element of A. Put another way:

$f : A \rightarrow B$ is onto if and only if, for all $b \in B$, there exists some $a \in A$ such that $f(a) = b$.

Another way to say this is that f is onto if and only if its codomain equals its range.

In this case, we'll say that f maps A *onto* B and we may write:

$$f : A \xrightarrow[\text{onto}]{} B$$

An example of an onto function:

Let $succ(n) : \mathbf{Z} \rightarrow \mathbf{Z}$ be the successor function for the integers. Every integer is the successor of some other integer.

An example of a function that isn't onto:

Let $succ(n) : \mathbf{N} \rightarrow \mathbf{N}$ be the successor function for the natural numbers. 0 isn't the successor of any natural number.

By the way, it may be possible to transform a non-onto function f into an onto function f' simply by cutting down the codomain of f' so that it includes only those elements that can be returned by f (i.e., f's range).

For example, here's an onto version of successor for the natural numbers:

$succ(n) : \mathbf{N} \rightarrow \mathbf{Z}^+$

We just remove 0 from the codomain.

Sometimes it's easy to do this and we may want to, depending on what problem we're trying to solve. Sometimes there is no straightforward way to do it.

Recall the function:

SS# : *EmployeeIDs* → *NineDigitNumbers* : *SS#*(*e*) = *e*'s social security number

To make *SS#* onto, we'd have to remove, from the set of nine digit numbers, all the ones that don't happen to correspond to someone's social security number. We can do that. Formally, it's easy. We have a finite set and we can just list the elements. But any benefit we might get from this is outweighed by the trouble.

If a function $f : A \rightarrow B$ is both one-to-one and onto, we will say that it is a ***bijection***. And we may write:

$$f: A \xrightarrow[\text{onto}]{\text{1-1}} B$$

Problems

1. Define *double* : $\mathbf{N} \to \mathbf{N}$: $double(n) = 2n$

Which of the following statements is true:

a) The *double* function that we've just described is onto.
b) The *double* function that we've just described is not onto but there is a simple description of a different codomain such that it would become onto.
c) The *double* function that we've just described is not onto and there's no simple way to shave its codomain that makes it so.

2. Define *EIDs* to be the set of legal electronic identification codes issued by our school.

Define *lookupeid* : *People* $\to$ *EIDs* : *lookupeid*(*p*) = *p*'s assigned eid

Which of the following statements is true:

a) The *lookupeid* function that we've just described is onto.
b) The *lookupeid* function that we've just described is not onto but there is a simple description of a different codomain such that it would become onto.
c) The *lookupeid* function that we've just described is not onto and there's no simple way to shave its codomain that makes it so.

3. Indicate whether each of the following functions is onto or not onto:

fingerprintpattern	:	*People* $\to$ *Fingerprints*
numsiblings	:	*People* $\to$ $\mathbf{N}$
managerof	:	*Employees* $\to$ *Employees* : *managerof*(*e*) = *e*'s direct supervisor
birthday	:	*People* $\to$ {1 – 365} : *birthday*(*p*) = *p*'s birthday given as a day of the year

One-to-One and Onto Functions

Relations may or may not be functions. Functions may be one-to-one, they may be onto, they may be both, or they may be neither.

Consider the following six examples. Imagine that each describes a relation whose domain is the first column and whose codomain is the second column:

1

$$\begin{tikzcd} A \arrow[dr] & X \\ B \arrow[dr] & Y \\ C & Z \end{tikzcd}$$

2

$$\begin{tikzcd} A \arrow[r] \arrow[dr] \arrow[ddr] & X \\ B \arrow[r] & Y \\ C \arrow[r] & Z \end{tikzcd}$$

3

$$\begin{tikzcd} A \arrow[r] & X \\ B \arrow[ur] & Y \\ C \arrow[ur] & Z \end{tikzcd}$$

4

$$\begin{tikzcd} A \arrow[r] & X \\ B \arrow[r] & Y \\ C \arrow[dr] & Z \\ & Q \end{tikzcd}$$

5

$$\begin{tikzcd} A \arrow[dr] & X \\ B \arrow[ur] & Y \\ C \arrow[r] & Z \\ D \arrow[ur] & \end{tikzcd}$$

6

$$\begin{tikzcd} A \arrow[dr] & X \\ B \arrow[ur] & Y \\ C \arrow[r] & Z \end{tikzcd}$$

What can we say about each of these relations? Is it a function or not? If it is a function, what properties does it possess?

1. Not a function. *C* is an element of the domain that is not related to any element of the codomain.
2. Not a function because there are three values associated with *A*.
3. Function. For each object in the first column, there is a single value in the second column. But this function is neither one-to-one (because *X* is derived from both *A* and *B*) nor onto (because *Z* is not the image of anything).
4. Function that is one-to-one (because no element of the second column is related to more than one element of the first column). But it still isn't onto because *Z* has been skipped: nothing in the first column derives it.
5. Function that is onto (since every element of column two has an arrow coming into it), but it isn't one-to-one, since *Z* is derived from both *C* and *D*.
6. Function that is both one-to-one and onto.

Problems

1. Let’s look again at:

1
$$\begin{tikzcd}
A \arrow[dr] & X \\
B \arrow[dr] & Y \\
C & Z
\end{tikzcd}$$

2
$$\begin{tikzcd}
A \arrow[r] \arrow[dr] \arrow[ddr] & X \\
B \arrow[r] & Y \\
C \arrow[r] & Z
\end{tikzcd}$$

3
$$\begin{tikzcd}
A \arrow[r] & X \\
B \arrow[ur] & Y \\
C \arrow[ur] & Z
\end{tikzcd}$$

4
$$\begin{tikzcd}
A \arrow[r] & X \\
B \arrow[r] & Y \\
C \arrow[dr] & Z \\
 & Q
\end{tikzcd}$$

5
$$\begin{tikzcd}
A \arrow[dr] & X \\
B \arrow[ur] & Y \\
C \arrow[r] & Z \\
D \arrow[ur] &
\end{tikzcd}$$

6
$$\begin{tikzcd}
A \arrow[dr] & X \\
B \arrow[ur] & Y \\
C \arrow[r] & Z
\end{tikzcd}$$

(Part 1) Consider relation 4. We said that it is a function that is one-to-one but not onto. Suppose that we want to make it onto (while keeping it a one-to-one function). Which of the following claims is correct:

a) We can do so by adding one arrow.
b) We can do so by subtracting one arrow.
c) We can do so by changing one of the existing arrows.
d) We can’t do so without changing at least one of the columns.

(Part 2) Now consider relation 5. We said that it is a function that is onto but not one-to-one. Suppose that we want to make it one-to-one (while keeping it an onto function). Which of the following claims is correct:

a) We can do so by adding one arrow.
b) We can do so by subtracting one arrow.
c) We can do so by changing one of the existing arrows.
d) We can’t do so without changing at least one of the columns.

When Are These Properties Important?

For some applications, a relation that isn't a function does exactly what we need. In other cases, we must have a function. When we need a function, it may or may not be important that the function we define be one-to-one, onto, or both.

Problems

1. We are designing a program that will encrypt the passwords of our users so that, even if we get hacked, passwords cannot be stolen. The idea is that we won't store "plaintext" passwords. We'll just store the encrypted versions. But we must design the encryption program so that, when a user types in a password, we can encrypt it and then compare the result to the stored (encrypted) password to determine whether the entered password is correct.

Let *Passwords* be the set of strings that satisfy our site's rules for choosing legal passwords.

We want to design *encrypt* : *Passwords* → *Strings*

Mark each of the following statements true or false:

a) *encrypt* must be a function.
b) It is okay if *encrypt* is not a function. It does not have to return a unique value for every password.
c) *encrypt* must not only be a function, it must be 1-1.
d) *encrypt* must not only be a function, it must be onto.
e) *encrypt* must be a function and it must not be 1-1.

2. Imagine that we are a video streaming service. We want to recommend videos to our customers based on their preferences. We want to design:

recommend : *Users* → *Movies*

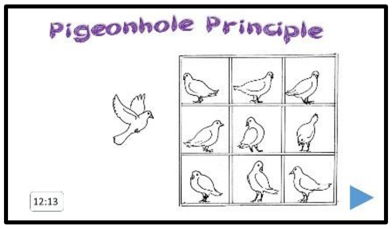
Mark each of the following statements true or false:

a) *recommend* must be a function.
b) It is okay if *recommend* is not a function. It does not have to return a unique value for every user.
c) *recommend* must not only be a function, it must be 1-1.
d) *recommend* must not only be a function, it must be onto.
e) *recommend* must be a function and it must not be 1-1.

Pigeonhole Principle

Counting Arguments and the Pigeonhole Principle

The ***pigeonhole principle***:



$$\left.\begin{array}{c}\text{more pigeons}\\ \textit{and}\\ \text{fewer holes}\end{array}\right\} \longrightarrow \text{pigeon roommates}$$

https://www.youtube.com/watch?v=IeTLZPNIPJQ

Or it could be stated in terms of college freshmen and dorm rooms.

Somewhat more formally:

> If $k > n$, you can't stuff k pigeons into n holes without having at least two pigeons in the same hole.

We can also state the principle more generally and even more formally. To do so, we'll use the notion of a one-to-one function:

> A function f from A to B is ***one-to-one*** if no two distinct elements of A map to the same element of B.

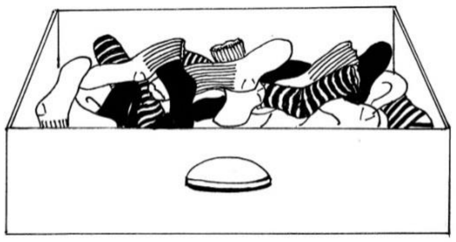
Now we can restate the pigeonhole principle:

> For any sets A and B:
>
> - $|A| > |B| \quad\rightarrow\quad$ there exists no one-to-one function from A to B.
> - there exists a one-to-one function from A to $B \quad\rightarrow\quad |A| \leq |B|$.

Notice that the second line is just the contrapositive of the first. We'll most often use it in the first direction, in which we know the sizes of the two sets and we want to conclude that no one-to-one mapping exists from one to the other. But sometimes the other direction is also useful.

Here's a straightforward proof by contradiction of the principle: Suppose that $|A| > |B|$ and that there *were* a one-to-one function f from A to B. Then no element of B is the image under f of more than one element of A (no hole contains more than one pigeon). Then $|A|$ could be no greater than $|B|$ (there could be no more than $|B|$ pigeons). But it is.

Suppose that we have a drawer full of white, black, and striped socks. How many socks must we pull out of the drawer in order to be *guaranteed* to have at least one pair?



Let: *A* be the set of socks in my hand
B be the set {white, black, stripes}

Define the mapping function $f : A \to B$ (*f* assigns a color to each sock.)

We want to know how large *A* must be in order that *f cannot* be one-to-one. (In other words, in order that at least two socks *must* map to the same color.)

The pigeonhole principle tells us that if $|A| > |B|$, then we'll have achieved our goal. $|B| = 3$. So if we pull 4 or more socks, we are guaranteed a pair.

The pigeonhole principle is an example of a ***counting argument***. Such arguments generally support claims of the sort, "Some number of something must exist," or "Some number of something must not exist". The argument is based on counting the elements of one or more sets and typically comparing them. These proofs are often not "constructive". They don't tell us what objects exist. Just that some do.

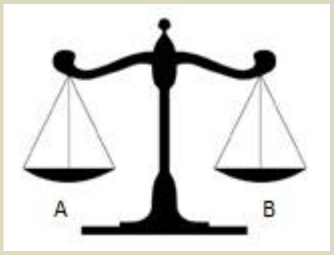
Nifty Aside

The first formalization of the pigeonhole principle is generally attributed to the German mathematician Peter Gustav Lejeune Dirichlet (1805 -1859), who called it the drawer principle.

It can be used to prove claims that would generally be regarded as obvious (such as the one about pigeons and holes).

But it can also be used to solve problems whose solutions are substantially less obvious:

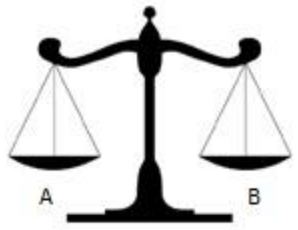
Suppose that we have 15 coins. We know that exactly 14 of them have the same weight. The odd one may weigh more or less than the other 14. We want to find the odd one and discover whether it is heavy or light. We are given a balance scale. At each weighing, we get one of three readings: Trays A and B have equal weight, A weighs more than B, or B weighs more than A. We can weigh coins on this scale by putting any number of coins on each of the two plates. Can we solve the problem in three weighings?



Problems

1. Suppose that there are 10 floors in our building. Assume that everyone gets off somewhere. The pigeon hole principle tells us that, if there are at least ____________ people on the elevator, there must be at least one floor where multiple people get off. Fill in the blank.

2. Suppose that we have 5 coins. We know that exactly 4 of them have the same weight. The odd one may weigh more or less than the other 4. We want to find the odd one and discover whether it is heavy or light. We are given a balance scale. At each weighing, we get one of three readings: Trays A and B have equal weight, A weighs more than B, or B weighs more than A. We can weigh coins on this scale by putting any number of coins on each of the two plates. How many weighings must we be prepared to do in order to guarantee that we know the answer? Which of the following is correct:



a) The Pigeonhole Principle tells us that a single weighing can't be enough but it doesn't rule out a 2 weighing solution.
b) The Pigeonhole Principle tells us that 2 weighings can't be enough but it doesn't rule out a 3 weighing solution.
c) The Pigeonhole Principle tells us that 3 weighings can't be enough.

Extended Pigeon Hole Principle

Sometimes even more than two pigeons will have to share a home. To see exactly how many roommates there must be, we first define the ***ceiling*** function from the reals to the integers:

ceiling: $\mathbf{R} \rightarrow \mathbf{N}$: *ceiling*(x) = the smallest integer n such that $x \leq n$

Although we don't need it right now, we'll pause and define another useful function:

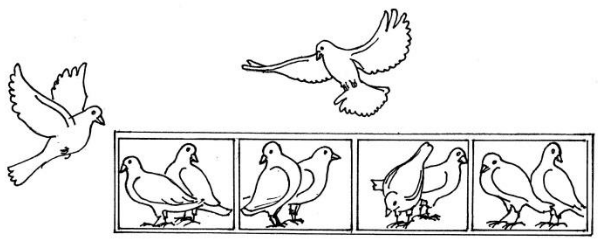
floor: $\mathbf{R} \rightarrow \mathbf{N}$: *floor*(x) = the largest integer n such that $n \leq x$

We can write *ceiling*(x) as $\lceil x \rceil$ and *floor*(x) as $\lfloor x \rfloor$

$\lceil 5.8 \rceil = 6$	$\lceil 5.001 \rceil = 6$	$\lceil 6 \rceil = 6$
$\lfloor 5.8 \rfloor = 5$	$\lfloor 5.001 \rfloor = 5$	$\lfloor 6 \rfloor = 6$

The extended pigeonhole principle:

If you try to stuff k pigeons into n holes there must be at least $\lceil k/n \rceil$ pigeons in some hole.



If we try to stuff 26 pigeons into 8 holes, there must be at least $\lceil 26/8 \rceil = \lceil 3.25 \rceil = 4$ pigeons in some hole.

Notice that this claim is true even in the trivial case in which there are more holes than pigeons.

If we try to stuff 5 pigeons into 8 holes, there must be at least $\lceil 5/8 \rceil = \lceil 0.625 \rceil = 1$ pigeon in some hole.

Problems

1. Suppose that there are 5 floors in our building and 12 people on the elevator. Assume that everyone gets off somewhere. Which of the following is true:

a) There must be some floor on which no one gets off.
b) There must be some floor on which at least 3 people get off.
c) There must be some floor on which at least 5 people get off.
d) It's possible that no more than 2 people get off on any floor.
e) We can't make any statement about how many people will get off on each floor.

Hash Functions

Now let's look at a very important application of the Pigeonhole Principle.

We know that one way to store a set is as a bit vector. To store a set S drawn from some universe U, we need $|U|$ bits.

For example, in this case, $|U| = 9$. $S = \{55, 99\}$.

11	22	33	44	55	66	77	88	99
				●				●

Now suppose that we want not just to indicate that some element x is in S. We also want to associate some information with it.

Suppose that U is the set of possible account numbers for a bank. S is the set of actual accounts today. We want to look up a number and, if it is an account, extract information about it.

We could extend the bit vector representation idea and assign more storage to each element of U.

1001	1002	1003	1004	1005	1006	1007	0008	0009	0010	0011	0012
	info							info			info

It may take several words of memory to store the information for each element of S. Yet, in real applications, it often happens that many (and sometimes even most) of the cells in a vector like this are empty. Now we've wasted a lot of memory storing blanks.

Let *Boxes* be the set of storage locations that we are going to use to store our information. So far, we have assumed that there is one box for every element of U.

Let *mapping* be a function that maps elements of S to elements of *Boxes*. So far we have that:

$|Boxes| = |U|$

$mapping : U \rightarrow Boxes$: $mapping(x)$ maps x to its uniquely corresponding cell in *Boxes*

The function *mapping* is one-to-one and onto. Each element of U (and thus of any particular S drawn from U) has exactly one cell in *Boxes* where it can go.

But suppose that we knew that only about 10% of the cells in *Boxes* could be expected to contain a value at any one time. We might define a vector *FewerBoxes* that had about 10% of the cells that *Boxes* had. Now we need to define a mapping function that puts each element of *U* into some cell in FewerBoxes.

The Pigeonhole Principle tells us that the new mapping function can no longer be one-to-one. There is no way around the fact that several elements of *U* will map to the same place.

So we'll have:

$|FewerBoxes| = |U| / 10$
$mapping : U \rightarrow FewerBoxes$:
$mapping(x)$ maps at least 10 different values of x to the same cell in FewerBoxes

In our banking application, we might have:

mapaccount (accountnumber) = accountnumber / 10

So: 1001, 1002, … , 1009 will all map to 100.
1010, 1011, 1012, …, 1019 will all map to 101, and so forth.

Let's describe the process more generally. We begin by choosing a size for *FewerBoxes* (generally based on some knowledge about how densely packed the elements of *S* are likely to be). We also assign every element of *U* a unique number, which we'll call a ***key***.

Then we define a function called a ***hash function***. The job of the hash function is to map every key to exactly one cell in *FewerBoxes*. We'll call *FewerBoxes* a ***hash table***. The *mapaccount* function that we defined above is an example of a hash function.

A well-designed hash function should be onto: We don't want to waste any of the cells of *FewerBoxes* by making them unreachable by any element of *S*.

But, as the Pigeonhole Principle tells us, if $|FewerBoxes| < |U|$, then no hash function that maps elements of *U* to elements of *FewerBoxes* can be one-to-one.

So the signature property of hash functions is that they are not one-to-one. This is both:

- their strength (because we can use a lot less storage space), and
- their weakness (because ***collisions*** may happen when multiple elements of *U* map to the same place).

So what about collisions? Two things:

- We must deal with them. Often, when they occur, what gets stored in the overcrowded cell in *FewerBoxes* is a link to a list of the inhabitants. That takes some extra space and some extra time, but the extra space should be a lot less than what would have been required if we'd stuck with the full *Boxes* vector.

- But we should attempt to minimize them by a wise choice of a hash function. We try to design one that spreads out the elements of *U* as uniformly as possible onto *FewerBoxes*. Two common, general purpose hash functions are:

  hash(*key*) = *key* mod |*FewerBoxes*|

  hash(*key*) = choose the appropriate number of bits out of the middle of *key*

Problems

1. We want to store information about 10-digit US phone numbers. There are, in principle, 10^{10} possible numbers. So we could store our information in a vector with 10^{10} cells. But many phone numbers are not in use and we cannot afford the wasted space. So we want to use a hash table. Suppose that the following hash functions have been proposed. All of them map the original 10^{10} numbers (which serve as keys) to a set of 10^7 hash values. Which one is likely to minimize the number of collisions? (Hint: Think a little bit about the structure of phone numbers. What we want to do is to spread numbers out as evenly as possible into the cells of the hash table):

a) Throw away the last three digits of each number. So, for example, 5127780123 becomes 5127780 and 1006672345 becomes 1006672.
b) Throw away the first three digits of each number. So, for example, 5127780123 becomes 7780123 and 1006672345 becomes 6672345.
c) Throw away digits 4, 5, and 6. So for example, 5127780123 becomes 5120123 and 1006672345 becomes 1002345.

Compositions, Inverses and the Identity

Function Composition

Just as we can compose relations, we can compose functions. Define function ***composition***:

Given: $f : A \rightarrow B$ and $g : B \rightarrow C$

We have: $g \circ f(x) : A \rightarrow C \quad : \quad g \circ f(x) = g(f(x))$

As with relations, we read composition right to left. Apply *f*; then apply *g*. The parenthesized notation that we use for functions now makes it clear that we're doing that. Read $g(f(x))$ as, "*g* of *f* of *x*."

fatherof(*fatherof*(x)) = *grandfatherof*(x)

SS#(*managerof*(x)) : *Employees* → *NineDigitNumbers*

When we write function compositions, we can use both prefix and infix notations (as well as any other notations, such as the postfix one usually used take powers) and we can combine them. Notice that you've been doing this for years. We've just formalized the process.

$$\begin{aligned}(succ((2+3)!))^2 &= (succ(5!))^2\\ &= (succ(120))^2\\ &= 121^2\\ &= 14{,}641\end{aligned}$$

As was the case with relations, not all compositions are well defined. For $g \circ f$ to exist, it must be the case that the range (output) of *f* is a subset of the domain (input) of *g*.

This composition isn't defined:

managerof(*SS#*(x)) + 1

The codomain of *SS#*(x) is *NineDigitNumbers*. We can't ask for the manager of 456900876.

Problems

1. Define: $f : \mathbf{N} \to \mathbf{N}$: $f(k) = k^2$
$g : \mathbf{N} \to \mathbf{N}$: $g(k) = k + 2$

(Part 1) What is $f \circ f$ (3)?

(Part 2) What is $f \circ g$ (3)?

(Part 3) What is $g \circ f$ (3)?

2. Assume reasonable definitions of the domain and codomain of each of these functions. Mark each of them as well-defined or not well-defined.

a) *motherof* ∘ *height*
b) *height* ∘ *motherof*
c) *height* ∘ *height*
d) *motherof* ∘ *motherof*

3. Are we guaranteed that, if f and g are functions, $f \circ g$ and $g \circ f$ are also functions? Which one of the following statements is true:

a) Yes, both $f \circ g$ and $g \circ f$ must also functions.
b) Yes, both $f \circ g$ and $g \circ f$ must also functions.
c) No. While both of them must be relations, it is possible that one or both of them might not be a function.
d) No. It is possible that one or both of them doesn't even exist.

4. What about the one-to-one and onto properties of composed functions? Assume that there exists some set X that is both the codomain of g and the domain of f.

(Part 1) Suppose that f and g are both one-to-one and onto functions. Which one of the following is true:

a) $f \circ g$ must also be a function that is both one-to-one and onto.
b) $f \circ g$ must also be a function that is one-to-one but not necessarily onto.
c) $f \circ g$ must also be a function that is onto but not necessarily one-to-one.
d) We know $f \circ g$ is a function but it might be neither one-to-one nor onto.
e) We don't even know whether $f \circ g$ is a function.

(Part 2) Suppose that f and g are both functions that are one-to-one but not necessarily onto. Which one of the following is true:

a) $f \circ g$ must also be a function that is both one-to-one and onto.
b) $f \circ g$ must also be a function that one-to-one but not necessarily onto.
c) $f \circ g$ must also be a function that is onto but not necessarily one-to-one.
d) We know $f \circ g$ is a function but it might be neither one-to-one nor onto.
e) We don't even know whether $f \circ g$ is a function.

(Part 3) Suppose that *f* and *g* are both functions that are onto but not necessarily one-to-one. Which one of the following is true:

a) $f \circ g$ must also be a function that is both one-to-one and onto.
b) $f \circ g$ must also be a function that one-to-one but not necessarily onto.
c) $f \circ g$ must also be a function that is onto but not necessarily one-to-one.
d) We know $f \circ g$ is a function but it might be neither one-to-one nor onto.
e) We don't even know whether $f \circ g$ is a function.

Why Function Composition is a Big Deal for Programmers

Most useful programs compute intermediate results, which they then combine to derive a final answer.

Recall the formula for computing body mass index (BMI). We can write it as a function:

$$BMI(p) = (weight(p) / height(p)^2) \cdot 703$$

We've written our definition of *BMI* as a composition of five other functions:

- *weight* – which returns (presumably after looking up *p* in some appropriate structure) *p*'s weight.
- *height* – similarly
- *square*
- *division*
- *multiplication*

It's straightforward to take this functional definition of *BMI* and convert it to code. Here it is in Python:

```
def BMI(p):
     return((weight(p) / height(p) **2) * 703)
```

Here's an example of the use of function composition in Excel:

```
=(G3/165)*19 + (F3/196)*19 + (M3/1400)*15 +(I3/267)*35+ K3+ L3
```

This expression composes four divisions, four multiplications, and five additions. It computes final weighted scores for one of our classes.

When we use primitive functions, like addition and multiplication, we usually exploit the standard infix notations for them. When we compose functions that we have had to define, we generally use explicit, prefix function notation, so it's even more obvious that we're composing them.

Suppose that we want to send a letter to the manager of the best salesperson in the company's most profitable division. To figure out who that manager is, we could write (assuming that our corporate database has implementations of the functions that we need):

managerof(*bestsellerin*(*mostprofitabledivisionof*(*entirecompany*)))

Nifty Aside

In most programming environments, we aren't limited to function composition as a way of combining smaller chunks of code into larger ones. We can also use sequences of commands, conditionals, loops and variables whose values we can change. For example, recall the code that we wrote for the 3n+1 problem:

```
def threen(value):
    while value is not equal to 1 do:
        if value is even then set value to value/2
                              else set value to 3 * value + 1
```

But there are purely functional programming languages and they have some die hard fans.

The Inverse of a Function

The inverse of a function is defined in exactly the way we have already defined the inverse of a relation:

Let: $f \subseteq A \times B$ be a function that maps $A \rightarrow B$.

Then: $f^{-1} \subseteq B \times A$ is the inverse of f.

$$f^{-1} = \{(b, a) : b = f(a)\}$$

Define the following function, which converts temperatures from centigrade to Fahrenheit:

CtoF : $\mathbf{Z} \rightarrow \mathbf{Z}$: $CtoF(d) = \frac{9}{5}d + 32$

If we have a formula for f, we can compute its inverse. Let $f(x) = y$. We need a function that, given y, returns the corresponding value of x. So we solve for y.

The inverse of *CtoF* converts from Fahrenheit to centigrade. To determine it:

CtoF(*d*):

$$\begin{aligned} y &= \tfrac{9}{5}d + 32 \\ y - 32 &= \tfrac{9}{5}d \\ \tfrac{5}{9}(y - 32) &= d \end{aligned}$$

So *FtoC* = *CtoF*$^{-1}$ is:

FtoC : $\mathbf{Z} \rightarrow \mathbf{Z}$: $FtoC(y) = \frac{5}{9}(y - 32)$

Problems

1. Let *predecessor* and *successor* be defined on **N**. Which of the following expressions is not true?

a) *predecessor* = *successor*$^{-1}$
b) *predecessor*(x) = (x + 1) – 1
c) *successor* = *predecessor*$^{-1}$
d) *successor* = *successor* ∘ *successor*$^{-1}$ ∘ *successor*
e) *successor* = *predecessor*$^{-1}$ ∘ *successor* ∘ *predecessor*

2. Let O = the set of odd natural numbers. Define:

$$f : \mathbf{N} \rightarrow O \quad : \quad f(x) = 2x + 1$$

(Part 1) True or false:

a) $f^{-1}(7) = 3$
b) $f^{-1}(3) = 7$
c) $f^{-1}(22) = 10$

(Part 2) Which of the following is a correct definition of f^{-1}:

a) $f^{-1} : \mathbf{N} \rightarrow \mathbf{N} : f^{-1}(x) = \frac{x}{2} - 1$
b) $f^{-1} : O \rightarrow \mathbf{N} : f^{-1}(x) = \frac{x-1}{2}$
c) $f^{-1} : \mathbf{N} \rightarrow \mathbf{N} : f^{-1}(x) = \frac{x-1}{2}$

The Inverse Isn't Always a Function

Every function has an inverse. But not every such inverse is a function. We'll say that a function f is ***invertible*** if and only if f^{-1} is also a function.

gpa: *Students* $\rightarrow$ *Scores* is a function.

gpa^{-1}: *Scores* $\rightarrow$ *Students* is not a function, since many students have the same gpa.

Define *clockhands* : $\mathbf{N} \rightarrow \{1, 2, 3, \ldots, 12\}$: *clockhands*(n) = n mod 12

clockhands describes the location of the hour hand of a clock, started at the top, after some number of hours have elapsed (except that, for some reason, clocks render 0 as 12).

clockhands is a function since every integer has a unique remainder when divided by 12.

But *clockhands*$^{-1}$ is not a function. For example, since 1, 13, 25, and many other numbers all have a remainder of 1 when divided by 12, *clockhands*$^{-1}$ contains all of the following elements:

(1, 1), (1, 13), (1, 25), …

In general, functions that throw away information aren't invertible.

ClockHands^{-1} is not a function because *clockHands* throws away information. Clocks don't tell us how many half days have elapsed. So *clockHands*$^{-1}$ cannot return a unique value.

Nifty Aside

The fact that many functions in modular arithmetic are not invertible plays an important role in modern cryptography.

Problems

1. Consider the function *square*(x) = x^2. It makes sense to define *square* on the reals, on the rationals, on the integers, or on the natural numbers. Now let's consider *squareroot*:

squareroot = *square*$^{-1}$: $\{(x, y) : x = y^2\}$

We know that *squareroot* is a relation. But is it a function? In particular, can we use functional notation for it? Can we write:

$$\sqrt{x}$$

(Part 1) Define *square* on the integers:

square: $\mathbf{Z} \rightarrow \mathbf{Z}$: *square*(x) = x^2

Then *squareroot* is also a subset of $\mathbf{Z} \times \mathbf{Z}$. Which one of the following claims is true:

a) *squareroot* is a function.
b) *squareroot* is not a function because, while it can return at least one value for each element of its domain, that value may not be unique.
c) *squareroot* is not a function because there are some elements of its domain for which it can return no value, even though, where there is a value, it is unique.
d) *squareroot* is not a function because there are elements of the domain for which no value can be returned and there are others for which multiple values would be returned.

(Part 2) Now define *square* on the reals:

square: $\mathbf{R} \rightarrow \mathbf{R}$: *square*(x) = x^2

Then *squareroot* is also a subset of $\mathbf{R} \times \mathbf{R}$. Which one of the following claims is true:

a) *squareroot* is a function.
b) *squareroot* is not a function because, while it can return at least one value for each element of its domain, that value may not be unique.
c) *squareroot* is not a function because there are some elements of its domain for which it can return no value, even though, where there is a value, it is unique.
d) *squareroot* is not a function because there are elements of the domain for which no value can be returned and there are others for which multiple values would be returned.

(Part 3) But this is crazy. Even the simplest calculators have a $\sqrt{}$ button. It is a function; it returns a unique value for every input. What's going on?

The answer is that *squareroot* becomes a function if we choose its domain and codomain correctly. Define, for some set *X*:

$$square: X \rightarrow X \ : \ square(x) = x^2$$

Then *squareroot* is also a subset of $X \times X$. Which one of the following values for *X* makes *squareroot* a function?

a) **Q** (the rational numbers)
b) $\mathbf{Q}^+$ (the positive rationals)
c) $\mathbf{R}^+$ (the positive reals)
d) $\mathbf{Z}^+$ (the positive integers)
e) **C** (the complex numbers)

2. Suppose that we manage an office building. Consider: $f : Offices \rightarrow OfficeThermostats$.

(Part 1) Assume that each office has its own thermostat. Mark each of the following statements true or false:

a) *f* is one-to-one.
b) *f* is onto.
c) f^{-1} is a function.

(Part 2) Now assume that there is one thermostat for every three offices. Mark each of the following statements true or false:

a) *f* is one-to-one.
b) *f* is onto.
c) f^{-1} is a function.

3. Recall the Social Security lookup function:

$$SS\# : EmployeeIDs \rightarrow NineDigitNumbers : SS\#(e) = e\text{'s social security number}$$

Mark each of the following statements true or false:

a) *SS#* is one-to-one.
b) *SS#* is onto.
c) $SS\#^{-1}$ is a function.

When Is f^{-1} a Function?

Let f be a function that maps from $A \rightarrow B$. Then f is invertible (i.e., f^{-1} is a function) if and only if f is both one-to-one and onto.

We can prove this claim as follows.

- Assume that f^{-1} is a function (that maps from $B \rightarrow A$) and show that f is both one-to-one and onto:
  - Prove that f is one-to-one: We do this by contradiction.

    Suppose that f were not one-to-one and there were two distinct values x and y such that:

    $f(x) = f(y) = k$ (k is just the name we are giving to this value)

    Then: since $(x, k) \in f$, (k, x) must be an element of f^{-1}, and
    since $(y, k) \in f$, (k, y) must be an element of f^{-1}.

    But this contradicts the assumption that f^{-1} is a function (that maps each element of its domain to a unique element of its codomain).

  - Prove that f is onto: Let b be an arbitrary element of B. Since f^{-1} is a function, it maps b to some unique element of A. So $(b, a) \in f^{-1}$. Thus $(a, b) \in f$. So f maps something to b. Since b is an arbitrary element of B, we have that every element of B is the image of some element of A. Thus f is onto.

- Assume that f is both one-to-one and onto and show that f^{-1} is a function: Let b be an arbitrary element of B. Since f is both one-to-one and onto, there is exactly one element a of A such that $(a, b) \in f$. That means that there is exactly one element a of A such that $(b, a) \in f^{-1}$. Thus f^{-1} is a function.

When f^{-1} is not a function but we want to compute it, we can let it return a set (possibly empty, possibly containing a single element, and possibly containing multiple elements). We just need to know that it must do that.

Sometimes the fact that a function's inverse is not itself a function (and thus a set of values must be returned) is precisely what mak es the function useful.

Recall the Soundex system, which takes a name and converts it to a four-symbol code that is designed so that similar sounding names will map to the same code. Once one has the code for one's name, one can run the system backwards to retrieve other names that map to the same code. So we defined:

SoundexCode $\subseteq$ *Names* $\times$ *Codes* = {(*name*, *c*) : *c* is the code produced by Soundex for *name*}

SoundexCode is a function. It returns a unique code for every input string. Its inverse, however isn't a function. One code can return many names. And that's exactly what we want. Soundex would be pretty useless if the process of computing $SoundexCode^{-1} \circ SoundexCode$ just got back our original name. We want a set of similar sounding names, and that's what we get.

By the way, even before trying out the Soundex system, we know that $SoundexCode^{-1}$ could not be a function. The Pigeonhole Principle tells us that the function *SoundexCode* cannot be one-to-one since there are way more names than there are codes. And we just proved that f^{-1} is only a function when f is one-to-one.

The Identity Function

Let A be any nonempty set. Define:

$$i : A \to A \qquad : \; i(x) = x$$

We'll call i the ***identity*** function. It maps every element of A to itself. Notice that i is identical to the identity relation that we've already defined:

$$I = \{(a, a) : a \in A\}$$

The identity function appears to be fairly useless since nothing changes after it's been applied. But that's important. Sometimes we want to be able to show that, after applying one or more functions to an arbitrary object, we will land back where we started.

One important case arises when we compose a function with its inverse:

if f is invertible, $f^{-1} \circ f = i$

To see why this must be so, let (a, b) be some element of f. Then $(b, a) \in f^{-1}$. Since f is a function, b is the only element related to a. So $f(a) = b$. Since f^{-1} is a function, the only element related to b is a. So $f^{-1}(f(a)) = a$.

Recall that *CtoF* converts temperatures from centigrade to Fahrenheit:

$$CtoF \; : \; \mathbb{Z} \to \mathbb{Z} \qquad : \qquad CtoF(d) = \tfrac{9}{5}d + 32$$

Its inverse converts from Fahrenheit to centigrade. $FtoC = CtoF^{-1}$ is:

$$FtoC \; : \; \mathbb{Z} \to \mathbb{Z} \qquad : \qquad FtoC(d) = \tfrac{5}{9}(d - 32)$$

So $FtoC(CtoF(d)) = \tfrac{5}{9}(\tfrac{9}{5}d + 32 - 32) = \tfrac{5}{9}(\tfrac{9}{5}d) = d$.

Problems

1. Let the universe U be **N** (the natural numbers). Let's look at what happens when we compose set functions. Which of the following compositions is equal to i (the identity function)?

a) $f_1 : \wp(\mathbf{N}) \rightarrow \wp(\mathbf{N}) \quad : \quad f_1(A) = (A \cup U) - U$
b) $f_2 : \wp(\mathbf{N}) \rightarrow \wp(\mathbf{N}) \quad : \quad f_2(A) = (A - U) \cup A$
c) $f_3 : \wp(\mathbf{N}) \rightarrow \wp(\mathbf{N}) \quad : \quad f_3(A) = (A \cap U) - A$
d) $f_4 : \wp(\mathbf{N}) \rightarrow \wp(\mathbf{N}) \quad : \quad f_4(A) = (A - U) \cup (U - A)$

2. Let A and B be some sets and let f be some function: $f : A \rightarrow B$.

The function f is not necessarily invertible (in other words, while it has an inverse, its inverse might not be a function).

Which of the following claims is true:

a) $f^{-1} \circ f$ must be equal to $f \circ f^{-1}$.
b) $f^{-1} \circ f$ must not be equal to $f \circ f^{-1}$.
c) For all x, if $f(x) = 0$ then $f^{-1}(x)$ must not equal 0.
d) $f^{-1} \circ f$ might be equal to $f \circ f$.
e) $f \circ f^{-1} \circ f$ must be equal to f^{-1}.