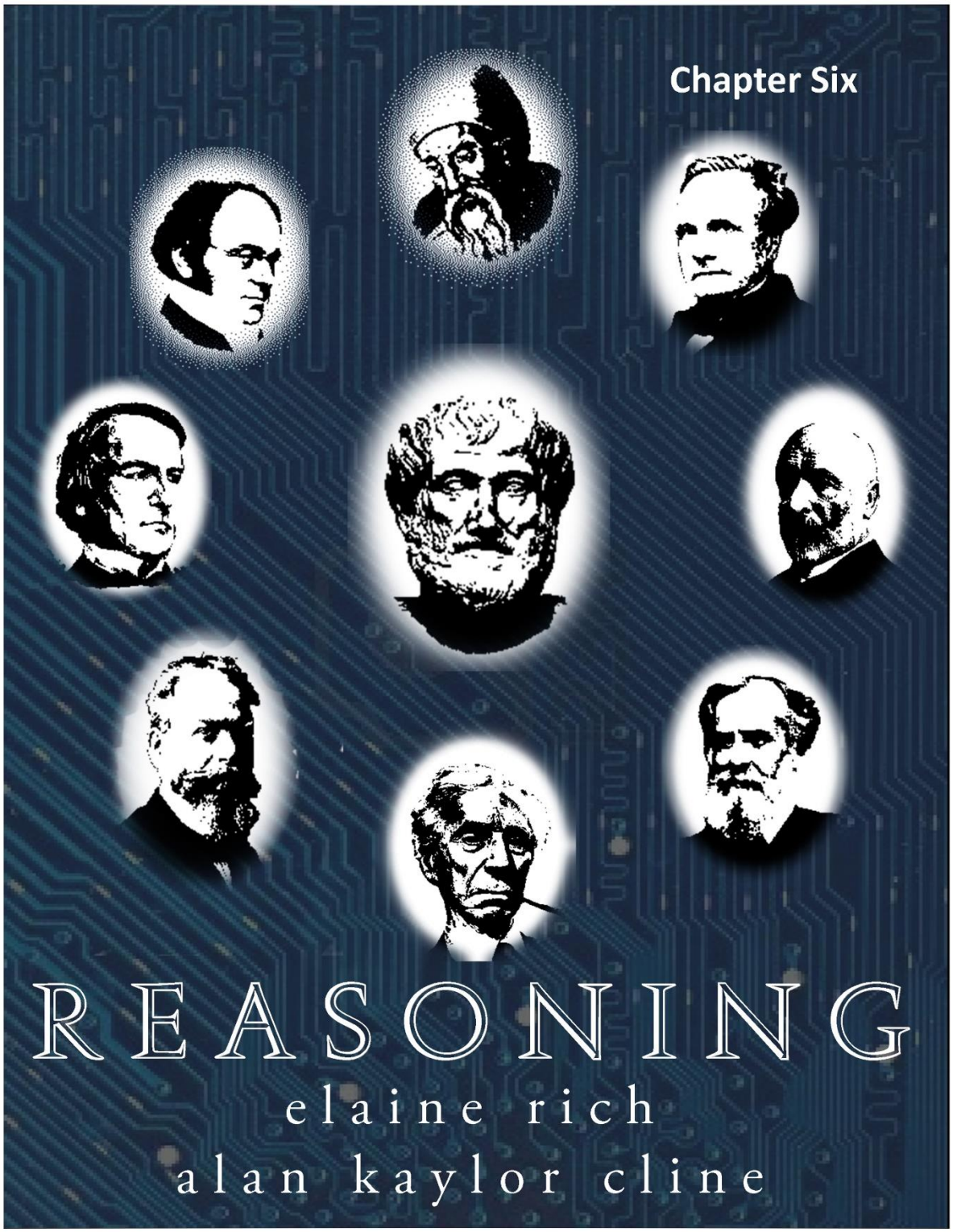

Chapter Six
REASONING
elaine rich
alan kaylor cline

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 6
PRACTICE REPRESENTING CLAIMS IN LOGIC

Elaine Rich
Alan Kaylor Cline

The University of Texas at Austin

Image credits:

Girl with cat: © Lynda Trader, 2007.

Bear with rifle: © Lynda Trader, 2007

Potatoes, Rice, Beans:

Balloons: http://www.balloondealer.com/app/images/Balloons21.jpg

Male bird song speeds: http://sites.sinauer.com/animalcommunication2e/chapter08.04.html

White peacock: © David Rich, 2014.

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

Practice Representing Claims in Logic

Some Key Ideas

Now We Need Practice

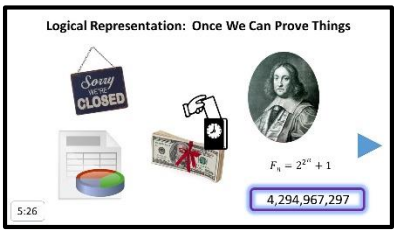
We've already covered everything you need to know to translate back and forth between:

- Ideas (stated in English or perhaps some other way)

  and

- Predicate logic statements

But this is a hard thing to do, particularly in the English to logic direction. So it makes sense to get some more practice. We're coming back to it now, after getting some experience at using logical identities and constructing proofs, so that we'll be able to see how both to encode knowledge and to reason with it.



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

Suppose that we are in charge of making sure everyone in our company complies with all the official rules. We want to represent the fact that there's someone who doesn't know all the rules (in which case, it's time for another training session).

First we have to choose our predicates. Let's assume that we want to handle some other issues as well. In particular, that there are things besides rules that someone might or might not know. So define:

Rule(x): True if x is an official rule.
Knows(x, y) True if x knows y.

We can represent our statement in either of two fairly natural ways:

[1] $\exists x\ (\neg(\forall y\ (Rule(y) \rightarrow Knows(x, y))))$

[2] $\exists x\ (\exists y\ (Rule(y) \wedge \neg Knows(x, y)))$

The natural reading of [1] is that there exists someone who doesn't know all the rules. The natural reading for [2] is that there exists some person/rule pair such that the person doesn't know the rule.

You should practice using our identity rules to show that [1] and [2] are equivalent.

Weaker Statements/Stronger Statements

The logical framework that we've just described gives us the tools we need to make statements that are exactly as strong as the facts warrant:

Strongest: [1] $\forall x\,(Q(x) \wedge R(x))$

[2] $\forall x\,(Q(x))$

[3] $\forall x\,(P(x) \vee Q(x))$ [4] $\forall x\,(P(x) \rightarrow Q(x))$

Weakest: [5] $\forall x\,((P(x) \wedge R(x)) \rightarrow Q(x))$

There's also what may appear to be an even weaker claim:

[6] $\exists x\,(Q(x))$

But this one isn't so straightforward. We'll look at existential quantification on the next slide.

Meanwhile, in examples [1] – [5], we've said:

[1] Both Q and R are true of every x.

[2] Q is true of every x. We aren't committing to anything about R.

[3], [4] Since [4] can be rewritten (using Conditional Disjunction) as $\forall x\,(\neg P(x) \vee Q(x))$, we see that [3] and [4] are equally strong/weak. Which one we pick depends just on whether we've defined P in a positive or a negative way (and thus whether we wish to assert it or its complement).

[5] $Q(x)$ can only be guaranteed to be true if both $P(x)$ and $R(x)$ are true.

Here's an important way to view versions [4] and [5]: They contain "guards" that limit the circumstances in which their conclusions are guaranteed to be true. So, in [4], Q is only required to be true for those values of x for which P is true. And, in [5], both P *and* R must be true in order to guarantee the truth of Q for a given x.

Big Idea

Given an implication of the from $P \rightarrow Q$, we can think of P as a ***guard***: it limits the circumstances in which we can guarantee that Q must be true.

Consider the following statements about people entering a big deal, special event. Define:

Ticket(x): True if x must have a ticket to enter.
ID(x): True if x must have a photo ID to enter.
Wrist(x): True if x must have a wristband to enter.
SignIn (x): True if x must sign in and be photographed at a special sign-in window.
Student(x): True if x has a special student ticket.
HasID(x): True if x is able to present a student ID.

Strongest:

[1] $\forall x\ (Ticket(x) \wedge ID(x) \wedge SignIn(x))$

[2] $\forall x\ (Ticket(x) \wedge SignIn(x))$

[3] $\forall x\ ((Ticket(x) \vee Wrist(x)) \wedge SignIn(x))$ [4] $\forall x\ (Ticket(x) \wedge (Student(x) \rightarrow SignIn(x)))$

Weakest:

[5] $\forall x\ (Ticket(x) \wedge ((Student(x) \wedge \neg HasID(x)) \rightarrow SignIn(x)))$

[1] says that everyone must have a ticket and an ID and must go to the sign-in window. [2] relaxes the ID requirement. [3] allows the option of a wristband or a ticket. [4] relaxes the requirements in a different way. It says that one only needs to sign in *if* one has a special student ticket. And [5] is even weaker. It says that one must sign in only if one *both* has a special student ticket *and* is not carrying a student ID.

Problems

1. Consider the following claim:

[1] $\forall x\ (S(x) \vee T(x))$

Which of the following sets of premises is strong enough to enable us to prove [1]:

a) $\forall x\ ((P(x) \wedge (Q(x) \wedge R(x))) \rightarrow S(x))$
$\forall x\ (P(x) \vee R(x))$

b) $\forall x\ ((P(x) \wedge (Q(x) \vee R(x))) \rightarrow S(x))$
$\forall x\ (P(x) \vee Q(x))$

c) $\forall x\ ((P(x) \vee (Q(x) \vee R(x))) \rightarrow S(x))$
$\forall x\ (P(x) \vee R(x))$

d) $\forall x\ (P(x) \vee (Q(x) \vee R(x)))$
$\forall x\ (P(x) \rightarrow T(x))$

2. Consider the following claim:

[2] $\forall x\ (R(x) \rightarrow T(x))$

Which of the following sets of premises is too weak to enable us to prove [2]:

a) $\forall x\ (P(x) \wedge T(x))$
$\forall x\ (Q(x) \rightarrow P(x))$

b) $\forall x\ (P(x) \rightarrow \neg R(x))$
$\forall x\ (P(x))$

c) $\forall x\ (P(x) \rightarrow \neg R(x))$
$\forall x\ (P(x) \vee Q(x))$

d) $\forall x\ (Q(x) \wedge T(x))$
$\forall x\ (P(x) \rightarrow Q(x))$

3. Consider the following claim:

[3] $\forall x\ (\neg Q(x))$

Which of the following sets of premises is so strong that [3] cannot be true:

$\forall x\ (P(x) \vee Q(x))$
$\forall x\ (R(x) \rightarrow \neg P(x))$

$\forall x\ (P(x) \vee Q(x))$
$\forall x\ (\neg P(x))$

$\forall x\ (P(x) \vee Q(x))$
$\forall x\ (\neg P(x) \vee \neg R(x))$

$\forall x\ (P(x) \vee Q(x))$
$\forall x\ (P(x) \rightarrow \neg Q(x))$

Existentials in Implications

You might be tempted to think that we should have added an even weaker line to our scale on the previous slide:

Even weaker: [6] $\exists x\,(Q(x))$

It's true that, almost all the time, asserting that there's at least one thing of which Q is true is weaker (generally a lot weaker) than asserting that Q is true of everything. The exception is when the universe about which we are making our claim is, in fact, empty. In that case, the universal claim is trivially true, while the existential claim is stronger (and false).

Recall the example of my non-card-playing cat Lucy:

[1] Lucy has won every game of solitaire she's ever played.

$\forall x\,(SolitaireGame(x) \wedge Played(Lucy, x) \rightarrow Won(Lucy, x))$

[2] There is a solitaire game, played by Lucy, that she's won.

$\exists x\,(SolitaireGame(x) \wedge Played(Lucy, x) \wedge Won(Lucy, x))$

I'm pretty sure that Lucy has never played solitaire. So [1] is trivially true. Yet [2], the seemingly weaker existential claim, is false.

That exception aside, however, existential claims are generally weak.

Does it make sense to weaken them further, as we've done with universal ones, by adding guards? For example, rather than saying:

[1] $\exists x\,(Q(x))$

could we make the weaker (guarded claim):

[2] $\exists x\,(P(x) \rightarrow Q(x))$

Of course we can say that. It's a legal logical expression. But it's hard to figure out what it means. Let's rewrite it, using Conditional Disjunction, so we have:

[3] $\exists x\,(\neg P(x) \vee Q(x))$

Perhaps this is true and we might want to say it. This form makes our claim more clear than [2] does.

Using an implication inside an existential quantifier is even stranger when it's negated. Suppose we write:

[4] $\neg\exists x\,(P(x) \rightarrow Q(x))$

This makes little sense: Let's simplify:

[5]	$\neg\exists x\,(\neg P(x) \vee Q(x))$	Conditional Disjunction	[4]
[6]	$\forall x\,\neg(\neg P(x) \vee Q(x))$	Quantifier Exchange	[5]
[7]	$\forall x\,(\neg\neg P(x) \wedge \neg Q(x))$	De Morgan	[6]
[8]	$\forall x\,(P(x) \wedge \neg Q(x))$	Double Negation	[7]

It's hard to think of a case in which [8] isn't the more natural thing to say.

Big Idea

Be very careful if you find yourself using an existentially quantified variable as a guard in an implication. It rarely does what you want it to do.

Recall our ***Bad Movie*** example. We want to claim that there's a movie that everyone in our class hates. Define:

Movie(x): True if x is a movie.
InClass(x): True if x is in our class
Hates(x, y): True if x hates y.

A good way to represent this claim is:

[1] $\exists x\,(Movie(x) \wedge \forall y\,(InClass(y) \rightarrow Hates(y, x)))$

(Notice that we have *Hates*(y, x), with y coming before x, because, in this expression, x is a movie, and we've defined *Hates* so that the hater is the first argument and the hated thing is the second one.)

But consider an alternative that looks as though it might be correct:

[2] $\exists x\,(\forall y\,((Movie(x) \wedge InClass(y)) \rightarrow Hates(y, x)))$

Read [2] as, "There exists an x such that if x is a movie then everyone in the class hates it." But notice that, if there exists even one non-movie, then this claim is trivially true. To see why, let *k* be the non-movie. Then *Movie*(*k*) is false (*F*). So, generalizing a bit, we have:

[3] $(F \wedge \langle\text{something}\rangle) \rightarrow \langle\text{conclusion}\rangle$

Problems

1. Suppose that we want to represent the fact that every store has a product that no one wants to buy. Define:

Store(*x*): True if *x* is a store.
Product(*x*): True if *x* is a product.
Carries(*x*, *y*): True if *x* carries *y*. (We assume that "carries" is the intended meaning of the very general word "has" in this case.)
WtB(*x*, *y*): True if *x* wants to buy *y*.

Consider the following logical expressions:

I. $\forall x\,(Store(x) \rightarrow \exists y\,((Product(y) \wedge Carries(x, y)) \rightarrow \neg\exists z\,(WtB(z, y))))$
II. $\forall x\,(Store(x) \rightarrow \exists y\,(Product(y) \wedge Carries(x, y) \wedge \forall z\,(\neg WtB(z, y))))$
III. $\forall x\,(Store(x) \rightarrow \neg\exists z\,((\exists y\,(Product(y) \wedge Carries(x, y)) \rightarrow WtB(z, y))))$
IV. $\forall x\,(Store(x) \rightarrow (\exists y\,(Product(y) \wedge Carries(x, y) \wedge \neg\exists z\,(WtB(z, y)))))$

Which of those expressions corresponds to our claim:

a) Just I and II.
b) Just I and III.
c) Just II and IV.
d) Three of them.
e) All of them.

Multiply Nested Quantifiers

We've already seen a lot of examples that exploit multiply nested quantifiers. But we'll do a few more here for practice.

When you use multiple quantifiers, keep the following things in mind:

- If the quantifiers are the same, it doesn't matter what order you write them in.
- If they are different, it does matter. In particular:
  - $\exists x\ (\forall y\ (P(x, y))$ means that there must be at least one x such that $P(x, y)$ holds for every y.
  - $\forall y\ (\exists x\ (P(x, y))$ means that there may be a different x for each y.

Problems

1. We want to encode the fact that any color that isn't black has a darker version. To say this precisely, we need to exploit some terminology that is standard in describing colors:

- Hue: Where the color is on the color wheel. Hues include red, blue, green, and yellow.
- Brightness (also called value): The amount of light. 100 is maximum (i.e., white). 0 is none (i.e., black).
- Saturation: The purity of the color. Think of low saturation as being greyer than high saturation.

Define one predicate:

Color(*x*): True if *x* is a color.

And define these functions, which are defined for colors:

hue(*x*): The hue of *x*.
brightness(*x*): The brightness of *x*. White will have 100 as its value. Black will have 0.
saturation(*x*): The saturation of *x*. Clear colors will have 100 as their value. Greyer colors will have lower values.

Which (one or more) of the following logical expressions encodes our claim?

I. $\forall x\ ((Color(x) \wedge (brightness(x) \neq 0)) \rightarrow \exists y\ (Color(y) \wedge (hue(y) = hue(x)) \wedge (saturation(y) = saturation(x)) \wedge (brightness(y) < brightness(x))))$

II. $\forall x\ ((Color(x) \wedge (brightness(x) = 0)) \rightarrow \neg\exists y\ \ (Color(y) \wedge (hue(y) = hue(x)) \wedge (saturation(y) = saturation(x)) \wedge (brightness(y) < brightness(x))))$

III. $\exists x\ (Color(x) \wedge \forall y\ (Color(y) \rightarrow ((hue(y) = hue(x)) \wedge (saturation(y) = saturation(x)) \wedge (brightness(y) > brightness(x)))))$

2. Suppose that we want to encode the following definition:

Anything anyone eats and isn't killed by is food.

Define:

Food(*x*): True if *x* is a food.
Eats(*x*, *y*) True if *x* eats *y*.
KilledBy(*x*, *y*): True if *x* is killed by *y*.

Consider the following statements:

I. $\forall x\ ((\exists y\ (Eats(y, x) \wedge \neg KilledBy(y, x))) \rightarrow Food(x))$
II. $\forall x\ (\exists y\ ((Eats(y, x) \wedge \neg KilledBy(y, x)) \vee Food(x)))$
III. $\forall x\ (\forall y\ ((\neg Eats(y, x) \vee KilledBy(y, x)) \vee Food(x)))$

Which (one or more) of those statements corresponds to our definition of food?

3. We want to encode a policy of our bank: Every loan without collateral has at least two signers. Define:

Loan(*x*): True if *x* is a loan.
Collat(x): True if there is collateral for *x*.
Signer(*x*, *y*): True if *x* signed *y*.

Which (one or more) of the following logical expressions encodes our claim?

I. $\neg\exists x\,(Loan(x) \wedge Collat(x) \wedge (\exists y, z\,(Signer(y, x) \wedge Signer(z, x) \wedge (y \neq z))))$
II. $\forall x\,((Loan(x) \wedge \neg Collat(x)) \rightarrow (\exists y, z\,(Signer(y, x) \wedge Signer(z, x) \wedge (y \neq z))))$
III. $\exists y, z\,(\exists x\,(Loan(x) \wedge \neg Collat(x) \wedge Signer(y, x) \wedge Signer(z, x) \wedge (y \neq z)))$

4. Define:

Loan(*x*): True if *x* is a loan.
Collat(x): True if there is collateral for *x*.
Signer(*x*, *y*): True if *x* signed *y*.

Match each logical expression:

a) $\exists x\,(\forall y\,(Loan(y) \rightarrow Signer(x, y)))$
b) $\forall x\,(\forall y\,(Signer(x, y) \rightarrow Loan(y)))$
c) $\forall x\,(Loan(x) \rightarrow \exists y\,(Signer(x, y)))$
d) $\exists x\,(\forall y\,(\neg Signer(x, y) \vee \neg Loan(y)))$
e) $\forall x\,(\exists y\,(Loan(y) \wedge Signer(x, y)))$
f) $\exists x\,(\neg\exists y\,(Loan(x) \wedge Signer(y, x)))$

To its corresponding English sentence:

i. There's at least one unsigned loan.
ii. There's someone who has signed every loan.
iii. There's no one who hasn't signed a loan.
iv. The only signed things are loans.
v. All loans have been signed.
vi. There's someone who hasn't signed any loans.

5. We want to encode the claim that no one likes anyone who doesn't tell the truth. Define:

Likes(x, *y*): True if *x* likes *y*.
TruthTeller(*x*): True if *x* is a truth teller.

Which (one or more) of the following logical expressions encodes our claim?

I. $\forall x\,(\forall y\,((\neg TruthTeller(y) \rightarrow \neg Likes(x, y)))$
II. $\forall x\,(\neg\exists y\,(\neg TruthTeller(y) \wedge Likes(x, y)))$
III. $\neg\exists x\,(\exists y\,(\neg TruthTeller(y) \wedge Likes(x, y)))$

Necessary and Sufficient Conditions

We've already talked about necessary and sufficient conditions in the context of Boolean logic. (You might want to take another look at that video now.)

https://www.youtube.com/watch?v=QtMFyTV8jfg

Recall that:

- P is a ***sufficient condition*** for Q if, whenever P is true, Q must also be true (regardless of any other conditions). Put another way:

  $P \rightarrow Q$ If P then Q.

- P is a ***necessary condition*** for Q if Q cannot be true without P also being true. Put another way:

  $Q \rightarrow P$ Q only if P.

So, if we write Q ***if and only if*** P (sometimes shortened to Q ***iff*** P), we have:

$$(P \rightarrow Q) \quad \wedge \quad (Q \rightarrow P)$$

In other words, P and Q must have the same truth value. So we have:

$$P \equiv Q$$

This is typically what we want to say when we are defining P. We'll also use it for other things, for example requirements specifications.

We've been exploiting *if and only if* all along. In some cases, we've used the mathematics convention of writing just "if" when we mean "iff". But now that we actually know how to reason with the sentences that we write, it makes sense to consider some more examples.

Let's encode the rule for deciding when leap years occur:

A year is a leap year if and only if it is divisible by 4 but not divisible by 100 unless it is also divisible by 400.

Define:

Leap(*y*): True if *y* is a leap year.
Div(*x*, *y*): True if *x* is evenly divisible by *y*.

Now we can write the leap year rule as a logical expression:

[1] $\forall y\ (Leap(y) \equiv (Div(y, 4) \wedge (\neg Div(y, 100) \vee Div(y, 400))))$

Read [1] as, "*y* is a leap year if and only if it is divisible by 4 and (it is not divisible by 100 or it is divisible by 400.)" Notice that we had to use parentheses to make the English sentence unambiguous. Also, notice that our *or* statement is, as usual, *inclusive or*. So, in principle, if *y* were not divisible by 100 but divisible by 400, it would be a leap year. Except, of course

Let's see how we can use [1] to decide whether a given year is a leap year. Assume that we can appeal to an arithmetic engine to tell us whether *Div*(x, y), where x and y are specific numbers, is true. So, for example, *Div*(100, 4) will return *T*, since 100 is divisible by 4. *Div*(100, 7) will return *F*.

Let's see whether 2000 was a leap year:

[1]	∀y (*Leap*(y)	≡	(*Div*(y, 4) ∧ (¬*Div*(y, 100) ∨ *Div*(y, 400))))	Definition	
[2]	*Leap*(2000)	≡	(*Div*(2000, 4) ∧ (¬*Div*(2000, 100) ∨ *Div*(2000, 400)))	Universal Instantiation	[1]
[3]	*Leap*(2000)	≡	(*T* ∧ (¬*T* ∨ *T*))	Arithmetic	[2]
[4]	*Leap*(2000)	≡	(*T* ∧ *T*)	Computation	[3]
[5]	*Leap*(2000)	≡	*T*	Computation	[4]

So we have that *Leap*(2000) is true.

Problems

1. Let's write a logical expression that captures what it means for two people to be first cousins. Define:

CousinOf(*x*, *y*): True if *x* is a first cousin of *y*.
ParentOf(*x*, *y*): True if *x* is a parent of *y*.
SiblingOf(*x*, *y*): True if *x* is a sibling of *y*.

Consider the following statements:

I. ∀*x*, *y*, *r*, *s* (*CousinOf*(*x*, *y*) ≡ (*ParentOf*(*r*, *x*) ∧ *ParentOf*(*s*, *y*) ∧ *SiblingOf*(*r*, *s*)))
II. ∀*x*, *y* (*CousinOf*(*x*, *y*) ≡ ∃*r*, *s* (*ParentOf*(*r*, *x*) ∧ *ParentOf*(*s*, *y*) ∧ *SiblingOf*(*r*, *s*)))
III. ∃ *x*, *y* (*CousinOf*(*x*, *y*) ≡ ∃*r*, *s* (*ParentOf*(*r*, *x*) ∧ *ParentOf*(*s*, *y*) ∧ *SiblingOf*(*r*, *s*)))

Which (one or more) of them correspond(s) to what it means to be first cousins?

2. Suppose that we want to describe UT's graduation requirements. Let's say that one can graduate from UT if and only if:

- one has at least 120 credits,
- one has completed the core requirements, and
- one has completed the requirements for a major.

(This is an oversimplification of reality, but it's close enough to being real to be an enlightening example.) Define:

CanGrad(*x*): True if *x* can graduate from UT.
Credits(*x*, *y*): True if *x* has accumulated *y* credits.
Major(*x*, *y*): True if *x* has met the requirements for a major in *y*.
Core(*x*): True if *x* has satisfied the core requirements.

(Part 1) Consider the following statements:

I. $\forall x\ (CanGrad(x) \equiv (\exists y\ (Credits(x, y) \wedge (y \geq 120) \wedge Major(x, z)) \wedge Core(x)))$
II. $\forall x\ (CanGrad(x) \equiv (\exists y\ (Credits(x, y) \wedge (y \geq 120)) \wedge \exists z\ (Major(x, z)) \wedge Core(x)))$
III. $\forall x, y\ (CanGrad(x) \equiv (Credits(x, y) \wedge (y \geq 120) \wedge \exists z\ (Major(x, z)) \wedge Core(x)))$

Which (one or more) of them correspond(s) to our graduation rule?

(Part 2) Assume that we take as a premise the correct answer to Part 1. Now we'd like to reason with it. We'll assume the following additional premises:

[1] The graduation rule, as given above
[2] *CanGrad*(*Kelly*)
[3] *Credits*(*Travis*, 130)
[4] *Major*(*Chris*, *Math*)
[5] *Core*(*Chris*)

Consider the following expressions:

I. $\exists m\ (Major(Kelly, m))$
II. $CanGrad(Travis)$
III. $Credits(Chris, 150) \rightarrow CanGrad(Chris)$

Which (one or more) of them can be proved from the premises we've got? Try doing each of the proofs and see which ones go through.

Converting Formal Claims

Formal Claims are Easier Than Everyday Claims

It's often difficult to write completely accurate logical statements about the real world. There are too many exceptions.

For example, we might be tempted to say:

$$\forall x\ (Bird(x) \rightarrow CanFly(x))$$

While this goes a long way toward describing an interesting fact of nature, it's not quite true. Penguins can't fly. Neither can birds that have just been born or ones with crude oil on their wings. The real world is messy. We'll have more to say about this soon.

But now suppose that, instead of describing the world as it happens to be, we want to specify how the world (or, more likely, some tiny, controllable part of it) must or ought to be. In other words, we are writing a set of specifications or requirements.

For example: I get to decide the rules for passwords on my website. So I might write:

$$\forall x\ (Password(x) \rightarrow (\exists y\ (Contains(x, y) \wedge Letter(y)) \ \wedge \ \exists z\ (Contains(x, z) \wedge Number(z))))$$

While this might be a bit off-putting for my customers (who'd rather be told, "Every password must contain at least one letter and one number"), it's a precise specification that I can give to the programmer who will write the code to make sure that every password actually meets the requirement.

Going even farther: in mathematics, we typically make claims that are even more abstract (and thus more general). Our claims don't talk about something as specific as passwords. They talk about, say, anything we can count. We'll see that there, the logic that we've just described is exactly what we need.

Problems

1. Suppose that I wanted to add a new constraint to my password rule. I want to say that no special characters are allowed. Assume the following predicate:

Spec(*y*): True if *y* is a special character.

My plan is to modify the specification given above so that it has this form:

$\forall x\ (Password(x) \rightarrow (\exists y\ (Contains(x, y) \wedge Letter(y)) \wedge \exists z\ (Contains(x, z) \wedge Number(z)) \wedge ****))$

Consider the following expressions:

I. $\forall y\ (Spec(y) \rightarrow \neg Contains(x, y))$
II. $\neg\exists y\ (Spec(y) \wedge Contains(x, y))$
III. $\exists y\ (Spec(y) \wedge \neg Contains(x, y))$
IV. $\forall y\ (Contains(x, y) \rightarrow \neg Spec(y))$
V. $\forall y\ (\neg Contains(x, y) \rightarrow Spec(y))$

Which (one or more) of those expressions could ***not*** be inserted in place of ****?

Mathematical Statements

Mathematical statements are formal claims. The logical language that we've developed is exactly what we need to encode them in a way that is unambiguous and that enables us to reason with what we know.

Let's look at how we can do two important things:

- Define concepts. We do this with the logical notion of equivalence.
- Assert properties and relationships of the objects that we've defined. We do this with our full arsenal of logical tools.

We derive a good deal of power from the fact that this approach lets us easily build new concepts on top of simpler ones. Let's see how this works.

Recall that we've already defined two useful predicates on the integers:

[1] $Div(x, y) \equiv \exists z\ (x = y \cdot z)$ — $Div(x, y)$ is true iff x is evenly divisible by y.
[2] $Even(x) \equiv Div(x, 2)$ — $Even(x)$ is true iff x is divisible by 2.

Now we can exploit them to define new useful concepts:

Definition of Prime Numbers

A ***prime number*** is an integer greater than 1 whose only divisors are itself and 1. We've already seen that we can write this definition formally. In fact, we gave two equivalent definitions (read them carefully to see what they are saying):

[3] $\forall x\ (Prime(x) \equiv (\ (x > 1) \wedge \forall y\ (Div(x, y) \rightarrow ((x = y) \vee (y = 1)))))$
[4] $\forall x\ (Prime(x) \equiv (\ (x > 1) \wedge \neg\exists y\ (Div(x, y) \wedge \neg(x = y) \wedge \neg(y = 1))))$

Definition of Composite Numbers

A ***composite number*** is an integer greater than 1 that isn't prime. Here's a definition stated in terms of our basic predicate *Div*:

[5] $\forall x\ (Composite(x) \equiv (\ (x > 1) \wedge \exists y\ (Div(x, y) \wedge \neg(x = y) \wedge \neg(y = 1)\ )\ )\ )$

Problems

1. We want to encode the fact that there is no largest integer. Another way to say this is that every integer has another one that is bigger than it is.

Which (one or more) of the following statements correctly encode(s) our claim:

I. $\neg\exists i\ (\forall j\ (i \geq j))$
II. $\forall i\ (\exists j\ (j > i))$
III. $\exists i\ (\forall j\ (j > i))$

2. We want to encode the fact that, for any nonzero integer, there's a different integer with the same absolute value. So, for example, consider 2. Another integer, namely -2, has the same absolute value as 2.

Let $|x|$ stand for the absolute value of x. Which (one or more) of the following statements correctly encode(s) our claim:

I. $\forall x\ (\forall y\ ((x = y) \rightarrow ((\neg(x = 0) \wedge (|x| = |y|))))$
II. $\exists y\ ((\neg(y = 0)) \rightarrow \exists x\ (|x| = |y|))$
III. $\forall x\ ((\neg(x = 0)) \rightarrow \exists y\ (\neg(x = y) \wedge (|x| = |y|)))$

Using the Definitions of Prime and Composite Numbers

Let's use our definitions of primes and composites to describe other useful classes of integers.

Definition of Mersenne Primes

An integer is a ***Mersenne number*** if and only if it is one less than some positive integer power of 2. Another way to say this is that a Mersenne number is the value of 2^n-1 for some positive integer *n*.

n	$\mathbf{2^n}$	$\mathbf{2^n - 1}$	
1	2	1	
2	4	3	*
3	8	7	*
4	16	15	
5	32	31	*
6	64	63	
7	128	127	*
8	256	255	

If a Mersenne number is also prime, then it is a ***Mersenne prime***. The Mersenne primes are marked with a * in the table above.

Let's represent this definition as a logical expression. Let the universe be the set of positive integers. Define:

MersenneP(*x*): True if *x* is a Mersenne prime.

Then we can write:

[6] $\forall x\ (MersenneP(x) \equiv (Prime(x) \wedge \exists n\ (x = 2^n\text{-}1)))$

Nifty Aside

Mathematicians have been studying Mersenne primes since the early 17th century. The French monk Marin Mersenne got his name attached to them. Mersenne claimed to have checked all values for *n* up to 257 and to have determined those for which 2^n-1 was prime. It turns out that he got several of them wrong. By 1947 (with the aid of some computing), Mersenne's range had been completely checked. The only values of *n*, in Mersenne's range, for which 2^n-1 is prime are 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107 and 127. By the way, it can be proved that if *n* is composite, then 2^n-1 is also composite. So it's only necessary to check prime values of *n*.

The Great Internet Mersenne Prime Search (GIMPS) project has been responsible for finding all the Mersenne primes that have been discovered since 1997. As of October, 2014, 48 Mersenne primes were known. It's conjectured that are infinitely many of them, but they are so big that it's not straightforward to compute with them.

Definition of Fermat Numbers

A ***Fermat number*** is an integer that can be expressed as $2^{2^n}+1$, for some positive integer *n*.

> ***Nifty Aside***
>
> The Fermat numbers are named for the 17th century French mathematician Pierre de Fermat, who first studied them. The first seven Fermat numbers are 3, 5, 17, 257, 65537, 4294967297, and 18446744073709551617, of which only the first five were actually computed by Fermat.
>
> Fermat was particularly interested in whether Fermat numbers are necessarily prime. All the ones he studied (i.e., the first five) are prime and he thought that all Fermat numbers were prime. It is now thought that the only prime ones are those first five, although there's no proof that that's so. Visit http://www.prothsearch.net/fermat.html to find out more about a distributed effort on the Internet to find factors of very large Fermat numbers.

We'll leave stating this definition as a logical expression as a problem for you to solve.

Fermat's Last Theorem

Pierre de Fermat did many different things. Here's another one (pretty much unrelated to the Fermat numbers that we just discussed).

Consider this equation:

[7] $x^n + y^n = z^n$

Suppose that $n = 1$. Then, if $x = 1$, $y = 1$ and $z = 2$, we have that $1^1 + 1^1 = 2^1$ and the equation is satisfied.

Suppose that $n = 2$. Then, if $x = 3$, $y = 4$ and $z = 5$, we have that $3^2 + 4^2 = 5^2$, and the equation is satisfied.

Fermat conjectured that, if $n > 2$, then there are no three positive integers *x*, *y*, and *z* that can satisfy the equation.

Nifty Aside

In 1637, Fermat wrote this conjecture in the margin of a book but claimed that his proof wouldn't fit in the margin. Mathematicians struggled for a couple of centuries to find a real proof. Andrew Wiles finally discovered one in 1995.

Fermat's conjecture may appear to be little more than a claim about a few numbers. But there's also a striking geometric interpretation of it.
Visit <http://www.youtube.com/watch?v=xG63O03lWZI > to try to visualize it.

Fermat's Last Theorem fascinated and puzzled mathematicians for so long that it has made its way into popular lore. Visit http://www.youtube.com/watch?v=ReOQ300AcSU for a video that gives several examples of this, including a fascinating one in which it appears that Homer Simpson (of all people) has shown that Fermat's conjecture is false. (You may only want to watch the first half or so of this one.)

We'll leave stating Fermat's famous claim as a logical expression as a problem for you to solve.

Goldbach's Conjecture

We'll next consider perhaps the most famous mathematical conjecture that remains unproven:

[8] Every even integer greater than 2 can be expressed as the sum of two primes.

For example, 18 = 5 + 13
= 7 + 11

Nifty Aside

This conjecture is named for the German mathematician Christian Goldbach, who stated a slightly different version of it in a letter to Leonhard Euler in 1742. While it remains unproven to this day, it is widely believed to be true. It is actually known to be true for all even integers up to $4 \cdot 10^{18}$.

We also know that almost all even integers greater than 2 can be expressed as the sum of two primes in multiple different ways. For example:

100 = 3 + 97 = 11 + 89 = 17 + 83 = 29 + 71 = 41 + 59 = 47 + 53

Again, we'll leave stating this famous claim as a logical expression as a problem for you to solve.

Problems

1. A Fermat number is an integer that can be expressed as $2^{2^n}+1$, for some positive integer n. Define:

FermatN(x): True if x is a Fermat number.

Which (one or more) of the following logical expressions define(s) the Fermat numbers:

I. $\exists n\ (x = 2^{2^n}+1)) \rightarrow \mathit{FermatN}(x))$
II. $\forall x\ (\mathit{FermatN}(x) \equiv \exists n\ (x = 2^{2^n}+1))$
III. $\neg\exists n\ (x = 2^{2^n}+1)) \rightarrow \neg\mathit{FermatN}(x))$

2. Fermat's Last Theorem is the conjecture, first made by Pierre de Fermat, that, if $n > 2$, then there are no three positive integers x, y, and z that can satisfy the equation:

[1] $x^n + y^n = z^n$

All we're going to do here is to state the claim formally. Since Fermat first made the claim in 1637, yet it wasn't proved until 1995, the proof is clearly beyond the scope of our effort here.

Assume a universe of positive integers. Which of the following logical expressions does *not* correspond to Fermat's claim:

a) $\forall n\ ((n > 2) \rightarrow (\forall x, y, z\ (x^n + y^n \neq z^n)))$
b) $\forall n, x, y, z\ \ ((n \leq 2) \vee (x^n + y^n \neq z^n))$
c) $\forall n\ ((n > 2) \rightarrow \neg\exists x\ (\neg\exists y\ (\neg\exists z\ (x^n + y^n = z^n))))$
d) $\forall n\ ((n > 2) \rightarrow \neg\exists x\ (\exists y\ (\exists z\ (x^n + y^n = z^n))))$

3. Goldbach's Conjecture: Every even integer greater than 2 can be expressed as the sum of two primes. Clearly we're not going to try to prove it. (No one ever has.) But let's express it formally. Assume a universe of positive integers. Which of the following logical expressions does *not* correspond to Goldbach's Conjecture:

a) $\forall x\ (\neg\mathit{Even}(x) \vee (x \leq 2) \vee \exists y, z\ (\mathit{Prime}(y) \wedge \mathit{Prime}(z) \wedge (x = y+z)))$
b) $\forall x\ ((\mathit{Even}(x) \wedge (x > 2)) \rightarrow \exists y, z\ (\mathit{Prime}(y) \wedge \mathit{Prime}(z) \wedge (x = y+z)))$
c) $\exists x\ (\neg\mathit{Even}(x) \wedge (x > 2) \wedge \exists y, z\ (\mathit{Prime}(y) \wedge \mathit{Prime}(z) \wedge (x = y+z)))$
d) $\neg\exists x\ (\mathit{Even}(x) \wedge (x > 2) \wedge \neg\exists y\ (\exists z\ (\mathit{Prime}(y) \wedge \mathit{Prime}(z) \wedge (x = y+z))))$

Rational and Irrational Numbers

Let's define two more useful classes of numbers.

Definition of Rational Numbers

A ***rational number*** is a number that can be expressed as the quotient of two integers. So, for example, the following are all rational numbers:

$$5,\ \frac{7}{11},\ \frac{3475}{58},\ \frac{-7}{985}$$

In case you're thinking that 5 doesn't appear to be a quotient, notice that we could have written it as $\frac{5}{1}$.

We'll leave writing this definition as a logical expression as a problem for you to solve.

Definition of Irrational Numbers

An ***irrational number*** is a real number that isn't rational. Two important irrational numbers are π and $\sqrt{2}$. There are, of course, infinitely many more.

> ***Big Idea***
>
> Modern mathematics could not exist without the logical tools that we have been describing.

Problems

1. Which (one or more) of the following proposed definitions for the rational numbers is correct:

I. $\exists x, y, z\ (Rational(x) \equiv (x = \frac{y}{z}))$

II. $\forall x\ (Rational(x) \equiv (\exists y, z\ (x = \frac{y}{z})))$

III. $\exists x\ (\forall y, z\ (Rational(x) \equiv (x = \frac{y}{z})))$

Business Policies and Database Constraints

We've already seen some examples of the use of logic to describe business policies and procedures (which, if I'm the boss, I get to write). Again, the win of logic is that it is completely unambiguous. No wishful thinking allowed.

For example, let's describe our policy for issuing loans. Define:

Approved(x):	True if x is an approved loan.
Super(x):	True if x is a supervising loan officer.
Signed(y, x):	True if y has signed off on loan x.

Then we can write:

$$\forall x\ ((Loan(x) \wedge Approved(x)) \rightarrow (\exists y\ (\exists z\ (Super(y) \wedge Signed(y, x) \wedge Signed(z, x) \wedge \neg(y = z)))))$$

Read this as, "Every approved loan has been signed off on by two (different) people, at least one of whom is a supervising loan officer."

Problems

1. Suppose that we are the Human Resources department of a large company. We want to provide an explicit statement of our company's personnel policies. We want to write down even things that might seem obvious to you. Define:

$Supervises(x, y)$:	True if x is the supervisor of y.
$CEO(x)$:	True if x is the CEO of the company.
$Super(x)$:	True if x holds the title of supervisor.

(Part 1) Which (one or more) of the following logical expressions best correspond(s) to the policy:

> Everyone except, of course, the CEO has a supervisor.

I. $\forall x\ (\neg CEO(x) \rightarrow \exists y\ (Supervises(y, x)))$
II. $\forall x\ ((\neg CEO(x) \rightarrow \exists y\ (Supervises(y, x))) \wedge (CEO(x) \rightarrow \neg\exists y\ (Supervises(y, x))))$
III. $\forall x\ (\forall y\ (CEO(x) \rightarrow \neg Supervises(y, x)))$

(Part 2) Which (one or more) of the following logical expressions correspond(s) to the policy:

> No one may be his/her own supervisor.

I. $\forall x\ (\forall y\ (Supervises(x, y) \rightarrow \neg(x = y)))$
II. $\forall x\ (\neg\exists y\ (Supervises(x, y) \wedge (x = y)))$
III. $\neg\exists y\ (\forall x\ (Supervises(x, y) \wedge (x = y)))$

(Part 3) Which (one ore more) of the following logical expressions correspond(s) to the policy:

> Every supervisor must have at least one employee reporting to him/her.

I. $\forall x\ (\forall y\ (Super(x) \rightarrow Supervises(x, y)))$
II. $\forall x\ (Super(x) \rightarrow \exists y\ (Supervises(x, y)))$
III. $\forall x\ (\forall y\ (Super(x) \rightarrow \neg Supervises(x, y)))$

2. We have a corporate policy that says that anyone who's been caught stealing twice doesn't work here anymore. Define:

$SEvent(e, x)$:	True if e is the unique identifier of a stealing event and x was a perpetrator of the event.
$Employee(x)$:	True if x works for the company.

Which (one or more) of the following logical expressions correspond(s) to our policy:

I. $\exists d\ (\exists e\ (\forall x\ (SEvent(d, x) \wedge SEvent(e, x) \wedge \neg(e = d))) \rightarrow \neg Employee(x))$
II. $\forall x\ (\exists d\ (\exists e\ (SEvent(d, x) \wedge SEvent(e, x))) \rightarrow \neg Employee(x))$
III. $\forall x\ (\exists d\ (\exists e\ (SEvent(d, x) \wedge SEvent(e, x) \wedge \neg(e = d))) \rightarrow \neg Employee(x))$

3. Suppose, in the interest of safety, we wish to enforce the following policy:

> Every department has at least two EMT-trained supervisors.

Define:

$Dept(x)$:	True if x is a department within the company.
$Super(x)$:	True if x holds the title of supervisor.
$HasTraining(x, y)$:	True if x has been trained in area y.
$WorksInDept(x, y)$:	True if x works in department y.

Using these predicates, write out a logical statement that corresponds to our claim. Which of these statements is correct:

I. $\exists y, z\ (Super(y) \wedge Super(z) \wedge \neg(y = z) \wedge (\forall x\ (Dept(x) \wedge WorksInDept(y, x) \wedge WorksInDept(z, x) \wedge HasTraining(y, EMT) \wedge HasTraining(z, EMT))))$

II. $\forall x\ (Dept(x) \wedge (\exists y, z\ (Super(y) \wedge Super(z) \wedge \neg(y = z) \wedge WorksInDept(x, y) \wedge WorksInDept(x, z) \wedge HasTraining(y, EMT) \wedge HasTraining(z, EMT))))$

III. $\forall x\ (Dept(x) \rightarrow (\exists y, z\ (Super(y) \wedge Super(z) \wedge \neg(y = z) \wedge WorksInDept(y, x) \wedge WorksInDept(z, x) \wedge HasTraining(y, EMT) \wedge HasTraining(z, EMT))))$

IV. $\exists y, z\ (Dept(x) \wedge Super(y) \wedge Super(z) \wedge \neg(y = z) \wedge WorksInDept(y, x) \wedge WorksInDept(z, x) \wedge HasTraining(y, EMT) \wedge HasTraining(z, EMT))$

V. $\exists y, z\ (\forall x\ (Dept(x) \wedge Super(y) \wedge Super(z) \wedge \neg(y = z) \wedge WorksInDept(y, x) \wedge WorksInDept(z, x) \wedge HasTraining(y, EMT) \wedge HasTraining(z, EMT)))$

Software Requirements Specifications

Suppose that I need someone (possibly even myself) to write a program to solve some problem that I have in mind. I need a formal way to describe what the program is supposed to do. The logical framework that we've just developed can help me.

In particular, we can write logical statements that describe three things:

- Anything that can be assumed to be true when the program begins. We'll call that the program's ***precondition*** (since "pre" means "before").

- Everything that we want to guarantee must be true when the program ends. We'll call that the program's ***postcondition*** (since "post" means "after").

- Key properties that must be maintained as the program executes. We'll call these ***invariants***. (They don't change even as other things are changing as the program makes progress toward its goal).

The Towers of Hanoi

To see how we can use those things to specify a program, let's look at a fun example that isn't exactly going to shake up the computer industry. But it's an interesting problem to write a program to solve.

We want to write a program to solve the Towers of Hanoi problem.

https://www.youtube.com/watch?v=atWdRyQKi5k

Various stories have been created to go along with the problem. One version is the following:

> In a monastery in India there are three poles and 64 golden disks, each of a different diameter. When God created the universe, he stacked the disks on the first of the poles, with the largest on the bottom. The remaining disks were stacked in order of size, with the smallest on the top. The monks were given the task of moving all 64 disks to the last pole. But the disks are sacred, so there are important rules that must be followed. Whenever a disk is removed from a pole, it must immediately be placed on some other pole. No disks may be placed on the ground or held. Further, a disk may never be placed on top of a smaller disk. The monks were told that they must begin working immediately, taking turns around the clock. When they finish, the world will end.

Nifty Aside

The Towers of Hanoi problem was invented by François Édouard Anatole Lucas, who published it in 1883 under the name of N. Claus of Siam. Can you describe a procedure to solve it?

Solving the Towers of Hanoi – The Power of Recursion

Can you describe a procedure to solve the Towers of Hanoi puzzle? How many moves does your procedure take to solve the problem of 2 disks? 3 disks? 4 disks? 8 disks? 64 disks?

https://www.youtube.com/watch?v=no_tN7kQ-O0

Here's a procedure. Assume that, when you call it, you give it an integer *n*, the number of disks on the stack. Notice that it calls itself. So, for example, to move three disks, it first moves two out of the way, then it moves the one remaining (bottom) one. Then it moves the two disks onto the bottom one.

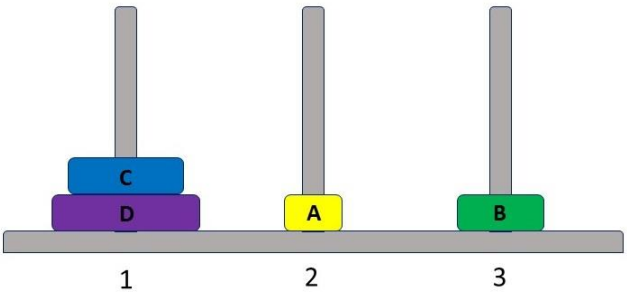
towersofHanoi(*n*: positive integer) =
1. If *n* = 1 then move the disk to the goal pole.
2. Else:
   2.1. Move the top *n*-1 disks to the pole that is neither the current one nor the goal.
   2.2. Move the bottom disk to the goal pole.
   2.3. Move the *n*-1 disks that were just set aside to the goal pole.

For 2 disks, it takes 3 moves. (Move one out of the way. Move the bottom one into place. Move the other one on top of it.) For 3 disks, it's necessary to move 2 disks twice, plus the bottom one once. So 3+3+1=7 moves. For 4 disks, it's necessary to move 3 disks twice, plus the bottom one once. So 7+7+1=15 moves. Can you see the pattern? Notice that $15 = 2^4 - 1$. For 8 disks, it's $2^8 - 1$, which is 255. For 64 disks, it's $2^{64} - 1$, which, at one move per second, would take 584,542,046,090 years, 228 days, 15 hours, 14 minutes and 45 seconds. That's more than the number of years since the Big Bang.

Problems

1. How many moves are required to move 8 disks from one pole to another?

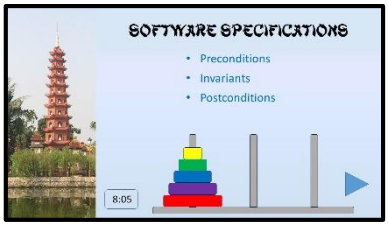
2. Suppose that we started with all four disks on pole 1. The goal is to move them all to pole 3. Here's what we've done so far. Our last move was of disk B to pole 3.



What move should we make next?

Specifications for the Tower of Hanoi Problem

How shall we write logical expressions that correspond to the specification for a program to solve the Towers of Hanoi problem?



https://www.youtube.com/watch?v=0-olSYhcELA

A useful way to do this is to write three things:

- A ***precondition***: What is guaranteed to be true when the program starts?
- A ***postcondition***: What must be true when the program finishes in order for it to have performed correctly?
- One or more ***invariants***: Claims that can be shown to be true all along as the program runs. The point of stating such invariants is to make it possible to prove that, if the precondition holds when the program begins, the postcondition will hold when it ends. So we want to capture key facts about the problem and its solution.

For this problem, in order to do this we'll need the following predicates:

Disk(*x*):	True if *x* is a disk.
Pole(*x*):	True if *x* is a pole.
On(*x*, *y*, *t*):	True if *x* is on *y* at time *t*.
Above(*x*, *y*, *t*):	True if *x* is above *y* on some pole at time *t*.
LargerThan(*x*, *y*):	True if *x* is larger than *y*.

We'll assume that all the disks start out on $Pole_1$ and should end up on $Pole_3$.

Let's first define the precondition (i.e., what's true when we start):

$\forall x\,(Disk(x) \rightarrow On(x, Pole_1, time_1))$ At time 1, every disk is on $Pole_1$.

Now let's define conditions that must always be true (except when we are in the middle of a single move). In other words, these conditions will be true when we start, they'll be true when we finish, and they must be maintained by each move as we are in the process of solving the puzzle. We call such conditions invariants because they never change during the solution process. Notice that we could include here such claims as "The Earth revolves around the Sun." It's an invariant. But it won't help us reason about a solution to this problem. Doing that requires encoding two key facts:

[1] At each step, every disk is on some pole.
[2] At each step, given any disk, any disks below it are bigger than it is.

Here we've written a single claim that is the conjunction of those two:

$\forall t\ (\forall x, y\ ((Disk(x) \wedge Disk(y) \wedge Above(x, y, t)) \rightarrow LargerThan(y, x)))$ At all times t, if disk x is above disk y, y is larger than x.

$\wedge$

$\forall t, x\ (Disk(x) \rightarrow \exists y\ (Pole(y) \wedge On(x, y, t)))$ At all times, every disk must be on some pole.

Finally, we must describe the postcondition (i.e., what must be true when we finish): We'll leave this for you to figure out as an exercise.

Problems

1. The last step, in writing our formal specification of the Towers of Hanoi problem, is to describe its postcondition. When can the program halt because its goal has been achieved? Which of the following logical expressions does that? (Assume that it is not necessary to repeat the requirements that we've already given, as an invariant, for *every* state. Just describe what must become true before the program can halt.)

a) $\exists t\ (\forall x, y\ (On(x, Pole_3, t) \rightarrow On(y, Pole_3, t)))$
b) $\forall x, y\ ((Disk(x) \wedge Pole(y)) \rightarrow On(x, y, time_3))$
c) $\exists t\ (\forall x\ (Disk(x) \rightarrow On(x, Pole_3, t)))$
d) $\exists t\ (\forall x, y\ (LargerThan(x, y) \rightarrow (On(x, Pole_3, time_3) \wedge On(y, Pole_3, time_3))))$
e) $\exists x, y\ (Pole(x) \wedge Disk(y) \wedge On(y, x, time_3))$

Specifications for a Sorting Program

Formal specifications of important programs do several things for us:

1. They force the writer to be completely clear about what the program is actually supposed to do. This means that the reader knows what the writer meant.

2. They may make it possible to match the current specification against specifications of programs that already exist. Maybe we don't have to write code from scratch.

3. They may make it possible for an automated reasoning system to check to make sure that the program does what it is supposed to do.

So, while writing these specifications isn't a trivial task, it is often worth it. Let's see how we can do it.

When we work with large amounts of data, we often want to sort the items first, before we do anything else. That will make other things much easier. For example, would you want to look someone up in a phone book if the entries were arranged randomly?

So suppose that we want someone to write a sorting program for us. Can we write a logical statement that specifies what the program has to do? Notice that we're not saying how the

How shall we capture, in a logical statement, how we want things ordered? When we're working with a complex problem, we want to focus on one thing at a time, if we can. So let's define a predicate that tells us whether one thing should come before another. We will hide the details of what determines that. So we'll define:

ComesBefore(x, *y*): True if x comes before *y* in whatever order we have agreed we want to produce.

We will need a few other predicates as well:

List(x): True if x is a list of elements.
In(x, *y*): True if x is in *y*.
Permutation(x, *y*): True if x is a permutation of *y*. That means that x and *y* contain the same elements but not necessarily (although possibly) in the same order. Again, let us assume that, as a separate effort, we can define precisely what it means to be a permutation.

Let's call the input to our desired program *inlist* and the output of our program *outlist*.

Now we're ready to write our specification:

- The precondition (i.e., what we guarantee will be true before the program starts):

  List(*inlist*)

- The postcondition: (i.e., what we require must be true when the program finishes):

List(*outlist*) $\wedge$ *Permutation*(*outlist*, *inlist*)	*outlist* is a list and it is a permutation of *inlist*.
$\wedge$	and
$\forall$x, *y* ((*In*(x, *outlist*) $\wedge$ *In*(*y*, *outlist*)	If x and *y* are in *outlist*
$\wedge$	and
(*position*(x, *outlist*) < *position*(*y*, *outlist*)))	x is in front of *y* in the list,
$\rightarrow$	then either:
((*key*(x) = *key*(*y*)) $\vee$	x and *y* have the same key, or
ComesBefore(*key*(x), *key*(*y*))))	x comes before *y* in the sort order.

Problems

1. Let's restate the postcondition that we just gave, this time with line numbers so that we can refer to them:

[1]	$List(outlist) \wedge Permutation(outlist, inlist)$	*outlist* is a list and it is a permutation of *inlist*.
	$\wedge$	and
[2]	$\forall x, y((In(x, outlist) \wedge In(y, outlist)$	If *x* and *y* are in *outlist*
	$\wedge$	and
[3]	$(position(x, outlist) < position(y, outlist)))$	*x* is in front of *y* in the list,
	$\rightarrow$	then:
[4]	$((key(x) = key(y)) \vee$	*x* and *y* have the same key, or
[5]	$ComesBefore(key(x), key(y))))$	*x* comes before *y* in the sort order.

Suppose that someone has written a program and has claimed that it satisfies our specification. We run the program:

With this input: (7, 5, 2, 3, 6)

And we get this output: (2, 7, 5, 6, 3)

Assume that the numbers shown above are the values of *key* for their respective list elements.

Which of the following claims is true:

a) At least on the basis of this one test case, it appears that the program is correct.
b) The program is not correct. The test case does not satisfy [1].
c) The program is not correct. There is exactly one (*x*, *y*) pair that satisfies [2] and [3] but does not satisfy either [4] or [5].
d) The program is not correct. There are at least two (*x*, *y*) pairs that satisfy [2] and [3] but do not satisfy either [4] or [5].

2. Let's restate the postcondition that we just gave, this time with line numbers so that we can refer to them:

[1]	*List*(*outlist*) ∧ *Permutation*(*outlist*, *inlist*)	*outlist* is a list and it is a permutation of *inlist*.
	∧	and
[2]	∀*x*, *y*((*In*(*x*, *outlist*) ∧ *In*(*y*, *outlist*)	If *x* and *y* are in *outlist*
	∧	and
[3]	(*position*(*x*, *outlist*) < *position*(*y*, *outlist*)))	*x* is in front of *y* in the list,
	→	then:
[4]	((*key*(*x*) = *key*(*y*)) ∨	*x* and *y* have the same key, or
[5]	*ComesBefore*(*key*(*x*), *key*(*y*))))	*x* comes before *y* in the sort order.

Suppose that someone has written a program and has claimed that it satisfies our specification. We run the program:

With this input: (7, 7, 2, 3, 6)

And we get this output: (2, 3, 5, 6, 7)

Assume that the numbers shown above are the values of *key* for their respective list elements.

Which of the following claims is true:

a) At least on the basis of this one test case, it appears that the program is correct.
b) The program is not correct. The test case does not satisfy [1].
c) The program is not correct. There is exactly one (*x*, *y*) pair that satisfies [2] and [3] but does not satisfy either [4] or [5].
d) The program is not correct. There are at least two (*x*, *y*) pairs that satisfy [2] and [3] but do not satisfy either [4] or [5].

Specifications for a Business Application

So now we've talked about:

(1) Writing logical expressions that describe constraints on values in application databases.
(2) Writing logical expressions that describe the functions that programs should perform.

Let's combine these two things.

Up until now, when we did (1), we mostly assumed that people were doing the work. The logical constraints just checked them. So, for example, people assigned employees to projects and to supervisors. The constraints made sure that they did so in an acceptable way.

But now suppose that we want to write a program to do a similar sort of task. Now we can write the same kinds of constraints but we'll treat them as program specifications. The job of the program is to make the constraints true.

Suppose that we want to schedule classes into classrooms. In a large university, this is a huge task, generally made worse by the fact that there's little latitude: there may be only barely enough classrooms to go around and then only if some classes happen at undesirable times like 8:00 in the morning. The program that does this may have to try a lot of different assignments before it finds one that works. We're not going to try to describe how it should do that. All we're going to do is to describe what an acceptable solution is. Once the program finds such a solution, it can stop.

Define:

Class(x):	True if x is a class.
Room(*r*):	True if *r* is a room
Timeslot(*t*):	True if *t* is a time slot
Scheduled(*c*, *r*, *t*):	True if *c* is scheduled in room *r* in time slot *t*.
size(*c*):	Returns the number of students in class *c*.
capacity(*r*):	Returns the capacity of room *r*.

Here's a first attempt at a specification for a scheduling program. We'll require that every class be scheduled (at some time and in some large enough room). We can say that with this expression:

[1] $\forall c\ (\exists r, t\ (Room(r) \wedge Timeslot(t) \wedge Scheduled(c, r, t) \wedge (size(c) \leq capacity(r))))$

This appears to be a good start. But closer inspection reveals several problems. Can you spot some of them? We'll work on fixing them as an exercise.

Problems

1. Recall that we are trying to write a specification for a class scheduling program. We've defined:

Class(x):	True if *x* is a class.
Room(*r*):	True if *r* is a room
Timeslot(*t*):	True if *t* is a time slot
Scheduled(*c*, *r*, *t*):	True if *c* is scheduled in room *r* in time slot *t*.
size(*c*):	Returns the number of students in class *c*.
capacity(*r*):	Returns the capacity of room *r*.

Our first attempt was:

[1] $\forall c\ (\exists r, t\ (Room(r) \wedge Timeslot(t) \wedge Scheduled(c, r, t) \wedge (size(c) \leq capacity(r))))$

But we observe that we can satisfy this constraint by scheduling all classes at the same time in the same room. Oops. We need to add a new requirement that says that a given room, at a given time slot, may be assigned to no more than one class.

Consider the following statements. Read each of these as saying that we require [1] above *and* a new conjunct. Which (one or more) of them correctly captures our requirement?

I. $[1] \wedge \forall r\ (Room(r) \rightarrow \exists t\ (\neg\exists d\ (\exists e\ ((d \neq e) \wedge Scheduled(d, r, t) \wedge Scheduled(e, r, t)))))$
II. $[1] \wedge \forall r\ (Room(r) \rightarrow \neg\exists t\ (\exists d\ (\exists e\ ((d \neq e) \wedge Scheduled(d, r, t) \wedge Scheduled(e, r, t)))))$
III. $[1] \wedge \forall t\ (\forall r, d, e\ ((Room(r) \wedge Scheduled(d, r, t) \wedge Scheduled(e, r, t)) \rightarrow (d = e)))$

2. Recall that we are trying to write a specification for a class scheduling program. We've defined:

Class(x):	True if *x* is a class.
Room(*r*):	True if *r* is a room
Timeslot(*t*):	True if *t* is a time slot
Scheduled(*c*, *r*, *t*):	True if *c* is scheduled in room *r* in time slot *t*.
size(*c*):	Returns the number of students in class *c*.
capacity(*r*):	Returns the capacity of room *r*.

After our second attempt, we had:

[1] $\forall c\ (\exists r, t\ (Room(r) \wedge Timeslot(t) \wedge Scheduled(c, r, t) \wedge (size(c) \leq capacity(r))))$

$\wedge$

[2] $\forall r\ (Room(r) \rightarrow \neg\exists t\ (\exists d\ (\exists e\ ((d \neq e) \wedge Scheduled(d, r, t) \wedge Scheduled(e, r, t)))))$

Now every class must be scheduled [1] and no two classes may be scheduled in the same room at the same time [2].

But we've still got a problem. Nothing prevents the scheduler from scheduling a class at more than one time or in more than one room. It probably wouldn't want to, but strange things happen in complicated programs and we certainly don't want to allow this. So we need to add another constraint.

Consider the following statements. Read each of these as saying that we require [1] and [2] above *and* a new conjunct. Which (one or more) of them correctly captures our requirement?

I. $[1] \wedge [2] \wedge \forall c, r, s, t, w\ ((Scheduled(c, r, t) \wedge Scheduled(c, s, w)) \rightarrow ((r = s) \wedge (t = w))$
II. $[1] \wedge [2] \wedge \neg\exists r\ (\exists c, s, t, w\ (Scheduled(c, r, t) \wedge Scheduled(c, s, w)) \wedge ((r \neq s) \vee (t \neq w))$
III. $[1] \wedge [2] \wedge \neg\exists t\ (\exists c, r, s, w\ (Scheduled(c, r, t) \wedge Scheduled(c, s, w)) \wedge \neg((r = s) \wedge (t = w))$

Converting Everyday Claims

Choosing Appropriate Predicates

When we're writing mathematics, it's usually clear what predicates we should define. When we're expressing formal claims about databases, we choose predicates that correspond to the fields and values in the database we're working with. When we're stating program specifications, we generally know what values of what objects are important.

But the story is very different when we want to encode facts about our complex world. Very quickly it becomes obvious that we won't be able to capture every aspect of everything. We will have to simplify. The key is that we have to simplify in a way that lets us accomplish all the reasoning that we care about.

Thus, one of the hardest things to do, when we try to encode facts about our complex world, is to choose appropriate predicates.

In our *Bad Movie* example, we used the predicate:

InClass(x): True if x is in our class.

That worked fine when all we cared about was reasoning about our classmates and their movie preferences.

But, suppose we'd wanted to reason more generally about the fact that various classes have different tastes and tendencies. Then, almost surely, we'd have needed something more like:

InClass(x, *y*): True if person x is in class *y*.

For example, now we could say that someone can't be in more than one class:

[1] $\forall x, y, z\ ((InClass(x, y) \wedge InClass(x, z)) \rightarrow (y = z))$

We couldn't say that before.

We've already seen some other examples of this problem:

Should we have:

HasLostWallet(x) or *HasLost*(x, *y*)

Should we have:

HasTenockritus(x) or *HasDisease*(x, *Tenockritus*)

A general rule of thumb is that if your predicate names get very long, you're probably trying to roll too much into a single predicate. If you want to be able to generalize what you know, even a little bit, it is probably a good idea to break the complex predicate apart into meaningful units.

But very short predicate names can also be symptoms of a problem.

Should we use the very simple predicate *Has* in all of these example:

[1]	John has tenockritus.	$Has(John, Tenockritus)$
[2]	All bears have tails.	$\forall x\ (Bear(x) \rightarrow Has(x, Tail))$
[3]	Terry has a red car.	$\exists x\ (Has(Terry, x) \wedge Car(x) \wedge ColorOf\ (x, Red))$
[4]	Kerry has a crush on Chris.	$\exists x\ (Has(Kerry, x) \wedge Crush(x) \wedge CrushObject(x, Chris))$

It's hard to find anything meaningful (about which we might want to prove something) in common across these four examples. [1] describes a disease state. [2] describes a part/whole relationship. [3] describes ownership. And [4] corresponds to an idiom that happens to include the word, "has".

On the other hand, if we have a problem and we want to reason about it, we will never get off the ground if we try to imagine every possible generalization and every possible related problem that we might, someday, decide to pursue. So it's necessary to stop someplace.

Big Idea

There's no perfect set of predicates. Just ones that get the job done.

Problems

1. Suppose that we have chosen some predicates that allow us to write, among other things:

[1] $\forall x\ (SpeakEnglish(x) \rightarrow CanCope(x, Europe))$

Now we want to say some more things. Consider each of the following claims:

I. Every employee speaks at least two languages.
II. English speakers have a big advantage in international business.
III. Learning one European language isn't hard if you already speak another one.
IV. Kerry speaks English and Chinese.

For which (one or more) of them can we make use of the *SpeakEnglish* predicate that we already have?

2. Back in our discussion of Boolean logic, we were forced to let single Boolean variables stand for arbitrarily complex ideas. We didn't have the tools of predicate logic to work with. But now we do.

Recall that we attempted to represent, "We'll have chips but no peanuts at the party." We defined:

C : We'll have chips at the party.
P: We'll have peanuts at the party.

Then we wrote:

[1] $C \wedge \neg P$

Now let's do this one in predicate logic. We could start by defining:

HaveAtParty(*x*): True if there will be *x* at the party.

Then we could write:

[2] $HaveAtParty(Chips) \wedge \neg HaveAtParty(Peanuts)$

But now suppose that we want to ask whether there will be food at the party. Consider the following additional premises that we might add to [2]:

I. $Food(Chips)$
II. $\forall x\ (Party(x) \rightarrow HasFood(x))$
III. $HaveAtParty(Chips) \rightarrow HaveAtParty(Food)$
IV. $HaveAtParty(Burgers) \rightarrow HaveAtParty(Food)$

Which of them would be sufficient to allow us to conclude that there would be food at the party? (Be as flexible as you like with how you would represent the fact that there would be food at the party. But don't add any additional premises.)