

Discrete Mathematics and Functional Programming

Thomas VanDrunen

June 23, 2006

Brief contents

I	Set	1
II	Logic	33
III	Proof	71
IV	Algorithm	91
V	Relation	127
VI	Function	155
VII	Program	181
VIII	Graph	215

Contents

Preface	ix
I Set	1
1 Sets and elements	3
1.1 Your mathematical biography	3
1.2 Reasoning about items collectively	6
1.3 Intuition about sets	6
1.4 Set notation	7
2 Expressions and Types	11
2.1 Expressions	11
2.2 Types	12
2.3 Variables	14
2.4 Making your own types	15
3 Set Operations	19
3.1 Axiomatic foundations	19
3.2 Operations and visualization	19
3.3 Powersets, cartesian products, and partitions	22
4 Tuples and Lists	25
4.1 Tuples	25
4.2 Lists	27
4.3 Lists vs. tuples vs. arrays	30
II Logic	33
5 Logical Propositions and Forms	35
5.1 Forms	35
5.2 Symbols	36
5.3 Boolean values	37
5.4 Truth tables	38
5.5 Logical equivalence	39
6 Conditionals	43
6.1 Conditional propositions	43
6.2 Negation of a conditional	44
6.3 Converse, inverse, and contrapositive	45
6.4 Writing conditionals in English	46
6.5 Conditional expressions in ML	46

7	Argument forms	49
7.1	Arguments	49
7.2	Common syllogisms	50
7.3	Using argument forms for deduction	51
8	Predicates and quantifiers	55
8.1	Predication	55
8.2	Universal quantification	56
8.3	Existential quantification	57
8.4	Implicit quantification	58
8.5	Negation of quantified propositions	59
9	Multiple quantification; representing predicates	61
9.1	Multiple quantification	61
9.2	Ambiguous quantification	63
9.3	Predicates in ML	64
9.4	Pattern-matching	65
III	Proof	71
10	Subset proofs	73
10.1	Introductory remarks	73
10.2	Forms for proofs	73
10.3	An example	75
10.4	Closing remarks	76
11	Set equality and empty proofs	79
11.1	Set equality	79
11.2	Set emptiness	80
11.3	Remarks on proof by contradiction	80
12	Conditional proofs	83
12.1	Worlds of make believe	83
12.2	Integers	84
12.3	Biconditionals	85
12.4	Warnings	85
13	<i>Special Topic:</i> Russell's paradox	89
IV	Algorithm	91
14	Algorithms	93
14.1	Problem-solving steps	93
14.2	Repetition and change	94
14.3	Packaging and parameterization	96
14.4	Example	98
15	Induction	101
15.1	Calculating a powerset	101
15.2	Proof of powerset size	102
15.3	Mathematical induction	104
15.4	Induction gone awry	105
15.5	Example	105

16 Correctness of algorithms	109
16.1 Defining correctness	109
16.2 Loop invariants	110
16.3 Big example	111
16.4 Small example	112
17 From Theorems to Algorithms	115
17.1 The Division Algorithm	115
17.2 The Euclidean Algorithm	116
17.3 The Euclidean Algorithm, another way	118
18 Recursive algorithms	121
18.1 Analysis	121
18.2 Synthesis	123
V Relation	127
19 Relations	129
19.1 Definition	129
19.2 Representation	130
19.3 Manipulation	132
20 Properties of relations	135
20.1 Definitions	135
20.2 Proofs	136
20.3 Equivalence relations	136
20.4 Computing transitivity	138
21 Closures	141
21.1 Transitive failure	141
21.2 Transitive and other closures	143
21.3 Computing the transitive closure	143
21.4 Relations as predicates	145
22 Partial orders	149
22.1 Definition	149
22.2 Comparability	150
22.3 Topological sort	151
VI Function	155
23 Functions	157
23.1 Intuition	157
23.2 Definition	158
23.3 Examples	159
23.4 Representation	160
24 Images	163
24.1 Definition	163
24.2 Examples	164
24.3 Map	165

25 Function properties	167
25.1 Definitions	167
25.2 Cardinality	168
25.3 Inverse functions	169
26 Function composition	171
26.1 Definition	171
26.2 Functions as components	172
26.3 Proofs	173
27 <i>Special Topic: Countability</i>	177
 VII Program	 181
28 Recursion Revisited	183
28.1 Scope	183
28.2 Recurrence relations	186
29 Recursive Types	189
29.1 Datatype constructors	189
29.2 Peano numbers	191
29.3 Parameterized datatype constructors	193
30 Fixed-point iteration	197
30.1 Currying	197
30.2 Problem	198
30.3 Analysis	200
30.4 Synthesis	202
31 Combinatorics	205
31.1 Counting	205
31.2 Permutations and combinations	206
31.3 Computing combinations	207
32 <i>Special Topic: Computability</i>	209
33 <i>Special topic: Comparison with object-oriented programming</i>	211
 VIII Graph	 215
34 Graphs	217
34.1 Introduction	217
34.2 Definitions	218
34.3 Proofs	219
34.4 Game theory	220
35 Paths and cycles	223
35.1 Walks and paths	223
35.2 Circuits and cycles	224
35.3 Euler circuits and Hamiltonian cycles	225

36 Isomorphisms	229
36.1 Definition	229
36.2 Isomorphic invariants	230
36.3 The isomorphic relation	231
36.4 Final bow	231

Preface

If you have discussed your schedule this semester with anyone, you have probably been asked what discrete mathematics is—or perhaps someone has asked would could make math *indiscreet*. While discrete mathematics is something few people outside of mathematical fields have heard of, it is comprised of topics that are fundamental to mathematics; to gather these topics together into one course is a more recent phenomenon in mathematics curriculum. Because these topics are sometimes treated separately or in various other places in an undergraduate course of study in mathematic, discrete math texts and courses appear like hodge-podges, and unifying themes are sometimes hard to identify. Here we will attempt to shed some light on the matter.

Discrete mathematics topics include symbolic logic and proofs, including proof by induction; number theory; set theory; functions and relations on sets; graph theory; algorithms, their analysis, and their correctness; matrices; sequences and recurrence relations; counting and combinatorics; discrete probability; and languages and automata. All of these would be appropriate in other courses or in their own course. Why teach them together? For one thing, students in a field like computer science need a basic knowledge of many of these topics but do not have time to take full courses in all of them; one course that meanders through these topics is thus a practical compromise. (And no one-semester course could possibly touch all of them; we will be completely skipping matrices, probability, languages, and automata, and number theory, sequences, recurrence relations, counting, and combinatorics will receive only passing attention.)

However, all these topics do have something in common which distinguishes them from much of the rest of mathematics. Subjects like calculus, analysis, and differential equations, anything that deals with the real or complex numbers, can be put under the heading of *continuous mathematics*, where a continuum of values is always in view. In contrast to this, *discrete mathematics* always has separable, indivisible, quantized (that is, *discrete*) objects in view—things like sets, integers, truth values, or vertices in a graph. Thus discrete math stands towards continuous math in the same way that digital devices stand toward analog. Imagine the difference between an electric stove and a gas stove. A gas stove has a nob which in theory can be set to an infinite number of positions between high and low, but the discrete states of an electric stove are a finite, numbered set.

This particular course, however, does something more. Here we also intertwine functional programming with the discrete math topics. Functional programming is a different style or paradigm from the procedural, imperative, and/or object-oriented approach that those of you who have programmed before have seen (which, incidentally, should place students who have programming experience and those who have not on a level playing field). Instead of viewing a computer program as a collection of commands given to a computer, we see a program as a collection of interacting functions, in the mathematical sense. Since functions are a major topic of discrete math anyway, the interplay is natural. As we shall see, functional programming is a useful forum for illustrating the other discrete math topics as well.

But like any course, especially at a liberal arts college, our main goal is to make you think better. You should leave this course with a sharper understanding of categorical reasoning, the ability to analyze logic formally, an appreciation for the precision of mathematical proofs, and the clarity of thought necessary to arrange tasks into an algorithm. The slogan of this course is, “Math majors should learn to write programs and computer science majors should learn to write proofs *together*.” Math majors will spend most of their time as undergraduates proving things, and computer science majors will do a lot of programming; yet they both need to do a little of the other. The fact is,

robust programs and correct proofs have a lot to do with each other, not the least of which is that they both require clear, logical thinking. We will see how proof techniques will allow us to check that an algorithm is correct, and that proofs can prompt algorithms. Moreover, the programming component motivates the proof-based discrete math for the computer science majors and keeps it relevant; the proof component should lighten the unfamiliarity that math majors often experience in a programming course.

There are three major theme pairs that run throughout this course. The theme of **proof and program** has already been explained. The next is **symbol and representation**. So much of precise mathematics relies on accurate and informative notation, and it is important to distinguish the difference between a symbol and the idea it represents. This is also a point where math and computer science streams of thought swirl; much of our programming discussions will focus on the best ways to represent mathematical concepts and structure on a computer. Finally, the theme of **analysis and synthesis** will recur. Analysis is the taking of something apart; synthesis is putting something together. This pattern occurs frequently in proofs. Take any proposition in the form “if q then p .” q will involve some definition that will need to be analyzed straight away to determine what is really being asserted. The proof will end by assembling the components according to the definitions of the terms used in p . Likewise in functional programming, we will practice decomposing a problem into its parts and synthesizing smaller solutions into a complete solution.

This course covers a lot of material, and much of it is challenging. However, with careful practice, none of it is beyond the grasp of anyone with the mathematical maturity that is usually achieved around the time a student takes calculus.

Part I

Set

Chapter 1

Sets and elements

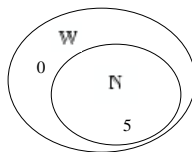
1.1 Your mathematical biography

Much of your mathematical education can be retraced by considering your expanding awareness of different kinds of numbers. In fact, human civilization's understanding of mathematics progressed in much the same way.

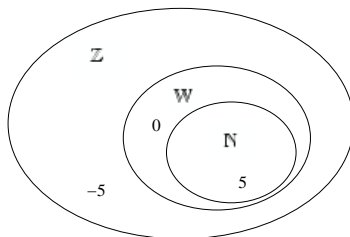
When you first conceptualized differences in quantities (10 Cheerios is more than 3 Cheerios) and learned to count, the numbers you used were 1, 2, 3, ... up to the highest number to which you could count. We call these numbers the *natural* numbers, and we will represent all of them with the symbol \mathbb{N} . It is critical to note that we are using this to symbolize *all* of them, not *each* of them. \mathbb{N} is not a variable that can stand for *any* natural number, but a constant that stands for *all* natural numbers as a group.

In the early grades, you learned to count ever higher (learned of more natural numbers) and learned operations you could perform on them (addition, subtraction, etc.). You also learned of an extra number, 0 (an important step in the history of mathematics). We will designate all natural numbers and 0 to be *whole* numbers, and represent them collectively by \mathbb{W} .

0 is a whole number but not a natural number. 5 is both. In fact, anything that is a natural number is also a whole number, but only some things (specifically, everything but zero) that are whole numbers are also natural numbers. Thus \mathbb{W} is a broader category than \mathbb{N} . We can visualize this in the following diagram.

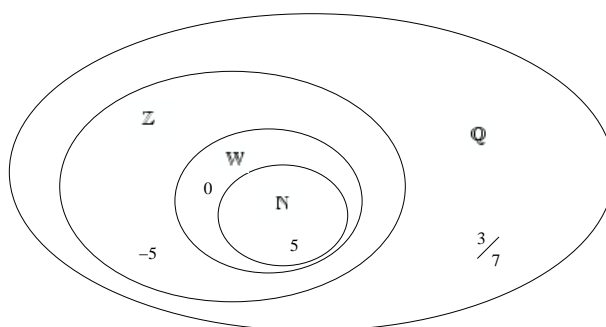


The whole number operation of subtraction eventually forced you to face a dilemma: what happens if you subtract a larger number from a smaller number? Since \mathbb{W} is insufficient to answer this, negative numbers were invented. We call all whole numbers with their *opposites* (that is, their negative counterparts) *integers*, and we use \mathbb{Z} (from *Zahlen*, the German word for “numbers”) to symbolize the integers.

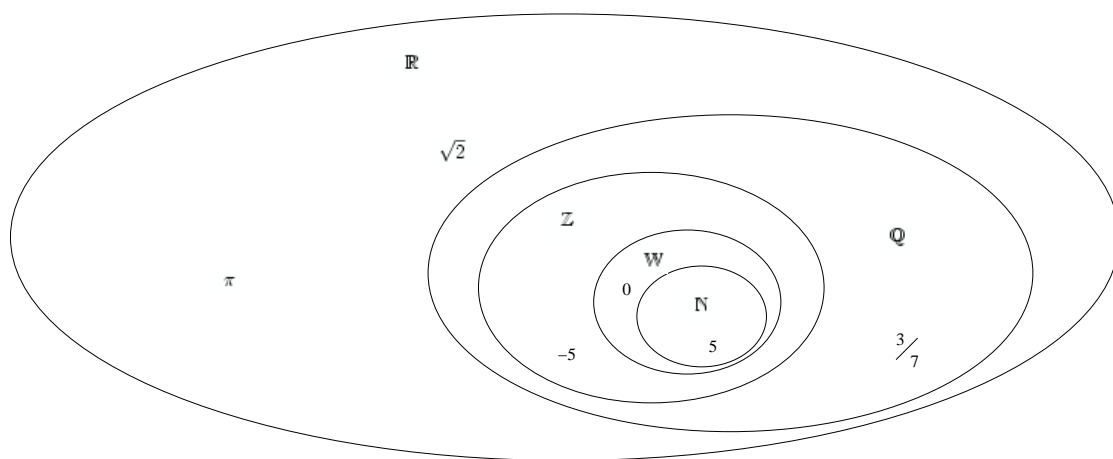


Division, as an operation on integers, results in a similar problem. What happens if we divide 5 by 2? In the history of mathematical thinking, we can imagine two cavemen arguing over how they

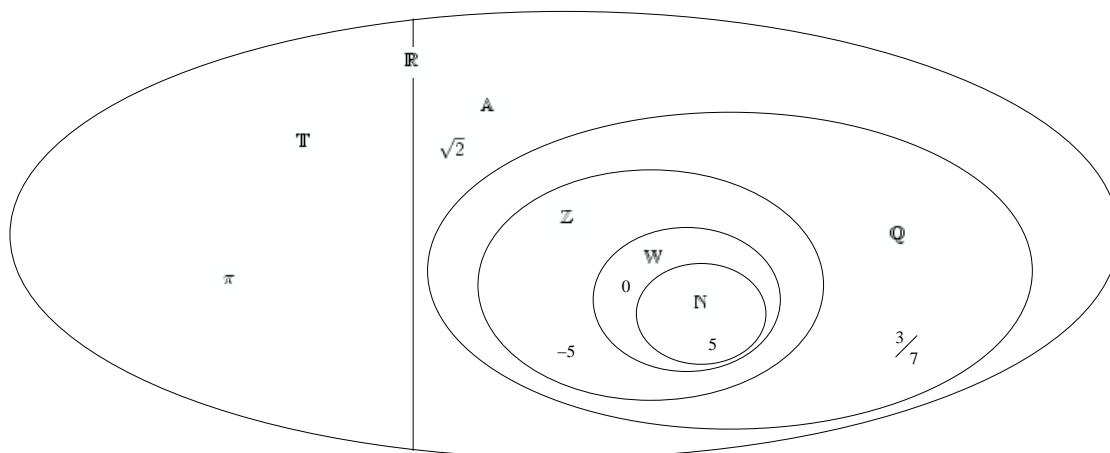
can split five apples. Physically, they could chop one of the apples into two equal parts and each get one part, but how can you describe the resulting quantity that each caveman would get? Human languages handle this with words like “half”; mathematics handles this with fractions, like $\frac{5}{2}$ or the equivalent $2\frac{1}{2}$, which is shorthand for $2 + \frac{1}{2}$. We call numbers that can be written as fractions (that is, ratios of integers) *rational numbers*, symbolized by \mathbb{Q} (for *quotient*). Since a number like 5 can be written as $\frac{5}{1}$, all integers are rational numbers.



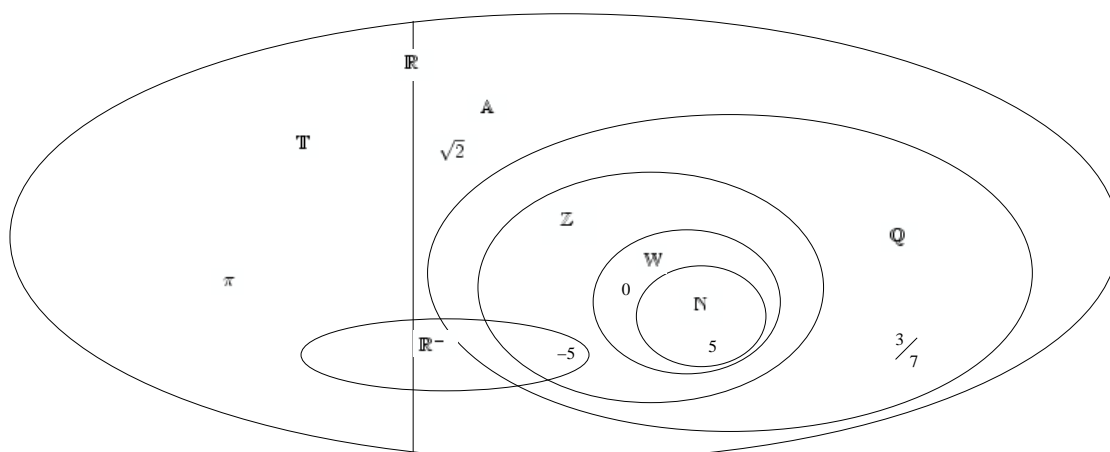
Geometry exposed the insufficiency of rational numbers. For example, an isosceles right triangle with sides of length 1 inch has a hypotenuse of length $\sqrt{2}$ by the Pythagorean theorem. A circle with diameter 1 inch has a circumference of π . Obviously such a triangle and circle can be drawn on paper, so $\sqrt{2}$ and π are measurable quantities possible in the physical world. However, they cannot be written as fractions, and hence they are not rational numbers. We call all of these “possible real world quantities” *real numbers*, and symbolize them by \mathbb{R} .



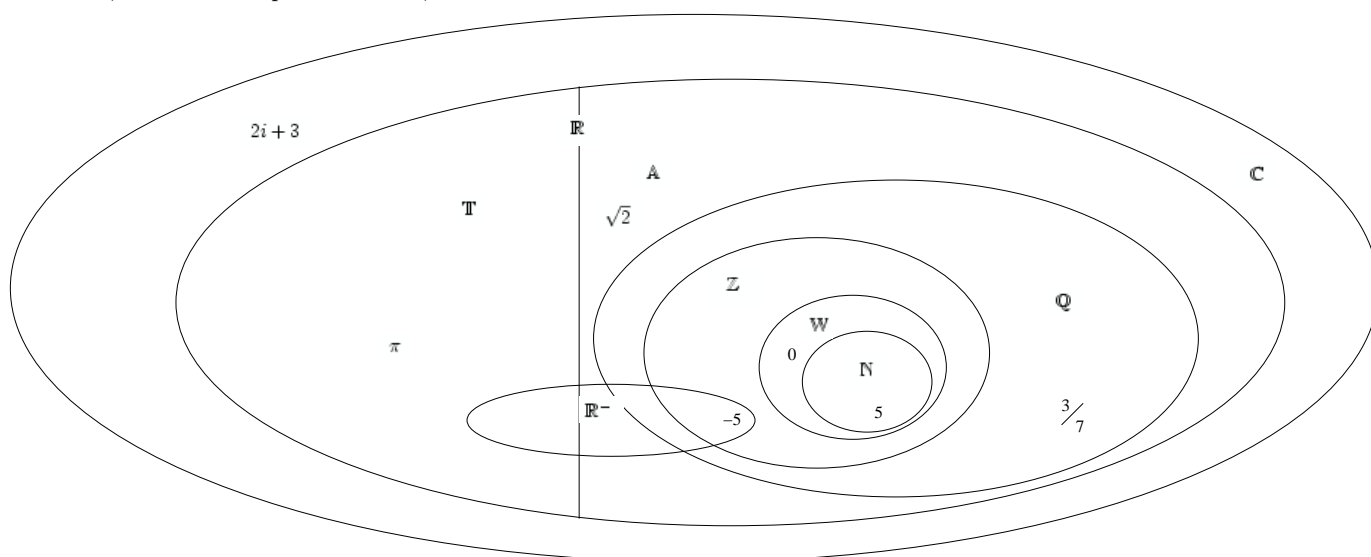
All of these designations ought to be second nature to you. A lesser known distinction that you may or may not remember is that real numbers can be split up into two camps: *algebraic numbers* (\mathbb{A}), each of which is a root to some polynomial function, like $\sqrt{2}$ and all the integers; and *transcendental numbers* (\mathbb{T}), which are not.



We first considered negative numbers when we invented the integers. However, as we expanded to rationals and reals, we introduced both new negative numbers and new positive numbers. Thus negative (real) numbers considered as a collection (\mathbb{R}^-) cut across all of these other collections, except \mathbb{W} and \mathbb{N} .



To finish off the picture, remember how \mathbb{N} , \mathbb{Z} , and \mathbb{Q} each in turn proved to be inadequate because of operations we wished to perform on them. Likewise \mathbb{R} is inadequate for operations like $\sqrt{-1}$. To handle that, we have *complex numbers*, \mathbb{C} .



1.2 Reasoning about items collectively

What is the meaning of this circle diagram and these symbols to represent various collections of numbers? Like all concepts and their representations, these are tools for reasoning and communicating ideas. The following table lists various statements that express ideas using the terminology introduced in the preceding exercise. Meanwhile, in the right column we write these statements symbolically, using symbols that will be formally introduced later.

5 is a natural number; <i>or</i> the collection of natural numbers contains 5.	$5 \in \mathbb{N}$
Adding 0 to the collection of natural numbers makes the collection of whole numbers.	$\mathbb{W} = \{0\} \cup \mathbb{N}$
Merging the algebraic numbers and the transcendental numbers makes the real numbers.	$\mathbb{R} = \mathbb{A} \cup \mathbb{T}$
Transcendental numbers are those real numbers which are not algebraic numbers.	$\mathbb{T} = \mathbb{R} - \mathbb{A}$
Nothing is both transcendental and algebraic, <i>or</i> the collection of things both transcendental and algebraic is empty.	$\mathbb{T} \cap \mathbb{A} = \emptyset$
Negative integers are both negative and integers.	$\mathbb{Z}^- = \mathbb{R}^- \cap \mathbb{Z}$
All integers are rational numbers.	$\mathbb{Z} \subseteq \mathbb{Q}$
Since all rational numbers are algebraic numbers and all algebraic numbers are real numbers, it follows that all rational numbers are real numbers.	$\mathbb{Q} \subseteq \mathbb{A}$ $\mathbb{A} \subseteq \mathbb{R}$ $\therefore \mathbb{Q} \subseteq \mathbb{R}$

element

set

We also note that in the circle diagram and the accompanying discussion, we are dealing with two very different sorts of things: on one hand we have 5, $\frac{3}{7}$, $\sqrt{2}$, π , $2i + 3$, and the like; on the other hand we have things like \mathbb{N} , \mathbb{Z} , \mathbb{Q} , \mathbb{R} , and \mathbb{C} . What is the difference? We have been referring to the former by terms such as “number” or “item”, but the standard mathematical term is *element*. We have called any of the latter category a “collection” or “group”; in mathematics, we call such a thing a *set*. Informally, a set is a group or collection of items categorized together because of perceived common properties.

This presents one of the fundamental shifts in thinking from continuous mathematics, such as pre-calculus and calculus. Up till now, you have concerned yourself with the *contents* of these sets; in discrete mathematics, we will be reasoning about *sets themselves*.

1.3 Intuition about sets

There is nothing binding about the sets we mentioned earlier. We can declare sets arbitrarily—such as the set of even whole numbers, or simply the set containing only 1, 15, and 23. We can have sets of things other than numbers. For example, other mathematical objects can be considered collectively—a set of geometric points, a set of matrices, or a set of functions. But we are not limited to mathematical objects—we can declare sets of physical objects or even abstract ideas. Grammatically, anything that is a noun can be discussed in terms of sets. We may speak of

The set of students in this course.

The set of car models produced by Ford (different from the set of *cars* produced by Ford).

The set of entrees served at Bon Appetit.

The set of the Fruits of the Spirit.

Since *set* is a noun, we can even have a set of sets; for example, the set of number sets included in \mathbb{R} , which would contain $\mathbb{Q}, \mathbb{Z}, \mathbb{W}$, and \mathbb{N} . In theory, a set can even contain itself—the set of things mentioned on this page is itself mentioned on this page and thus includes itself—though that leads to some paradoxes.

Hrbacek and Jech give an important clarification to our intuition of what a set is:

Sets are not objects of the real world, like tables or stars; they are created by our mind, not by our hands. A heap of potatoes is not a set of potatoes, the set of all molecules in a drop of water is not the same object as that drop of water[9].

It is legitimate, though, to speak of the set of molecules in the drop and of the set of potatoes in the heap.

1.4 Set notation

Finally, we explain some of the notation we used earlier:

We can describe a set explicitly by listing the elements of the set inside curly braces. Remembering that order does not matter in a set, then for example, to define the set of the colors red and green,

{}

$$X = \{\text{Red}, \text{Green}\} = \{\text{Green}, \text{Red}\}$$

...but do not forget that

$$\text{Red} \neq \{\text{Red}\}$$

Using this system, {} stands for a set with no elements, that is, the empty set, but we also have a special symbol for that, \emptyset . The symbol \in stands for set membership and should be read “an element of” or “is an element of”, depending on the grammatical context (sometimes just “in” works if you are reading quickly).

\emptyset

\in

$$\text{Red} \in \{\text{Green}, \text{Red}\}$$

The curly braces can be used more flexibly if you want to specify the elements of a set by property rather than listing them explicitly. Begin an expression like this by giving a variable to stand for an arbitrary element of the set being defined, a vertical bar (read as “such that”), a statement that the element is already a member in another set, and finally a statement that some other property is true for these elements. For example, one way to define the set of natural numbers is

$$\mathbb{N} = \{x | x \in \mathbb{Z}, x > 0\}$$

which reads “the set of natural numbers is the set of all x such that that x is an integer and x is greater than 0.” Recall from analysis that you can specify a range on the real number line, say all from one exclusive to 5 inclusive, by the notation $(1, 5]$. Indeed, a range is a set; note

$$(1, 5] = \{x | x \in \mathbb{R}, 1 < x \leq 5\}$$

If one set is completely contained in another, as $\{\text{Green}, \text{Red}\}$ is completely contained in $\{\text{Red}, \text{Green}, \text{Blue}\}$, we say that the first is a *subset* of the second. The symbol \subseteq reads “a subset of” or “is a subset of”, as in

subset, \subseteq

$$\{\text{Green}, \text{Red}\} \subseteq \{\text{Red}, \text{Green}, \text{Blue}\}$$

Note that with this definition, it so happens that

For any set X , $\emptyset \subseteq X$

For any set X , $X \subseteq X$

For any sets X and Y , $X \subseteq Y$ and $Y \subseteq X$ iff $X = Y$

superset, \supseteq

Those with Hebraic tendencies will appreciate the less-used \supseteq , standing for *superset*, that is, $X \supseteq Y$ if Y is completely contained in X . Also, if you want to exclude the possibility that a subset is equal to the larger set (say X is contained in Y , but Y has some elements not in X), what you have in mind is called a *proper subset*, symbolized by $X \subset Y$. Compare \subseteq and \subset with $<$ and \leq . Rarely, though, will we want to restrict ourselves to proper subsets.

proper subset, \subset

union, \cup

Often it is useful to take all the elements in two or more sets and consider them together. The resulting set is called the *union*, and its construction is symbolized by \cup . The union of two sets is the set of elements in *either* set.

$$\{\text{Orange, Red}\} \cup \{\text{Green, Red}\} = \{\text{Orange, Red, Green}\}$$

intersection, \cap

If any element occurs in both of the original sets, it still occurs only once in the resulting set. There is no notion of something occurring twice in a set. On the other hand, sometimes we will want to consider only the elements in *both* sets. We call that the *intersection*, and use \cap .

$$\{\text{Orange, Red}\} \cap \{\text{Green, Red}\} = \{\text{Red}\}$$

At this point it is very important to understand that $X \cap Y$ means “the set where X and Y overlap.” It does not mean “ X and Y overlap at some point.” It is a noun, not a sentence. This will be reemphasized in the next chapter.

difference, $-$

The fanciest operation on sets for this chapter is set *difference*, which expresses the set the resulting when we remove all the element of one set from another. We use the subtraction symbol for this, say $X - Y$. Y may or may not be a subset of X .

$$\{\text{Orange, Red, Green, Blue}\} - \{\text{Blue, Orange}\} = \{\text{Red, Green}\}$$

$$\{\text{Red, Green, Blue}\} - \{\text{Blue, Orange}\} = \{\text{Red, Green}\}$$

Finally, those circle diagrams have a name. *Venn diagrams* are so called after their inventor John Venn. They help visualize how different sets relate to each other in terms of containment and overlap. Note that the areas of the regions have no meaning—a large area might not contain more elements than a small area, for example.

Exercises

Let T be the set of trees, D be the set of deciduous trees, and C be the set of coniferous trees. In exercises 1–6, write the statement symbolically.

1. Oak is a deciduous tree.
2. Pine is not a deciduous tree.
3. All coniferous trees are trees.
4. Deciduous trees are those that are trees but are not coniferous.
5. Deciduous trees and coniferous trees together make all trees.
6. There is no tree that is both deciduous and coniferous.
7. Write $[2.3, 9.5)$ in set notation.

In exercises 8–15, determine whether each statement is true or false.

8. $-12 \in \mathbb{N}$.

9. $\mathbb{A} \subseteq \mathbb{C}$.

10. $\mathbb{R} \subseteq \mathbb{C} \cap \mathbb{R}^{-1}$

11. $4 \in \mathbb{C}$.

12. $\mathbb{Q} \cap \mathbb{T} = \emptyset$.

13. $\frac{1}{63} \in \mathbb{Q} - \mathbb{R}$.

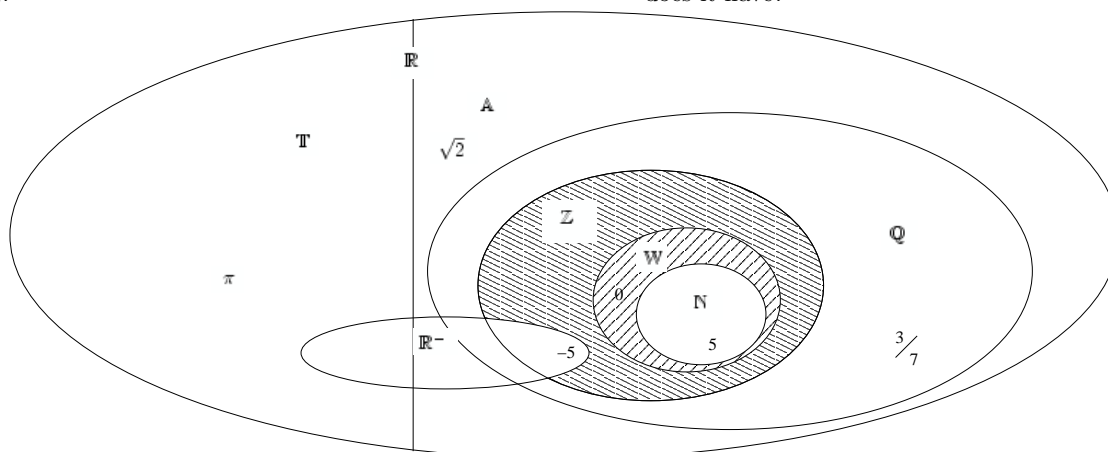
14. $\mathbb{Z} - \mathbb{R}^{-1} = \mathbb{W}$.

15. $\mathbb{T} \cup \mathbb{Z} \subseteq \mathbb{A}$.

16. All of the labeled sets we considered in Section 1.1 have an infinite number of elements, even though some are completely contained in others. (We will later consider whether all infinities should be considered equal.) However, two regions have a finite number of elements.

a. Describe the region shaded . How many elements does it have?

b. Describe the region shaded . How many elements does it have?



Chapter 2

Expressions and Types

2.1 Expressions

One of the most important themes in this course is the modeling or representation of mathematical concepts in a computer system. Ultimately, the concepts are modeled in computer memory; the arrangements of bits is information which we interpret as representing certain concepts, and we program the computer to operate on that information in a way consistent with our interpretation.

An *expression* is a programming language construct that expresses something. ML’s interactive mode works as a cycle: you enter an expression, which it will *evaluate*. A *value* is the result of the evaluation of an expression. To relate this to the previous chapter, a value is like an element. An expression is a way to describe that element. For example, 5 and $7 - 2$ are two ways to express the same element of \mathbb{N} .

expression

evaluate

value

When you start ML, you will see a hyphen, which is ML’s prompt, indicating it is waiting for you to enter an expression. The way you communicate to ML is to enter an expression followed by a semicolon and pressing the “enter” key. (If you press “enter” before the expression is finished, you will get a slightly different prompt, marked by an equals sign; this indicates ML assumes you have more to say.)

Try entering 5 into the ML prompt. Text that the user types into the prompt will be in **typewriter font**; ML’s response will be in *slanted typewriter font*.

```
- 5;
```

```
val it = 5 : int
```

The basic form you use is

<expression> ;

This is what the response means:

val is short for “value”, indicating this is the value the ML interpreter has found for the expression you entered.

it is a *variable*. A variable is a symbol that represents a value in a given context. Note that this means that a variable, too, is an expression; however, unlike the symbol 5, the value associated with the variable changes as you declare it to. Variables in ML are like those you are familiar with from mathematics (and other programming languages, if you have programmed before), and you can think of a variable as a box that stores values. Unless directed otherwise, ML automatically stores the value of the most recently evaluated expression in a variable called *it*.

variable

5 is the value of the expression (not surprisingly, the value of 5 is 5).

`int` is the type of the expression (in this case, short for “integer”), about which we will say more soon.

We can make more interesting expressions using mathematical operators. We can enter

```
- 7 - 2;
```

```
val it = 5 : int
```

subexpression
operator
operand

Note that this expression itself contains two other expressions, 7 and 2. Smaller expressions that compose a larger expression are called *subexpressions* of that expression. `-` is an *operator*, and the subexpressions are the *operands* of that operator. `+` means what you would expect, `*` stands for multiplication, and `~` is used as a negative sign (having one operand, to distinguish it from `-`, which has two); division we will discuss later. To express (and calculate) $67 + 4 \times -13$, type

```
- 67 + 4 * ~ 13;
```

```
val it = 15 : int
```

2.2 Types

type

So far, all these values have had the type `int` (we will use sans serif font for types). A *type* is a set of values that are related by the operations that can be performed on them. This provides another example of modeling concepts on a computer: a type models our concept of set.

Nevertheless, this also demonstrates the limitations of modeling because types are more restricted than our general concept of a set. ML does not provide a way to use the concepts of subsets, unions, or intersections on types. We will later study other ways to model sets to support these concepts. Moreover, the type `int`, although it corresponds to the set \mathbb{Z} in terms of how we interpret it, does not equal the set \mathbb{Z} . The values (elements) of `int` are computer representations of integers, not the integers themselves, and since computer memory is limited, `int` comprises only a finite number of values. On the the computer used to write this book, the largest integer ML recognizes is 1073741823. Although $1073741824 \in \mathbb{Z}$, it is not a valid ML `int`.

```
- 1073741824;
```

```
stdIn:6.1-6.11 Error: int constant too large
```

ML also has a type `real` corresponding to \mathbb{R} . The operators you have already seen are also defined for reals, plus `/` for division.

```
- ~4.73;
```

```
val it = ~4.73 : real
```

```
- 7.1 - 4.8 / 63.2;
```

```
val it = 7.02405063291 : real
```

```
- 5.3 - 0.3;
```

```
val it = 5.0 : real
```


Notice that `5.0` has type `real`, not type `int`. Again the set modeling breaks down. `int` is not a subset (or subtype) of `real`, and `5.0` is a completely different value from `5`.

A consequence of `int` and `real` being unrelated is that you cannot mix them in arithmetic expressions. English requires that the subject of a sentence have the same number (singular or plural) as the main verb, which is why it does not allow a sentence like, “Two dogs walks down the street.” This is called subject-verb agreement. In the same way, these ML operators require *type agreement*. That is, `+`, for example, is defined for adding two `reals` and for adding two `ints`, but not one of each. Attempting to mix them will generate an error.

type agreement

```
- 7.3 + 5;
```

```
stdIn:16.1-16.8 Error: operator and operand don't agree [literal]
operator domain: real * real
operand:         real * int
in expression:
  7.3 + 5
```

This rule guarantees that the result of an arithmetic operation will have the same type as the operands. This complicates the division operation on `ints`. We expect that $5 \div 4 = 1.25$ —as we noted in the previous chapter, division takes us out of the circle of integers. Actually, the `/` operator is not defined for `ints` at all.

```
- 5/4;
```

```
stdIn:20.2 Error: overloaded variable not defined at type
symbol: /
type: int
```

Instead, another operator performs *integer division*, which computes the integer quotient (that is, ignoring the remainder) resulting from dividing two integers. Such an operation is different enough from real number division that it uses a different symbol: the word `div`. The remainder is calculated by the modulus operator, `mod`.

integer division

```
- 5 div 3;
```

```
val it = 1 : int
```

```
- 5 mod 3;
```

```
val it = 2 : int
```

But would it not be useful to include both `reals` and `ints` in some computations? Yes, but to preserve type purity and reduce the chance of error, ML requires that you convert such values explicitly using one of the converters in the table below. Note the use of parentheses. These “converters” are functions, as we will see in a later chapter.

Converter	converts from	to	by
<code>real()</code>	<code>int</code>	<code>real</code>	appending a 0 decimal portion
<code>round()</code>	<code>real</code>	<code>int</code>	conventional rounding
<code>floor()</code>	<code>real</code>	<code>int</code>	rounding down
<code>ceil()</code>	<code>real</code>	<code>int</code>	rounding up
<code>trunc()</code>	<code>real</code>	<code>int</code>	throwing away the decimal portion

For example,

```

- 15.3 / real(6);

val it = 2.55 : real

- trunc(15.3) div 6;

val it = 2 : int

```

2.3 Variables

Since variables are expressions, they are fair game for entering into a prompt.

```

- it;

val it = 2 : int

```

We too little appreciate what a powerful thing it is to know a name. Having a name by which to call something allows one to exercise a certain measure of control over it. As Elwood Dowd said when meeting Harvey the rabbit, “You have the advantage on me. You know my name—and I don’t know yours” [2]. More seriously, the Third Commandment shows how zealous God is for the right use of his name. Geerhardus Vos comments:

It is not sufficient to think of swearing and blasphemy in the present-day common sense of these terms. The word is one of the chief powers of pagan superstition, and the most potent form of word-magic is name-magic. It was believed that through the pronouncing of the name of some supernatural entity this can be compelled to do the bidding of the magic-user. The commandment applies to the divine disapproval of such practices specifically to the name “Jehovah.” [13].

Compare also Ex 6:2 and Rev 19:12. A name is a blessing, as in Gen 32:26-29 and Rev 2:17. Even more so, to *give* a name to something is act of dominion over it, as in Gen 2:19-20. Think of how in the game of tag the player who is “it” has the power to place that name on someone else.

The name `it` in ML gives us the power to recall the previous value

```

- it * 15;

val it = 30 : int

- it div 10;

val it = 3 : int

```

To name a value something other than `it`, imitate the interpreter’s response using `val`, the desired variable, and equals, something in the form of

```

val <identifier> = <expression>;

- val x = 5;

val x = 5 : int

```

You could also add a colon and a type after the expression, like the interpreter does in its response, but there is no need to—the interpreter can figure that out on its own. However, we will see a few occasions much later when complicated expressions need an explicit typing for disambiguation.

An *identifier* is a programmer-given name, such as a variable. ML has the following rules for valid identifiers:

identifier

1. The first character must be a letter¹.
2. Subsequent characters must be letters, digits, or underscores.
3. Identifiers are case sensitive.

It is convention to use mainly lowercase letters in variables. If you use several words joined together to make a variable, capitalize the first letter of the subsequent words.

```
- val secsInMinute = 60;

val secsInMinute = 60 : int

- val minutesInHour = 60;

val minutesInHour = 60 : int

- val hoursInDay = 24;

val hoursInDay = 24 : int

- val daysInYear = 365;

val daysInYear = 365 : int

- val secsInHour = secsInMinute*minutesInHour;

val secsInHour = 3600 : int

- val hoursInYear = hoursInDay * daysInYear;

val hoursInYear = 8760 : int

- val secsInYear = secsInHour * hoursInYear;

val secsInYear = 31536000 : int
```

2.4 Making your own types

Finally, we will briefly look at how to define your own type. A programmer-defined type is called a *datatype*, and you can think of such a type as a set of arbitrary elements. Suppose we want to model species of felines or trees.

datatype

```
- datatype feline = Leopard | Lion | Tiger | Cheetah | Panther;

datatype feline = Cheetah | Leopard | Lion | Panther | Tiger

- datatype tree = Oak | Maple | Pine | Elm | Spruce;

datatype tree = Elm | Maple | Oak | Pine | Spruce
```

¹Identifiers may also begin with an apostrophe, but only when used for a special kind of variable

When defining a datatype, separate the elements by vertical lines called “pipes,” a character that appears with a break in the middle on some keyboards. The name of the type and the elements must be valid identifiers. As demonstrated here, it is conventional for the names of types to be all lower case, whereas the elements have their first letters capitalized. Notice that in its response, ML alphabetizes the elements; as with any set, their order does not matter. Until we learn to define functions, there is little of interest we can use datatypes for. Arithmetic operators, not surprisingly, cannot be used.

```
- val cat = Leopard;
```

```
val cat = Leopard : feline
```

```
- val evergreen = Spruce;
```

```
val evergreen = Spruce : tree
```

```
- cat + evergreen;
```

```
stdIn:49.1-49.16 Error: operator and operand don't agree [tycon mismatch]
```

```
operator domain: feline * feline
```

```
operand:         feline * tree
```

```
in expression:
```

```
cat + evergreen
```

```
stdIn:49.5 Error: overloaded variable not defined at type
```

```
symbol: +
```

```
type: feline
```

Exercises

1. Determine the type of each of the following.
 - (a) `5.3 + 0.3`
 - (b) `5.3 - 0.3`
 - (c) `5.3 < 0.3`
 - (d) `24.0 / 6.0`
 - (e) `24.0 * 6.0`
 - (f) `24 * 6`
2. Is `ceil(15.2)` an expression? If no, why not? If so, what is its value and what is its type? Would ML accept `ceil(15)`? Why or why not?
3. Make two points, stored in variables `point1` and `point2`, and calculate the distance between the two points. (Recall that this can be done using the Pythagorean theorem.)
4. Which of the following are valid ML identifiers?
 - (a) `wheaton`
 - (b) `wheaton_college`
 - (c) `wheaton'college`
 - (d) `wheatonCollege`
 - (e) `wHeAtoN`
 - (f) `_wheaton`
 - (g) `wheaton12`
 - (h) `12wheaton`
5. Redo the computation of the number of seconds in a year in Section 2.3, but take into consideration that there are actually 365.25 days in a year, and so `daysInYear` should be of type `real`. Your final answer should still be an `int`.
6. Mercury orbits the sun in 87.969 days. Calculate how old you will be in 30 Mercury-years. Your answer should depend on your birthday (that is, don't simply add a number of years to your age), but it should be an `int`.
7. Create a datatype of varieties of fish.
8. Use ML to compute the circumference and area of a circle and the volume and surface area of a sphere, first each of radius 12, then of radius 12.75.
9. Store the values 4.5 and 6.7 in variables standing for base and height, respectively, of a rectangle, and use the variables to calculate the area. Then do the same but assuming they are the base and height of a triangle.

Chapter 3

Set Operations

3.1 Axiomatic foundations

It is worth reemphasizing that the definitions we gave for *set* and *element* in Chapter 1 were informal. Formal definitions for them are, in fact, impossible. They constitute basic building blocks of mathematical language. In geometry, the terms “point,” “straight line,” and “planes” have a similar primitivity[12]. Instead of formal definitions, these sorts of objects were described using axioms, propositions that were assumed rather than proven, and by which all other propositions were proven.

Here are two axioms (of many others) that are used to ground set theory.

Axiom 1 (Existence.) *There is a set with no elements.*

Axiom 2 (Extensionality.) *If every element of a set X is an element of a set Y and every element of Y is an element of X , then $X = Y$.*

We may not know what sets and elements are, but we know that it is possible for a set to have no elements; Axiom 1 tells us that there is an empty set. Axiom 2 tells us what it means for sets to be equal, and this implicit definition captures what we mean when we say that sets are *unordered*, since if two different sets had all the same elements but in different orders, they in fact would not be two different sets, but one and the same.

Moreover, putting these two axioms together confirms that we may speak meaningfully not only of *an* empty set, but *the* empty set, since there is only one. Suppose for the sake of argument there were two empty sets. Since they have all the same elements—that is, none at all—they are actually the same set. This is what we would call a *trivial* application of the axiom, but it is still valid. Hence the empty set is unique.

A complete axiomatic foundation for set theory is tedious and beyond the scope of our purposes. We will touch on these axioms once more when we study proof in Chapter 10, but the important lesson for now is the use of axioms to describe basic and undefinable terms. Axioms are good if they correctly capture our intuition; notice that on page 8 we essentially derived the Axiom 2 from our informal definition of set.

3.2 Operations and visualization

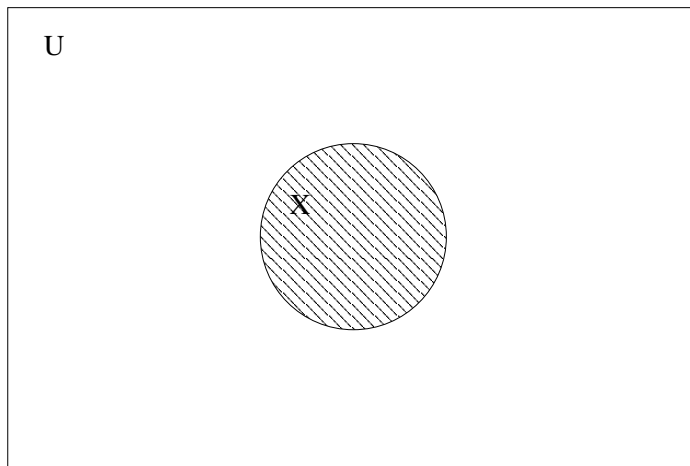
One of the two main objectives of this course is learning how to write formal proofs. To prime you for that study, we will verify some propositions visually.

Any discussion that uses set theory takes place within an arbitrarily (and sometimes implicitly) fixed set. If we were talking about the sets { Panther, Lion, Tiger } and sets { Bob cat, Cheetah, Panther, Tiger }, then the context likely implies that all sets in the discussion are subsets of the set of felines. However, if the set { Panther, Cheetah, Weasel, Sponge } is mentioned, then the backdrop

is probably the set of animals, or the set { Cheetah, Sponge, Apple tree } would imply the set of living things. Either way, there is some context of which all sets in the discussion are a subset. It is unlikely one would ever speak of the set { Green, Sponge, Acid reflux, Annuciation } unless the context is, say, the set of English words.

universal set

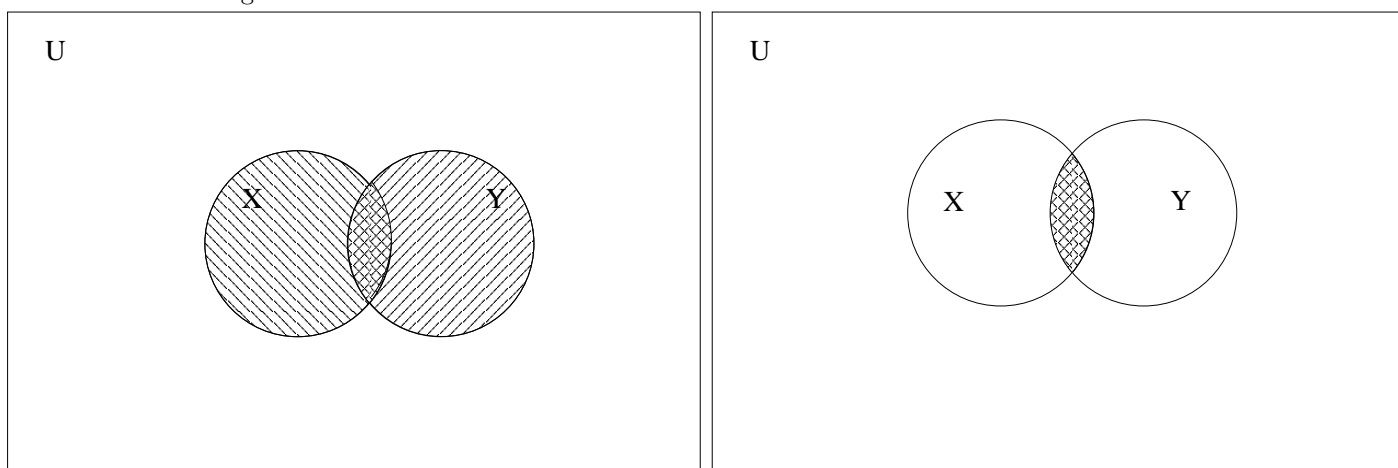
That background set relevant to the context is called the *universal set*, and designated U . In Venn diagrams, it is often drawn as a rectangle framing the other sets. Further, shading is often used to highlight particular sets or regions among sets. A simple diagram showing a single set might look like this:



cardinality

We also pause to define the *cardinality* of a finite set, which is the number of elements in the set. This is symbolized by vertical bars, like absolute value. If U is the set of lowercase letters, then $|\{a, b, c, d\}| = 4$. Note that this definition does not allow you to say, for example, that the cardinality of \mathbb{Z} is infinity; rather, cardinality is defined only for finite sets. Expanding the idea of cardinality to infinite sets brings up interesting problems we will explore sometime later.

We can visualize basic set operations by drawing two overlapping circles, shading one of them (X , defined as above) with diagonal lines and the other ($Y = \{c, d, e, f\}$) with cross-hatching. The union $X \cup Y$ is anything that is shaded at all, and the intersection $X \cap Y$ is the region shaded with both patterns, all shown on the left; on the right is the intersection alone.



Note that it is not true that $|X \cup Y| = |X| + |Y|$, since the elements in the intersection, $X \cap Y = \{c, d\}$, would be counted twice. However, do not assume that just because sets are drawn so that they overlap that they in fact share some elements. The overlap region may be empty. We say that two sets X and Y are *disjoint* if they have no elements in common, that is, $X \cap Y = \emptyset$. Note that if X and Y are disjoint, then $|X \cup Y| = |X| + |Y|$.

disjoint

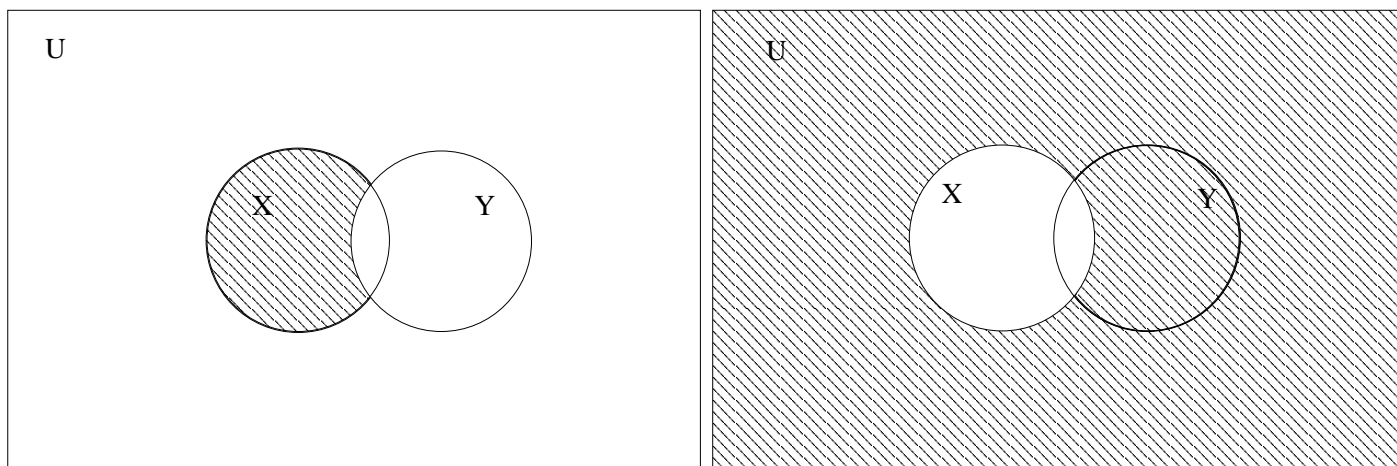
pairwise disjoint

We can expand the notion of disjoint by considering a larger collection of sets. A set of sets $\{A_1, A_2, \dots, A_n\}$ is *pairwise disjoint* if no pair of sets have any elements in common, that is, if for

all i, j , $1 \leq i, j \leq n$, $i \neq j$, $A_i \cap A_j = \emptyset$.

Remember that the difference between two sets X and Y is $X - Y = \{x | x \in X \text{ and } x \notin Y\}$. From our earlier X and Y , $X - Y = \{a, b\}$. Having introduced the universal set, it now makes sense also to talk about the *complement* of a set, $\overline{X} = \{x | x \notin X\}$, everything that (is in the universal set but) is not in the given set. In our example, $\overline{X} = \{e, f, g, h, \dots, z\}$. Difference is illustrated to the left and complement to the right; Y does not come into play in set complement, but is drawn for consistency.

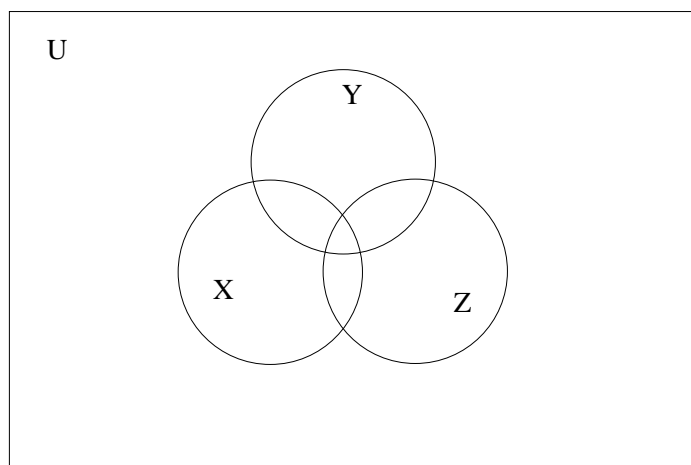
complement






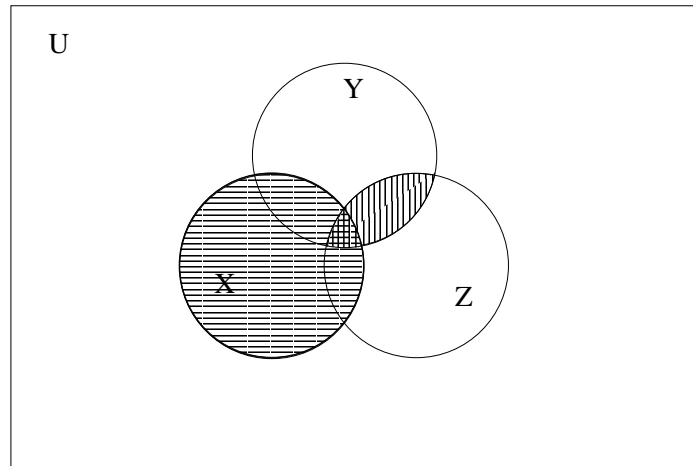
Now we can use this drawing and shading method to verify propositions about set operations. For example, suppose X , Y , and Z are sets, and consider the proposition




$$X \cup (Y \cap Z) = (X \cup Y) \cap (X \cup Z)$$

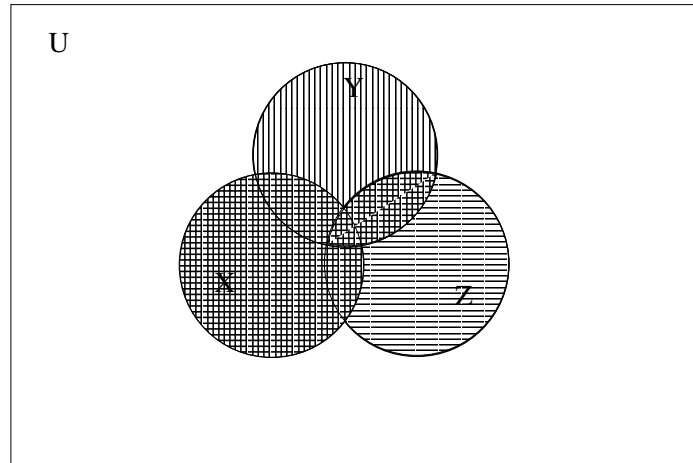
This is called the *distributive law*; compare with the law from algebra $x \cdot (y + z) = x \cdot y + x \cdot z$ for $x, y, z \in \mathbb{R}$. First we draw a Venn diagram with three circles.



Then we shade it according to the left side of the equation and, separately, according to the right side and compare the two drawings. First, shade X with  and $Y \cap Z$ with . The union operation indicates all the shaded regions together. (Note that $X \cap (Y \cap Z)$ is shaded , but that is not important for our present task.) Thus the left side of the equation:



Now, in a separate picture, shade $X \cup Y$ with  and $X \cup Z$ with . The intersection of these two sets is the region that is double-shaded, .



Since the total shaded region in the first picture is the same as the double-shaded region in the second picture, we have verified the proposition. Notice how graphics, with a little narration to help, can be used for an informal, intuitive proof.

3.3 Powersets, cartesian products, and partitions

powerset

Finally, we introduce three specialized set definitions. The *powerset* of a set X is the set of all subsets of X and is symbolized by $\mathcal{P}(X)$. Formally,

$$\mathcal{P}(X) = \{Y \mid Y \subseteq X\}$$

If $X = \{1, 2, 3\}$, then $\mathcal{P}(X) = \{\{1, 2, 3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{1\}, \{2\}, \{3\}, \emptyset\}$. It is important to notice that for any set X , $X \in \mathcal{P}(X)$ and $\emptyset \in \mathcal{P}(X)$, since X is a subset of itself and \emptyset is a subset of everything. It so happens that for finite sets, $|\mathcal{P}(X)| = 2^{|X|}$.

ordered pair

An *ordered pair* is two elements (not necessarily of the same set) written in a specific order. Suppose X and Y are sets, and say $x \in X$ and $y \in Y$. Then we say that (x, y) is an ordered pair *over* X and Y . We say two ordered pairs are equal, say $(x, y) = (w, z)$ if $x = w$ and $y = z$. An ordered pair is different from a set of cardinality 2 in that it is ordered. Moreover, the *Cartesian product* of two sets, X and Y , written $X \times Y$, is the set of all order pairs over X and Y . Formally,

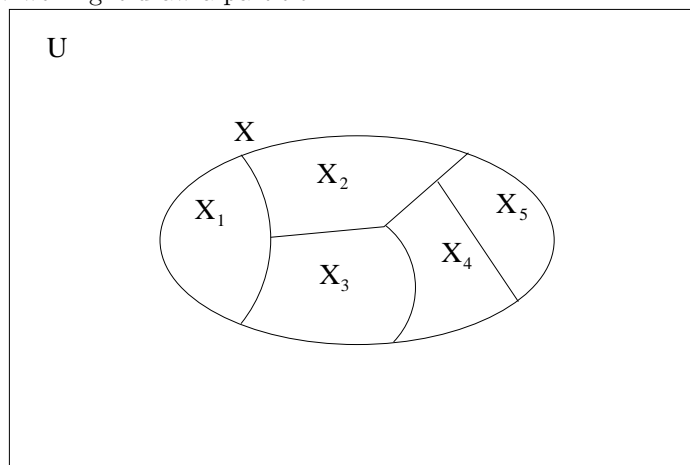
Cartesian product

$$X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}$$

If $X = \{1, 2\}$ and $Y = \{2, 3\}$, then $X \times Y = \{(1, 2), (1, 3), (2, 2), (2, 3)\}$. The Cartesian product, named after Decartes, is nothing new to you. The most famous Cartesian product is $\mathbb{R} \times \mathbb{R}$, that is, the Cartesian plane. Similarly, we can define ordered triples, quadrupals, and n -tuples, and corresponding higher-ordered products.

If X is a set, then a *partition* of X is a set of non-empty sets $\{X_1, X_2, \dots, X_n\}$ such that X_1, X_2, \dots, X_n are pairwise disjoint and $X_1 \cup X_2 \cup \dots \cup X_n = X$. Intuitively, a partition of a set is a bunch of non-overlapping subsets that constitute the entire set. From Chapter 1, \mathbb{T} and \mathbb{A} make up a partition of \mathbb{R} . Here is how we might draw a partition:

partition



Exercises

1. Complete the on-line Venn diagram drills found at www.ship.edu/~deensl/DiscreteMath/flash/ch3/sec3_1/venntwoset.html and www.ship.edu/~deensl/DiscreteMath/flash/ch3/sec3_1/vennthreeset.html

Let A , B , and C be sets, subsets of the universal set U . Draw Venn diagrams to show the following (do not draw C in the cases where it is not used).

2. $(A \cap B) - A$.
 3. $(A - B) \cup (B - A)$.
 4. $(A \cup B) \cap (A \cup C)$.
 5. $\overline{(A \cap B)} \cap (A \cup C)$.
- Describe the powerset (by listing all the elements) of the following.
6. $\{1, 2\}$
 7. $\{a, b, c, d\}$
 8. \emptyset
 9. $\mathcal{P}(\{1, 2\})$.
 10. Describe three distinct partitions of the set \mathbb{Z} . For example, one partition is the set of evens and the set of odds (remember that these *two* sets make *one* partition).

Chapter 4

Tuples and Lists

4.1 Tuples

One of the last things we considered in the previous chapter was the Cartesian product over sets. ML has a ready-made way to represent ordered pairs—or their generalized counterparts, *tuples*, as they are more frequently spoken of in the context of ML. In ML, a tuple is made by listing expressions, separated by commas and enclosed in parentheses, following standard mathematical notation. The expressions are evaluated, and the values are displayed, again in a standard way.

tuples

```
- (2.3 - 0.7, ~8.4);
```

```
val it = (1.6,~8.4) : real * real
```

Note that the type is `real * real`, corresponding to $\mathbb{R} \times \mathbb{R}$. We can think of this as modeling a point in the real plane. `(1.6, 8.4)` is itself a value of that type. We can store this value in a variable; we also can extract the components of this value using `#1` and `#2` for the first and second number in the pair, respectively.

```
- val point = (7.3, 27.85);
```

```
val point = (7.3,27.85) : real * real
```

```
- #1(point);
```

```
val it = 7.3 : real
```

```
- #2(point);
```

```
val it = 27.85 : real
```

Suppose we want to shift the point up 5 units and over .3 units.

```
- val newx = #1(point) + 5.0;
```

```
val newx = 12.3 : real
```

```
- val newy = #2(point) + 0.3;
```

```
val newy = 28.15 : real
```

```
- (newx, newy);

val it = (12.3,28.15) : real * real
```

Note that although we mentioned “shifting the point,” we really are not making effecting a change on the variable `point`—it has stayed the same. (If you have programmed before, note well that this is different from changing the state of an object or array.)

```
- point;

val it = (7.3,27.85) : real * real
```

We can make tuples of any size and of non-uniform types. We can make tuples of any types, even of tuple types.

```
- (4.3, 7.9, ~0.002);

val it = (4.3,7.9,~0.002) : real * real * real

- datatype Bird = Sparrow | Finch | Robin | Owl | Gull | Dove | Eagle
=                | Vulture | Hawk | Falcon | Penguin | Duck | Loon
=                | Goose | Chicken | Turkey;

    datatype Bird
    = Chicken
    | Dove
    ...

- (4.5, Eagle, 17, Finch);

val it = (4.5,Eagle,17,Finch) : real * Bird * int * Bird

- (~3.1, Owl, (3, 5));

val it = (~3.1,Owl,(3,5)) : real * Bird * (int * int)

- #2(#3(it));

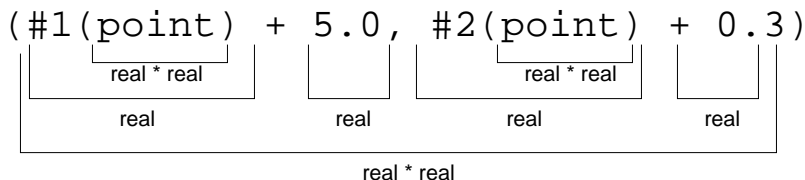
val it = 5 : int
```

It is most important to observe the types of the various expressions we are considering. Type correctness is how parts of a computer program are assembled in a meaningful way. For the simplicities sake, let us confine ourselves to thinking of pairs of *reals*. The operators `#1` and `#2` consume a pair (a value of type `real * real`) and produces a *real*. The parentheses and comma consume two *reals* and produces a pair of *reals* (`real * real`). Notice the difference between “two *reals*” (two distinct values) and “a pair of *reals*” (one value). Consider again

```
- (#1(point) + 5.0, #2(point) + 0.3);

val it = (12.3,28.15) : real * real
```

We can analyze the types of this expression and its subexpressions:



4.2 Lists

At first glance, it might seem that tuples are reasonable candidates for representing sets in ML. We can group three values together and consider them a single entity, as in

```
- (Robin, Duck, Chicken);

val it = (Robin,Duck,Chicken) : Bird * Bird * Bird
```

This is a poor solution because even though a tuple's length is arbitrary, it is still fixed. The type `Bird * Bird * Bird` is more restricted than a "set of Birds." A 4-tuple of Birds has a completely different type.

```
- (Finch, Goose, Penguin, Dove);

val it = (Finch,Goose,Penguin,Dove) : Bird * Bird * Bird * Bird
```

An alternative which will enable us better to represent the mathematical concept of sets is a *list*. Cosmetically, the difference between the two is to use square brackets instead of parentheses. Observe how the interpreter responds to these various attempts at using lists.

list

```
- [Finch, Robin, Owl];

val it = [Finch,Robin,Owl] : Bird list

- [Vulture, Sparrow];

val it = [Vulture,Sparrow] : Bird list

- [Eagle];

val it = [Eagle] : Bird list

- [];

val it = [] : 'a list

- [13, 28, 22, 23, 4, 57, 86];

val it = [13,28,22,23,4,57,86] : int list

- [5, 5.5, Sparrow];

stdIn:29.1-29.18 Error: operator and operand don't agree [tycon mismatch]
operator domain: real * real list
operand:          real * Bird list
in expression:
  5.5 :: Sparrow :: nil
```

Every list made up of birds has the same type, `Bird list`, regardless of how many Birds there are. Likewise we can have a list of ints, but unlike tuples we cannot have a list of mixed types. The type of which the list is made up is called its *base type*. The interpreter typed the expression `[]` as `'a list`; `'a` is a *type variable*, a symbol ML uses to stand for an unknown type. Two lone square braces is obviously an empty list, but even an empty list must have a base type. This is the first case we have seen of an expression whose type ML cannot infer from context. We can disambiguate this using *explicit typing*, which is done by following an expression with a colon and the type. For example, we can declare that we want an empty list to be considered a list of Birds.

base type

type variable

explicit typing

```
- [] : Bird list;

val it = [] : Bird list
```

It is perfectly logical to speak of lists of lists—that is, the base type of a list may be itself a list type.

```
- [[Loon], [Robin, Duck]];

val it = [[Loon],[Robin,Duck]] : Bird list list
```

Make sure the difference between tuples and lists is understood. An n -tuple has n components, and n is a fundamental aspect of the type. Lists (with base type 'a), on the other hand, are considered to have exactly two components: the first element (called the *head*, of type 'a) and the rest of the list (called the *tail*, of type 'a list)—all this, of course, unless the list is empty. Thus we defy the grammar school rule that one cannot define something in terms of itself by saying an 'a list is

- an empty list, or
- an 'a followed by an 'a list.

Corresponding to the difference in definition, a programmer interacts with lists in a way different from how tuples are used. A tuple has n components referenced by #1, #2, etc. A list has these two components referenced by the *accessors* `hd` for head and `tl` for tail.

```
- val sundryBirds = [Robin, Penguin, Gull, Loon];

val sundryBirds = [Robin,Penguin,Gull,Loon] : Bird list

- hd(sundryBirds);

val it = Robin : Bird

- tl(sundryBirds);

val it = [Penguin,Gull,Loon] : Bird list

- tl(it);

val it = [Gull,Loon] : Bird list

- tl(it);

val it = [Loon] : Bird list

- tl(it);

val it = [] : Bird list

- tl(it);

uncaught exception Empty
```


`hd` and `tl` can be used to slice up a list, one element at a time. However, it is an error to try to extract the tail (or head, for that matter) on an empty list.

We have seen how to make a list using square braces and how to take lists apart. We can take two lists and *concatenate* them—that is, tack one on the back of the other to make a new list—using the *cat* operator, `@`.

*concatenate**cat, @*

```
- [Owl, Finch] @ [Eagle, Vulture];

val it = [Owl,Finch,Eagle,Vulture] : Bird list
```

We also can take a value of the base type and a list and considering them to be the head and tail, respectively, of a new list. This is by the *construct* or *cons* operator, `::`.

cons, ::

```
- Robin::it;

val it = [Robin,Owl,Finch,Eagle,Vulture] : Bird list
```

Cons must take an item and a list; cat must take two lists.

```
- it::Hawk;

stdIn:40.1-40.9 Error: operator and operand don't agree [tycon mismatch]
operator domain: Bird list * Bird list list
operand:         Bird list * Bird
in expression:
  it :: Hawk

- Hawk@[Loon];

stdIn:1.1-1.12 Error: operator and operand don't agree [tycon mismatch]
operator domain: 'Z list * 'Z list
operand:         Bird * Bird list
in expression:
  Hawk @ Loon :: nil

- Sparrow::Robin::Turkey;

stdIn:1.1-30.7 Error: operator and operand don't agree [tycon mismatch]
operator domain: Bird * Bird list
operand:         Bird * Bird
in expression:
  Robin :: Turkey
```

However, cons works from right to left, so these next two are fine:

```
- Sparrow::Robin::[Turkey];

val it = [Sparrow,Robin,Turkey] : Bird list

- Sparrow::Robin::Turkey::[];

val it = [Sparrow,Robin,Turkey] : Bird list
```

And, once more, using cat and cons on lists of lists:

```

- [Loon]::[];

val it = [[Loon]] : Bird list list

- [Robin]::it;

val it = [[Robin],[Loon]] : Bird list list

- [[Duck, Vulture]]@it;

val it = [[Duck,Vulture],[Robin],[Loon]] : Bird list list

```

4.3 Lists vs. tuples vs. arrays

random access
sequential access

By using a list instead of a tuple, we are surrendering the ability to extract an element from an arbitrary position in one step, a feature of tuples called *random access*. The way we consider items in a list is called *sequential access*. This is the price of using lists—we sacrifice random access for indefinite length. ML also provides another sort of container called an array, something we will touch on only briefly here, though it certainly is familiar to all who have programmed before.

array
index

An *array* is a finite, ordered sequence of values, each value indicated by an integer *index*, starting from 0. Since it is a non-primitive part of ML, the programmer must load a special package that allows the use of arrays.

```

- open Array;

```

One creates a new array by typing `array(n, v)`, which will evaluate to an array of size *n*, with each position of the array initialized to the value *v*. The value at position *i* in an array is produced by `sub(A, i)`, and the value at that position is modified to contain *v* by `update(A, i, v)`.

```

- val A = array(10, 0);

val A = [|0,0,0,0,0,0,0,0,0,0|] : int array

- update(A, 2, 16);

val it = () : unit

- update(A, 3, 21);

val it = () : unit

- A;

val it = [|0,0,16,21,0,0,0,0,0,0|] : int array

- sub(A, 3);

val it = 21 : int

```

The interpreter’s response `val it = () : unit` will be explained later. Note that arrays can be changed—they are *mutable*. Although we can generate new tuples and lists, we cannot *change* the value of a tuple or list. However, unlike lists, new arrays cannot be made by concatenating two arrays together.

mutable

Much of the work of programming is weighting trade-offs among options. In this case, we are considering the appropriateness of various data structures, each of which has its advantages and liabilities. The following table summarized the differences among tuples, lists, and arrays.

	Access	Concatenation	Length	Element types	Mutability
Tuples	random	unsupported	fixed	unrelated	immutable
Lists	sequential	supported	indefinite	uniform	immutable
Arrays	random	unsupported	indefinite	uniform	mutable

In this case, we want a data structure suitable for representing sets. Our choice to use lists comes because the concept of “list of X ” is so similar to “set of X ”—and because ML is optimized to operate on lists rather than arrays. However, there are downsides. A list, unlike a set, may contain multiple copies of the same element. The cat operator is, for example, a poor union operation because it will keep both copies if an element belongs to both subsets that are being unioned. Later we will learn to write our own set operations to operate on lists.

Exercises

In Exercises 1–9, analyze the type of the expression, as on page 26, or indicate that it is an error.

1. `[(Loon, 5), (Vulture, 6)][]`
2. `[tl([Sparrow, Robin, Turkey])]@[Owl, Finch]`
3. `[Owl, Finch]@tl([Sparrow, Robin, Turkey])`
4. `[Owl, Finch]::tl([Sparrow, Robin, Turkey])`
5. `[Owl, Finch]::[tl([Sparrow, Robin, Turkey])]`
6. `hd([Sparrow, Robin, Turkey])::[Owl, Finch]`
7. `((Finch, Robin), [2, 4])`
8. `[(Finch, Robin), (2, 4)]`
9. `[5, 12, ceil(7.3 * 2.1), #2(Owl, 17)]`

In Exercises 10–12, assume `collection` is a list with sufficient length for the given problem. Write an ML expression to produce a list as indicated.

10. Remove the second item of `collection` and tack it on the front.
11. Remove the third item of `collection` and tack it on the back.
12. Remove the first two items of `collection` and tack them on the back.

In Exercises 13–15, state whether it would be best to use an array, a tuple, or a list.

13. You have a collection of numbers that you wish to sort. The sorting method you wish to use involves cutting the collection into smaller parts and then joining them back together.
14. The units of data you are using have components of different type, but the units are always the same length, with the same types of components in the same order.
15. You have a collection of numbers that you wish to sort. The sorting method you wish to use involves interchanging pairs of values from various places in the collection.

Part II

Logic

Chapter 5

Logical Propositions and Forms

In this chapter, we begin our study of formal logic. Logic is the set of rules for making deductions, that is, for producing new bits of information by piecing together other known bits of information. We study logic, in this course in particular, because logic is a foundational part of the language of mathematics, and it is the basis for all computing. The circuits of microchips, after all, are first modeling logical operations; that logic is then used to emulate other work, such as arithmetic. Logic, further, trains the mind and is a tool for any field, whether it be natural science, rhetoric, philosophy, or theology. Two things should be clarified before we begin.

First, you must understand that logic studies the *form* of arguments, not their contents. The argument

*If the moon is made of blue cheese, then Napoleon Dynamite is President.
The moon is made of blue cheese.
Therefore, Napoleon Dynamite is President.*

is perfectly logical. Its absurdity lies in the content of its premises. Similarly, the argument

*If an integer is a multiple of 3, then its digits sum to a multiple of 3.
The digits of 216 sum to 9, a multiple of 3.
Therefore 216 is a multiple of 3.*

is illogical. Nevertheless, everything except for *therefore* is true.

Second, and consequently, we must recognize logic as a *tool* for thinking; it is not thinking itself, nor is it a substitute for thinking. Logic just as easily can be employed to talk about unreality as it can reality. Common sense, observation, and intuition are necessary to find proper raw materials (premises) on which to apply logic. G. K. Chesterton observed that a madman's "most sinister quality is a horrible clarity of detail; a connection of one thing with another in a map more elaborate than a maze. . . He is not hampered by a sense of humor or by charity. . . He is the more logical for losing certain sane affections. . . The madman is not the man who has lost his reason. The madman is the man who has lost everything except his reason" [3].

5.1 Forms

Consider these two arguments:

If it is Wednesday or spinach is on sale, then I go to the store.
So, if I do not go to the store, then it is not Wednesday and spinach is not on sale.
If $x < -5$ or $x > 5$, then $|x| > 5$.
So, if $|x| \not> 5$, then $x \not< -5$ and $x \not> 5$.

What do these have in common? If we strip out everything except the words *if*, *or*, *then*, *so*, *not* and *and* (that is, if we strip out all the content but leave the logical connectors), we are left in either case with

If p or q , then r .
So, if not r , then not p and not q .

proposition

Notice that we replaced the content with variables. This allowed us to abstract from the two arguments to find a form common to both. These variables are something new because they do not stand for numbers but for independent clauses, grammatical items that have a complete meaning and can be true or false. In the terminology of logic, we say a *proposition* is a sentence that is true or false, but not both. The words *true*, *false*, and *sentence* we leave undefined, like *set* and *element*.

Since a proposition can be true or false, the following qualify as propositions:

$7 - 3 = 4$
 $7 - 4 = 3$
Bob is taking discrete mathematics.

By saying a proposition must be one or the other and not both, we disallow the following:

$7 - x = 4$
He is taking discrete mathematics.

In other words, a proposition must be complete enough (no variables) to be true or false.¹

5.2 Symbols

Logic uses a standard system of symbols, much of which was formulated by Alfred Tarski. We have already seen that variables can be used to stand for propositions, and you may have noticed that propositions can be joined together by connective words to make new propositions. These connective words stand for logical operations, similar to the operations used on numbers to perform arithmetic. The three basic operations are

Symbolization	Meaning
$\sim p$	Not p — having truth value opposite that of p
$p \wedge q$	p and q — true if both are true, false otherwise
$p \vee q$	p or q — true if at least one is true, false otherwise

negation
conjunction
disjunction

We call these operations *negation*, *conjunction*, and *disjunction*, respectively. Negation has a higher precedence than conjunction and disjunction; $\sim p \vee q$ means $(\sim p) \vee q$, not $\sim (p \vee q)$. Conjunction and disjunction are performed from left to right.

Take notice of how things transfer from everyday speech to the precision of mathematical notation. The words *and*, *but*, and *not* are not grammatical peers—that is, they are not all the same part of speech. The words *but* and *and* are conjunctions; *not* is an adverb. How can they become equivalent in their applicability when treated formally? In fact, they are not equivalent, and the difference appears in the number of operands they take. Remember that truth values correspond to independent clauses in English. \vee and \wedge are *binary*—they operate on two things, just as in natural language conjunctions hook two independent clauses together. \sim is *unary*—it operates on only one thing, just as the adverb *not* modifies a single independent clause.

To grow accustomed to mathematical notation, it is helpful to understand how to translate from English to equivalent symbols. In the following examples, assume r stands for “it is raining” and s stands for “it is snowing.” Compare the propositions in English with their symbolic representation.

¹Technically, *Bob is taking discrete mathematics* might not be a proposition if the context does not determine which Bob we are talking about; similarly, *He is taking discrete mathematics* might be a proposition if the context makes plain who the antecedent is of *he*. We use these only as examples of the need for a sentence to be well-defined to be a proposition.

It is neither raining nor snowing.	$\sim r \wedge \sim s$
It is raining and not snowing.	$r \wedge \sim s$
It is both raining and snowing.	$r \wedge s$
It is raining but not snowing.	$r \wedge \sim s$
It is either raining or snowing but not both.	$(r \vee s) \wedge \sim (r \wedge s)$.

Notice that “and” and “but” have the same symbolic translation. This is because both conjunctions have the same denotational, logical meaning. Their difference in English is in their connotations. Whichever we choose, we are asserting that two things are both true; “*a* but *b*” merely spins the statement to imply something like “*a* and *b* are both true, and *b* is surprising in light of *a*.”

If it is hard to swallow the idea that “and” and “but” mean the same thing, observe how another language differentiates things more subtly. Greek has three words to cover the same semantic range as our “and” and “but”: *kai*, meaning “and”; *alla*, meaning “but”; and *de*, meaning something halfway between “and” and “but,” joining two things together in contrast but not as sharply as *alla*.

5.3 Boolean values

Truth value is the most basic idea that digital computation machinery models because there are only two possible values—true and false—which can correspond to the two possible switch configurations—on and off. ML provides a type to stand for truth values, `bool`, short for *boolean*, named after George Boole. The two values of `bool` are `true` and `false`. Expressions and variables, accordingly, can have the type `bool`.

```
- true;

val it = true : bool

- false;

val it = false : bool

- it;

val it = false : bool

- val p = true;

val p = true : bool

- val q = false;

val q = false : bool
```

The basic boolean operators are named `not`, `andalso`, and `orelse`.

```
- p andalso q;

val it = false : bool

- p orelse q;

val it = true : bool
```

```
- (p orelse q) andalso not (p andalso q);
```

```
val it = true : bool
```

Comparison operators, testing for equality and related ideas, are different from others we have seen in that their results have types different from their operands. If we compare two ints, we do not get another int, but a *bool*. The ML comparison operators are `=`, `<`, `>`, `<=`, `>=`, and `<>`, the last three for \leq , \geq , and \neq , respectively.

```
- 5 <> 4;
```

```
val it = true : bool
```

```
- 5 <= 4;
```

```
val it = false : bool
```

Round-off error makes testing *reals* for equality and inequality unreliable, so ML disallows it. Instead, check for both `<` and `>`.

```
- 5.2 <> 4.3;
```

```
stdIn:24.1-24.11 Error: operator and operand don't agree [equality type required]
operator domain: ''Z * ''Z
operand:          real * real
in expression:
  5.2 <> 4.3
```

```
- 5.2 < 4.3 orelse 5.2 > 4.3;
```

```
val it = true : bool
```

5.4 Truth tables

Our definitions of the logical operators was informal, based on an intuitive appeal to our understanding of English words *not*, *and*, and *or*. Moreover, since we accepted *true* and *false* as primitive, undefined terms, you may have guessed that an axiomatic system underlies symbolic logic indicating how *true* and *false* are used. Our axioms are the formal descriptions of the basic operators, defining them as functions, with \sim taking one argument and \wedge and \vee taking two. We specify these functions using *truth tables*, tabular representations of logical operations where arguments to a function appear in columns on the left side and results appear in columns on the right. The rows stand for the possible combinations of argument values.

truth tables

p	$\sim p$	p	q	$p \wedge q$	$p \vee q$
T	F	T	T	T	T
T	F	T	F	F	T
F	T	F	T	F	T
F	T	F	F	F	F

Truth tables also can be used as a handy way to evaluate composite logical propositions. Suppose we wanted to determine the value of $(p \wedge q) \vee \sim (p \vee q)$ for the various assignments to p and q . We do this by making a truth table with columns on the left for the arguments; a set of columns in the middle for intermediate propositions, each one the result of applying the basic operators to the arguments or to other intermediate propositions; and a column on the right for the final answer.

p	q	$p \wedge q$	$p \vee q$	$\sim(p \vee q)$	$(p \wedge q) \vee \sim(p \vee q)$
T	T	T	T	F	T
T	F	F	T	F	F
F	T	F	T	F	F
F	F	F	F	T	T

Truth tables are a “brute force” way of attacking logical problems. For complicated propositions and arguments, they can be tedious, and we will learn more expeditious ways of exploring logical properties. However, truth tables are a sure method of evaluation to which we can fall back when necessary.

5.5 Logical equivalence

Consider the forms $\sim(p \wedge q)$ and $\sim p \vee \sim q$. This truth table evaluates them for all possible arguments.

p	q	$\sim p$	$\sim q$	$p \wedge q$	$\sim(p \wedge q)$	$\sim p \vee \sim q$
T	T	F	F	T	F	F
T	F	F	T	F	T	T
F	T	T	F	F	T	T
F	F	T	T	F	T	T

The two rightmost columns are identical. This is because the forms $\sim(p \wedge q)$ and $\sim p \vee \sim q$ are *logically equivalent*, that is, they have the same truth value for any assignments of their arguments. (We also say that propositions are logically equivalent if they have logically equivalent forms.) We use \equiv to indicate that two forms are logically equivalent. For example, similar to the equivalence demonstrated above, it is true that $\sim(p \vee q) \equiv \sim p \wedge \sim q$. These two equivalences are called DeMorgan’s laws, after Augustus DeMorgan.

logically equivalent

It is important to remember that the operators \vee and \wedge flip when they are negated. For example, take the sentence

x is even and prime.

We do not call this a proposition, because x is an unknown, but by supplying a value for x (taking \mathbb{N} as the universal set) we would make it a proposition. The set of values that make this a true proposition is $\{2\}$. The set of values that make this proposition false needs to be the complement of that set—that is, the set of all natural numbers besides 2. It may be tempting to negate the not-quite-a-proposition as

x is not even and not prime.

But this is wrong. The set of values that makes this a true proposition is the set of all numbers except evens and primes—a much different set from what is required. The correct negation is

x is not even or not prime.

A form that is logically equivalent with the constant value T (something always true, no matter what the assignments are to the variables) is called a *tautology*. A form that is logically equivalent to F (something necessarily false) is called a *contradiction*. Obviously all tautologies are logically equivalent to each other, and similarly for contradictions. The following truth table explores some tautologies and contradictions.

tautology

contradiction

p	$\sim p$	$p \vee \sim p$	$p \wedge \sim p$	$p \wedge T$	$p \vee T$	$p \wedge F$	$p \vee F$
T	F	T	F	T	T	F	T
F	T	T	F	F	T	F	F

Hence $p \vee \sim p$ and $p \vee T$ are tautologies, $p \wedge \sim p$ and $p \wedge F$ are contradictions, $p \wedge T \equiv p$, and $p \vee F \equiv p$.

The following Theorem displays common logical equivalences.

Theorem 5.1 (Logical equivalences.) *Given logical variables p , q , and r , the following equivalences hold.*

Commutative laws:	$p \wedge q \equiv q \wedge p$	$p \vee q \equiv q \vee p$
Associative laws:	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
Distributive laws:	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
Absorption laws:	$p \wedge (p \vee q) \equiv p$	$p \vee (p \wedge q) \equiv p$
Idempotent laws:	$p \wedge p \equiv p$	$p \vee p \equiv p$
Double negative law:	$\sim \sim p \equiv p$	
DeMorgan's laws:	$\sim (p \wedge q) \equiv \sim p \vee \sim q$	$\sim (p \vee q) \equiv \sim p \wedge \sim q$
Negation laws:	$p \vee \sim p \equiv T$	$p \wedge \sim p \equiv F$
Universal bound laws:	$p \vee T \equiv T$	$p \wedge F \equiv F$
Identity laws:	$p \wedge T \equiv p$	$p \vee F \equiv p$
Tautology and contradiction laws:	$\sim T \equiv F$	$\sim F \equiv T$

These can be verified using truth tables. They also can be used to prove other equivalences without using truth tables by means of a step-by-step reduction to a simpler form. For example, $q \wedge (p \vee T) \wedge (p \vee \sim (\sim p \vee \sim q))$ is equivalent to $p \wedge q$:

$$\begin{aligned}
 & q \wedge (p \vee T) \wedge (p \vee \sim (\sim p \vee \sim q)) \\
 \equiv & q \wedge T \wedge (p \vee \sim (\sim p \vee \sim q)) && \text{by universal bounds} \\
 \equiv & q \wedge (p \vee \sim (\sim p \vee \sim q)) && \text{by identity} \\
 \equiv & q \wedge (p \vee \sim \sim (p \wedge q)) && \text{by DeMorgan's} \\
 \equiv & q \wedge (p \vee (p \wedge q)) && \text{by double negative} \\
 \equiv & q \wedge p && \text{by absorption} \\
 \equiv & p \wedge q && \text{by commutativity.}
 \end{aligned}$$

Exercises

Determine which of the following are propositions.

1. Spinach is on sale.
2. Spinach on sale.
3. $3 > 5$.
4. If $3 > 5$, then Spinach is on sale.
5. Why is Spinach on sale?

Let s stand for “spinach is on sale” and k stand for “kale is on sale.” Write the following using logical symbols.

6. Kale is on sale, but spinach is not on sale.
7. Either kale is on sale and spinach is not on sale or kale and spinach are both on sale.
8. Kale is on sale, but spinach and kale are not both on sale.
9. Spinach is on sale and spinach is not on sale.

Verify the following equivalences using a truth table. Then verify them using ML (that is, type in the left and right sides for all

four possible assignments to q and p , checking that they agree each time).

10. $\sim (p \wedge q) \equiv \sim p \vee \sim q$.
11. $p \wedge (p \vee q) \equiv p$.

Verify the following equivalences using a truth table.

12. $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$.
13. $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$.

Verify the following equivalences by applying known equivalences from Theorem 5.1.

14. $\sim (\sim p \vee (\sim p \wedge \sim q)) \vee \sim p \equiv T$.
15. $p \wedge (\sim q \vee (p \wedge \sim p)) \equiv p \wedge \sim q$.
16. $(q \wedge p) \vee \sim (p \vee \sim q) \equiv q$.
17. $((q \wedge (p \wedge (p \vee q))) \vee (q \wedge \sim p)) \wedge \sim q \equiv F$.
18. $\sim (\sim (p \wedge p) \vee (\sim q \wedge T))$.

Chapter 6

Conditionals

6.1 Conditional propositions

In the previous chapter, we took intuitive notions of *and*, *or*, and *not* and considered them as mathematical operators. However, our initial example

If it is Wednesday or spinach is on sale, then I go to the store.

or, with the details abstracted by variables,

If p or q , then r .

has more than just an *or* indicating its logical form. We also have the words *if* and *then*, which together knit “ p or q ” and “ r ” into a logical form. Let us simplify this a bit to

If spinach is on sale, then I go to the store.

which has the form

If p , then q .

A proposition in this form is called a *conditional*, and is symbolized by the operator \rightarrow . The symbolism $p \rightarrow q$ reads “if p then q ” or “ p implies q .” p is called the *hypothesis* and q is called the *conclusion*.

conditional
hypothesis
conclusion

To define this operator formally, consider the various scenarios for the truth of the hypothesis and conclusion and how they affect the truth of the conditional proposition.

Is spinach on sale?	Do I go to the store?	Is it true that if spinach is on sale, I go to the store?
Yes.	Yes.	It would be consistent with the evidence, but not proved absolutely
Yes.	No.	No, it cannot be true because spinach is on sale and I do not go to the store.
No.	Yes.	Not enough information—the proposition does not seem to address the situation when spinach is not on sale.
No.	No.	Similar to above.

From this we see that we need to be more precise about what is intended when we say that a conditional proposition is true. In only one case were we able to say that it is definitely false—in the

vacuously true

other three cases, the hypothesis and conclusion do not disprove the conditional, but they do not prove it either. To clarify this, we say that a conditional proposition is true if the truth values of the hypothesis and conclusion are *consistent* with the proposition being true. This means the cases where the hypothesis is false are both true, by default. (We call this being *vacuously true*). Thus we have this truth table for \rightarrow :

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

We can further use a truth table to show, for example, that $p \rightarrow q \equiv q \vee \sim(p \vee q)$.

p	q	$p \vee q$	$\sim(p \vee q)$	$q \vee \sim(p \vee q)$	$p \rightarrow q$
T	T	T	F	T	T
T	F	T	F	F	F
F	T	T	F	T	T
F	F	F	T	T	T

In other words, “If spinach is on sale, then I go to the store” is equivalent to “I go to the store or it is not true that either spinach is on sale or I go to the store.”

6.2 Negation of a conditional

Negating a conditional is tricky. Obviously one can simply put a “not” in front of the whole proposition ($\sim(p \rightarrow q)$ or “it is not true that if spinach is on sale, then I go to the store”), but we would like something more useful, a way to evaluate such a negation to get a simpler, equivalent proposition—a simplification rule like De Morgan’s laws.

The negation of a proposition must be true exactly when the proposition is not true. In other words, it must have the opposite truth value. What has opposite truth to “If spinach is on sale, then I go to the store”? Consider these attempts:

If spinach is not on sale, then I go to the store. This is not right because it does not adequately address the situation where one goes to the store every day, whether spinach is on sale or not. In that case, both this and the original proposition would be true, so this is not a negation.

If spinach is not on sale, I do not go to the store. Merely propagating the negation to hypothesis and conclusion does not work at all. If spinach is on sale and I go, or spinach is not on sale and I do not go, both this and the original proposition hold.

If spinach is on sale, I do not go to the store. This attempt is perhaps the most attractive, because it does indeed contradict the original proposition. However, it can be considered “too strong” and so not a negation—both it and the original proposition are vacuously true if spinach is not on sale.

To find a true negation, use a truth table to identify the truth values for $\sim(p \rightarrow q)$; then we will try to construct a simple form equivalent to it.

p	q	$p \rightarrow q$	$\sim(p \rightarrow q)$
T	T	T	F
T	F	F	T
F	T	T	F
F	F	T	F

That is, we are looking for a proposition that is true only when both p is true and q is not true. Thus we have $\sim(p \rightarrow q) \equiv p \wedge \sim q$.

This is a surprising result. The negation of a conditional is not itself a conditional. The negation of “If spinach is on sale, then I go to the store” is “Spinach is on sale and I do not go to the store.”

6.3 Converse, inverse, and contrapositive

There are three common forms that are variations on the conditional. Switching the order of the hypothesis and the conclusion, $q \rightarrow p$, is the *converse* of $p \rightarrow q$.

converse

If I go to the store, then spinach is on sale.

The converse is not logically equivalent to the proposition.

p	q	$p \rightarrow q$	$q \rightarrow p$
T	T	T	T
T	F	F	T
F	T	T	F
F	F	T	T

Many common errors in reasoning come down to a failure to recognize this. p being correlated to q is not the same thing as q being correlated to p . I may go to the store every time spinach is on sale, but that does not mean that I will never go to the store if spinach is not on sale, and so my going to the store does not imply that spinach is on sale.

The *inverse* is formed by negating each of the hypothesis and conclusion separately (*not* negating the entire conditional), $\sim p \rightarrow \sim q$.

inverse

If I do not go to the store, then spinach is not on sale.

For the same reason as above, the inverse is not logically equivalent to the proposition either.

p	q	$p \rightarrow q$	$\sim p \rightarrow \sim q$
T	T	T	T
T	F	F	T
F	T	T	F
F	F	T	T

The *contrapositive* is formed by negating and switching the components of a conditional, $\sim q \rightarrow \sim p$.

contrapositive

If I do not go to the store, then spinach is not on sale.

The contrapositive is logically equivalent to the proposition.

p	q	$p \rightarrow q$	$\sim q \rightarrow \sim p$
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	T

Compare the truth tables of the converse and the inverse. Notice that they are logically equivalent. In fact, the converse and inverse are contrapositives of each other.

6.4 Writing conditionals in English

There is a variety of phrasings in mathematical parlance alone (let alone everyday speech) that express the idea of conditionals and their variations. For example, think about what is meant by

$x > 7$ only if x is even.

This certainly does not say that if x is even, then $x > 7$, but rather leaves open the possibility that x is even although it is 7 or less. No, this means if $x > 7$, then x is even. In other words, *only if* is a way of saying a converse proposition. p only if q means $q \rightarrow p$.

Along the same lines, the phrase “if and only if,” often abbreviated “iff,” is of immense use in mathematics. If we expand

$x \bmod 2 = 0$ iff x is even.

we get

If $x \bmod 2 = 0$, then x is even, and if x is even, then $x \bmod 2 = 0$.

That is, we get both the conditional and its converse. The connector *iff* is sometimes called the *biconditional*, is symbolized by a double arrow, $p \leftrightarrow q$, and is defined to be equivalent to $(p \rightarrow q) \wedge (q \rightarrow p)$.

biconditional

p	q	$p \rightarrow q$	$q \rightarrow p$	$p \leftrightarrow q$
T	T	T	T	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	T

necessary conditions
sufficient conditions

We also sometimes speak of *necessary conditions* and *sufficient conditions*, which refer to converse conditional and conditional propositions, respectively.

An even degree is a necessary condition for a polynomial to have no real roots
means

If a polynomial function has no real roots, then it has an even degree.

A positive global minimum is a sufficient condition for a polynomial to have no real roots
means

If a polynomial function has a positive global minimum, then it has no real roots.

Values all of the same sign is a necessary and sufficient condition for a polynomial to have no real roots.

means

A polynomial function has values all of the same sign if and only if the function has no real roots.

6.5 Conditional expressions in ML

ML does not have an operator that performs logical conditionals, because that would be rarely used. However, it is appropriate now to introduce functionality it has to make decisions based on the truth or falsity of an expression.

A step function is a function over real numbers whose value is zero up until some number, after which it is one, for example,

$$f(x) = \begin{cases} 0 & \text{if } x < 5 \\ 1 & \text{otherwise} \end{cases}$$

Step functions have applications in electronics and statistics. Notice that computing them requires making a decision, and that decision is based on evaluating a statement: *is x less than 5 or not?* We can express this in ML using the form

if <expression>₁ then <expression>₂ else <expression>₃

The first expression is called the condition. The second two are called the then-clause and else-clause, respectively. The condition must have type `bool`. The other expressions must have the same type, but that type can be anything. If the condition is true, then the value of the entire expression is the value of the then-clause; if it is false, then the entire expression's value is that of the else-clause. Thus the type of the second two expressions is also the type of the entire expression.

```
- val x = 3;

val x = 3 : int

- if x < 5 then 0 else 1;

val it = 0 : int

- datatype mammal = Dolphin | Monkey | Hog;

datatype mammal = Dolphin | Hog | Monkey

- datatype collective = Pod | Troop | Herd;

datatype collective = Herd | Pod | Troop

- val a = Hog;

val a = Hog : mammal

- if a = Dolphin then Pod else if a = Monkey then Troop else Herd;

val it = Herd : collective

- if a = Dolphin then 12 else Pod;

stdIn:1.1-2.21 Error: types of rules don't agree [literal]
  earlier rule(s): bool -> int
  this rule: bool -> collective
in rule:
  false => Pod
```

At this point, the number of interesting things for which we can use this are limited, but it is important to understand how types fit together in an expression like this for later use.

Exercises

1. Use a truth table to show that $(p \vee q) \rightarrow r \equiv \sim r \rightarrow (\sim p \wedge \sim q)$.
2. The equivalence $p \rightarrow q \equiv \sim q \rightarrow \sim p$ is called the contrapositive law. Use this and known equivalences from Theorem 5.1 to show that $(p \vee q) \rightarrow r \equiv \sim r \rightarrow (\sim p \wedge \sim q)$.

Let r stand for “It is raining,” s stand for “It is snowing,” and p stand for “spinach is on sale.” Write the following using logical symbols.

3. If it is not snowing, then spinach is on sale.
4. If spinach is on sale and it is raining, then it is not snowing.
5. If it is raining, then spinach is not on sale or it is snowing.
6. If spinach is on sale, then if it is not raining, then it is snowing.
7. In Illinois, the drinking age is 21. Each of the following cards gives information about one of the four people at a table in a certain restaurant. On each card, one side gives the person’s age and the other side indicates what that person is drinking. The manager claims, “If a person is drinking beer, then that person is at least 21.” Which card or cards would you need to turn over to determine whether the claim is false? Explain.

a.	b.	c.	d.
Beer	Coke	20	24

A person named Bob passed through the following maze without using the same door twice.

T	S	R	Q	P
K	L	M	N	O
J	I	H	G	F
E	D	C	B	A

In Exercises 8–13, indicate if the sentence is true or false or if you cannot tell.

8. Bob passed through P .
9. Bob passed through N .
10. Bob passed through M .
11. If Bob passed through O , then Bob passed through F .
12. If Bob passed through K , then Bob passed through L .
13. If Bob passed through L , then Bob passed through K . (Be careful.)
14. Write the negation, converse, inverse, and contrapositive (all in English) of the statement in exercise 3.
15. Write the negation, converse, inverse, and contrapositive of the statement in exercise 4.
16. Write the negation, converse, inverse, and contrapositive of the statement in exercise 5.
17. Write the negation, converse, inverse, and contrapositive of the statement in exercise 6.
18. Suppose you wanted to divide x by some value y , but only if y was non-zero; you would like the result simply to be x otherwise. Write an ML expression for this calculation, but let x appear in the expression only once.
19. Find a form that will always produce the same result (that is, is equivalent to)

if <boolean expression>₁ then true else <boolean expression>₂

If you do not see it immediately, experiment on some specific expressions.

Exercises 7 and 8–13 were presented by Susanna Epp at a workshop in June 2006.

Chapter 7

Argument forms

7.1 Arguments

So far we have considered symbolic logic on the proposition level. We have considered the logical connectives that can be used to knit propositions together into new propositions, and how to evaluate propositions based on the value of their component propositions. However, to put this to use—either for writing mathematical proofs or engaging in any other sort of discourse—we need to work in units larger than propositions. For example,

During the full moon, spinach is on sale.
The moon is full.
Therefore, spinach is on sale.

contains several propositions, and they are not connected to become a single proposition. This, instead, is an *argument*, a sequence of propositions, with the last proposition beginning with the word “therefore”—or “so” or “hence” or some other such word, and possibly in a postpositive position, as in “Spinach, therefore, is on sale.” Similarly, an *argument form* is a sequence of proposition forms, with the last prefixed by the symbol \therefore . All except the last in the sequence are called *premises*; the last proposition is called the *conclusion*.

argument

argument form

premises

conclusion

valid

Propositions are true or false. Arguments, however, are not spoken of as being true or false; instead, they are *valid* or invalid. We say that an argument form is valid if, whenever all the premises are true (depending on the truth values of their variables), the conclusion is also true.

Consider another argument:

If my crystal wards off alligators, then there will be no alligators around.
There are no alligators around.
Therefore, my crystal wards off alligators.

The previous argument and this one have the following argument forms, respectively (rephrasing “During the full moon...” as “If the moon is full, then...”):

$p \rightarrow q$
 p
 $\therefore q$

$p \rightarrow q$
 q
 $\therefore p$

It is to be hoped that you readily identify the left argument form as valid and the right as invalid. The truth table verifies this.

p	q	$p \rightarrow q$	q		p	q	$p \rightarrow q$	p	
T	T	T	T	\leftarrow critical row	T	T	T	T	\leftarrow critical row
T	F	F	F		T	F	F	T	
F	T	T	T		F	T	T	F	\leftarrow critical row
F	F	T	F		F	F	T	F	

critical rows

The first argument has only one case where both premises are true, and we see there that the conclusion is also true. The rest of the truth table does not matter—only the rows where all premises are true count. We call these *critical rows*, and when you are evaluating large argument forms, it is acceptable to leave the entries in non-critical rows blank. Moreover, once you have found a critical row where the conclusion is false, nothing more needs to be done. The second argument has a critical row where the conclusion is false; hence it is an invalid argument.

7.2 Common syllogisms

syllogism

modus ponens

A *syllogism* is a short argument. Usually the definition restricts the term to apply only to arguments having exactly two premises, but we will include one- and three-premised arguments as well in our discussion. The correct argument form above is the most famous, and is called *modus ponens*, Latin for “method of affirming,” or, more literally, “method of placing” (or, more literally still, “placing method,” since *ponens* is a participle, not a gerund).

$p \rightarrow q$	If Socrates is a man, then he is mortal.
p	Socrates is a man.
$\therefore q$	Therefore, Socrates is mortal.

Since the contrapositive of a conditional is logically equivalent to the conditional itself, a truth table from the previous chapter proves

$$p \rightarrow q \\ \therefore \sim q \rightarrow \sim p$$

If we substitute “ $\sim q$ ” for “ p ” and “ $\sim p$ ” for “ q ” in modus ponens, we get

$$\sim q \rightarrow \sim p \\ \sim q \\ \therefore \sim p$$

modus tollens

Putting these two together results in the second most famous syllogism, *modus tollens*, “lifting [i.e., denying] method.”

$p \rightarrow q$	If the moon is full, then Lupin will be a werewolf.
$\sim q$	Lupin is not a werewolf.
$\therefore \sim p$	Therefore, the moon is not full.

We can also prove this directly with a truth table, with only the critical row completed for the conclusion column.

p	q	$p \rightarrow q$	$\sim q$	$\sim p$
T	T	T	F	
T	F	F	T	
F	T	T	F	
F	F	T	T	T

The form *generalization* may seem trivial and useless, but, in fact, it captures a reasoning technique we often use subconsciously. It relies on the fact that a true proposition *or*'ed to any other proposition is still a true proposition.

generalization

p	We are in Pittsburgh.
$\therefore p \vee q$	Therefore, we are in Pittsburgh or Mozart wrote <i>The Nutcracker</i> .

A similar (though symmetric) argument form using *and* is called *specialization*.

specialization

$p \wedge q$	$x > 5$ and x is even.
$\therefore p$	Therefore, $x > 5$.

Sherlock Holmes describes the process of elimination as, "...when you have eliminated the impossible, whatever remains, however improbable, must be the truth." Formally, *elimination* is

elimination

$p \vee q$	x is even or x is odd.
$\sim p$	x is not even.
$\therefore q$	Therefore, x is odd.

We will later prove that, given sets A , B , and C , if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$. This means that the \subseteq relation is transitive. The analogous logical form *transitivity* is

transitivity

$p \rightarrow q$	If $x > 5$, then $x > 3$.
$q \rightarrow r$	If $x > 3$, then $x > 0$.
$\therefore p \rightarrow r$	Therefore, if $x > 5$, then $x > 0$.

Sometimes a phenomenon has two possible causes (or, at least, two things that are correlated to it). Then all that needs to be shown is that at least one such cause is true. Or, if we know that at least one of two possibilities is true and that they each imply a fact, that fact is true. We call this form *division into cases*.

division into cases

$p \vee q$	It's either snowing or raining.
$p \rightarrow r$	If it's snowing, then spinach is on sale.
$q \rightarrow r$	If it's raining, then spinach is on sale.
$\therefore r$	Therefore, spinach is on sale.

Finally, we have proof by *contradiction*.

contradiction

$p \rightarrow F$	If $x - 2 = x$ then $-2 = 0$ and $-2 \neq 0$.
$\therefore \sim p$	Therefore, $x - 2 \neq x$.

Because of certain paradoxes that have arisen in the study of the foundations of mathematics, some logicians call into question the validity of proof by contradiction. If p leads to a contradiction, it might not be that p is false; it could be that p is neither true nor false, that is, p might not be a proposition at all. For our purposes, however, we can rely on proof by contradiction.

7.3 Using argument forms for deduction

Non-trivial argument forms would require huge truth tables to verify. However, we can use known argument forms to verify larger argument forms in a step-wise fashion. Take the argument form

- (a) $\sim p \wedge \sim r \rightarrow s$
- (b) $p \rightarrow \sim q$
- (c) $\sim t$
- (d) $t \vee \sim s$
- (e) $r \rightarrow \sim q$
- (f) $\therefore \sim q$

This does not follow immediately from the argument forms we have given. However, we can deduce immediately that $\sim s$ by applying division into cases to (c) and (d). Our goal is to generate new propositions from known argument forms until we have verified proposition (f).

- | | | |
|-------|-------------------------------|--|
| (i) | $\sim s$ | by (c), (d), and division into cases. |
| (ii) | $\sim (\sim p \wedge \sim r)$ | by (a), (i), and modus tollens |
| (iii) | $p \vee r$ | by (ii) and DeMorgan's laws |
| (iv) | $\sim q$ | by (iii), (b), (c), and division into cases. |

Notice that using logical equivalences from Theorem 5.1 is fair game.

Exercises

Verify the following syllogisms using truth tables.

1. Generalization.
2. Specialization.
3. Elimination.
4. Transitivity.
5. Division into cases.
6. Contradiction.

Use known syllogisms and logical equivalences to verify the following arguments.

7. (a) $p \rightarrow q$
 (b) $r \vee s$
 (c) $r \rightarrow t$
 (d) $\sim q$
 (e) $u \rightarrow v$
 (f) $s \rightarrow p$
 (g) $\therefore t$
8. (a) $\sim p \vee q \rightarrow r$
 (b) $s \vee \sim q$
 (c) $\sim t$
 (d) $p \rightarrow t$
 (e) $\sim p \wedge r \rightarrow \sim s$
 (f) $\therefore \sim q$

9. (a) $p \vee q$
 (b) $q \rightarrow r$
 (c) $p \wedge s \rightarrow t$
 (d) $\sim r$
 (e) $\sim q \rightarrow u \wedge s$
 (f) $\therefore t$
10. (a) $\sim p \rightarrow r \wedge \sim s$
 (b) $t \rightarrow s$
 (c) $u \rightarrow \sim p$
 (d) $\sim w$
 (e) $u \vee w$
 (f) $\therefore \sim t$
11. (a) $p \rightarrow q$
 (b) $r \vee s$
 (c) $\sim s \rightarrow \sim t$
 (d) $\sim q \vee s$
 (e) $\sim s$
 (f) $\sim p \wedge r \rightarrow u$
 (g) $w \vee t$
 (h) $\therefore u \wedge w$

Exercises 7–11 are taken from Epp [5].

Chapter 8

Predicates and quantifiers

Recall the most famous syllogism, *modus ponens*,

If Socrates is a man, then he is mortal.
Socrates is a man.
Therefore, Socrates is mortal.

It is unlikely that we would either presume or prove such a narrow premise as “If Socrates is a man, then he is mortal.” What is so special about Socrates that this conditional mortality accrues to him? Rather, we would be more likely to say

All men are mortal.
Socrates is a man.
Therefore, Socrates is mortal.

Our premise, “All men are mortal,” now addresses a wider scope, and our syllogism merely applies this universal truth to a specific case. However, we have not yet discussed how to deal with concepts like “all” in formal logic. Is it necessary to expand our notation (and reasoning rules), or can this be captured by the logical forms we already know? For example, could we not express the first premise using a conditional?

If someone is a man, then he is mortal.

This is equivalent, but now we have introduced the pronouns “someone” and “he,” which is English’s way of referring to the same but unknown value. Therefore we could simplistically replace the pronouns with pseudo-mathematical notation to get

If x is a man, then x is mortal.¹

However, variables (and pronouns with uncertain antecedents) mean that a sentence is no longer a proposition. In this chapter, we will see the use of unknowns in formal reasoning.

8.1 Predication

When we introduced conditionals, we moved from the specific case to the general case by replacing parts of a conditional sentence with variables.

If Socrates is a man, then he is mortal.

If p then q .

These variables are *parameters*, that is, independent variables that can be supplied with any

parameters

¹This is not quite right; we still need to say that this is true “for all x .”

values from a certain set (in this case, the set of propositions) and that affect the value of the entire expression (in this case, also a proposition, once the values have been supplied). When we replace parts of a mathematical expression with independent variables, we are parameterizing that expression. We see the same process here:

Socrates is a man

x is a man.

predicate

This makes a proposition with a hole in it. A sentence that is a proposition but for an independent variable is a *predicate*. This is the same term *predicate* that you remember from grammar school; a predicate is the part of a clause that expresses something about the subject.



If we want to represent a given predicate with a symbol, it would be useful to incorporate the parameter. Hence we can define, for example,

$$P(x) = x \text{ is mortal}$$

domain

You should recognize this notation as being the same as that used for functions in algebra and calculus. We will study functions formally in Part VI, but you can use what you remember from prior math courses to recognize that a predicate is alternately defined as a function whose codomain is the set of truth values, $\{\text{true}, \text{false}\}$. In fact, another term for *predicate* is *propositional function*. The *domain* of a predicate is the set of values that may be substituted in place of the independent variable.

To play with the two sentences above, let

$P(x) = x \text{ hit the ball.}$

$Q(x) = x \text{ is green.}$

And so we can note $P(\text{the boy})$, $P(\text{the bat})$, $Q(\text{spinach})$, $Q(\text{Kermit})$, and $\sim Q(\text{ruby})$. It is important to note that the domain of a predicate is not just those things that make it true, but rather all things it would make sense to talk about in that context (here we can assume something like “visible objects and substances”). It is not invalid to say $Q(\text{ruby})$ or “ruby is green.” It merely happens to be false.

Here is a mathematical example. Let $P(x) = x^2 > x$. What is $P(x)$ for various values of x , if we assume \mathbb{R} as the domain?

x	5	2	1	$\frac{1}{2}$	0	$-\frac{1}{2}$	-1
$P(x)$	T	T	F	F	F	T	T

truth set

It is not too difficult to characterize the numbers that make this predicate true—positive numbers greater than or equal to one, and all negative numbers. The *truth set* of a predicate $P(x)$ with domain D is the set of all elements in D that make $P(x)$ true when substituted for x . We can denote this using set notation as $\{x \in D | P(x)\}$. In this case,

$$\{x \in \mathbb{R} | P(x)\} = (-\infty, 0) \cup [1, \infty)$$

8.2 Universal quantification

Now we return to the original question. How can we express “All men are mortal” formally? It does not do to define a predicate

$$P(x) = \text{if } x \text{ is a man, then } x \text{ is mortal.}$$

because “All men are mortal” is a proposition, no mere predicate. It truly does make a claim that is either true or false. In other words, the predicate does not capture the assertion being made about *all* men. The problem is that the variable x is not truly free, but rather we want to remark on the extent of x , the values for which we are asserting this predicate. Words that express this are called *quantifiers*. Here is a rephrasing of “all men are mortal” that uses variables correctly:

quantifiers

For all men x , x is mortal.

The symbol \forall stands for “for all.” Then, if we let \mathbb{M} stand for the set of all men and $M(x) = x$ is mortal, we have

$$\forall x \in \mathbb{M}, M(x)$$

\forall is called the *universal quantifier*. Likewise, a *universal proposition* is a proposition in the form $\forall x \in D, P(x)$, where $P(x)$ is a predicate and D is the domain (or a subset of the domain) of $P(x)$. Unfortunately, defining the meaning of a universal proposition cannot be done simply with a truth table. Instead we say, almost banally, that the proposition is true if $P(x)$ is true for every element in the domain. For example, let $D = \{3, 54, 219, 318, 471\}$. Which of the following are true?

universal quantifier

universal proposition

$$\forall x \in D, x^2 \leq x$$

No, this is actually false for all of them.

$$\forall x \in D, x \text{ is even}$$

No, this fails for 3.

$$\forall x \in D, x \text{ is a multiple of 3}$$

Yes: $3 = 3 \cdot 1, 54 = 3 \cdot 18, 219 = 3 \cdot 73, 318 = 3 \cdot 106$, and $471 = 3 \cdot 157$.

What we used on the last proposition was the *method of exhaustion*, that is, we tried all possible values for x until we exhausted the domain, demonstrating each one made the predicate true. Obviously this method of proof is possible only with finite sets, and it is impractical for any set much larger than the one in this example. On the other hand, *disproving* a universal proposition is quite easy, since it takes only one hole to sink the ship. If for any element of D , $P(x)$ is false, then the entire proposition is false. Having found 3, not an even number, we disproved the second proposition, without even noting that the predicate fails also for 219 and 471. An element of the domain for which the predicate is false is called a *counterexample*.

method of exhaustion

counterexample

8.3 Existential quantification

It is not true that

$$x^2 = 16 \text{ for all } x \in \mathbb{R}.$$

It is true, however, that

$$x^2 = 16 \text{ for some } x \in \mathbb{R}.$$

namely, for 4 and -4 . While it is true that

$$\sim (\forall x \in \mathbb{R}, x^2 = 16)$$

it is not true that

$$\forall x \in \mathbb{R}, \sim (x^2 = 16)$$

The word “some” expresses the situation that falls between being true for all and being true for none—in other words, the predicate is true for at least one, perhaps more. It is an *existential quantifier*, because it asserts that at least one thing exists with the given predicate as a property. We can rephrase the second proposition of this section to get

existential quantifier

There exists an $x \in \mathbb{R}$ such that $x^2 = 16$.

Now, the symbol \exists means “there exists.” Hence we have

$$\exists x \in \mathbb{R} \mid x^2 = 16$$

Formally, an existential proposition is a proposition of the form $\exists x \in D \mid P(x)$ for some predicate $P(x)$ with domain (or domain subset) D .

Revisiting our earlier example with $D = \{3, 54, 219, 318, 471\}$, which of the following are true?

$\exists x \in D, x^2 \leq x$	No, this is false for all of them.
$\exists x \in D, x$ is even	Yes, $54 = 2 \cdot 27$.
$\exists x \in D, x$ is a multiple of 3	Yes: $3 = 3 \cdot 1$.

Notice that with existentially quantified propositions, we must use the method of exhaustion to *disprove* it. Only one specimen is needed to show that it is true.

8.4 Implicit quantification

The most difficult part of getting used to quantified propositions is taking note of the various forms in which they appear in English and various mathematical forms that are equivalent. For example, \forall can just as easily stand in for “for any,” “for every,” “for each,” and “given any.” Consider the proposition

A positive integer evenly divides 121.

While this is awkward and ambiguous (is this supposed to define the term *positive integer*?), it can be read to mean “there exists a positive integer that evenly divides 121,” or,

$$\exists x \in \mathbb{Z}^+ \mid x \text{ divides } 121$$

which is true, letting $x = 11$. The existential quantification is implicit, hanging on the indefinite article *a*. However, the indefinite article can also imply universal quantification, depending on the context. Hence

If a number is a rational number, then it is a real number.

becomes

$$\forall x \in \mathbb{Q}, x \in \mathbb{R}$$

Notice also that this is equivalent to

$$\forall x \in U, x \in \mathbb{Q} \rightarrow x \in \mathbb{R}$$

where we take the universal set to be all numbers. More generally,

$$\forall x \in D, Q(x) \quad \equiv \quad \forall x \in U, x \in D \rightarrow Q(x)$$

Quantification in natural language is subtle and a frequent source of ambiguity for the careless (though the intended meaning is usually clear from context or voice inflection). Adverbs (besides *not*) usually do not affect the logical meaning of a sentence, but notice how *just* turns

I wouldn’t give that talk to any audience.

into

I wouldn’t give that talk to just any audience.

8.5 Negation of quantified propositions

Finally, we consider how to negate propositions with universal or existential quantifiers. We saw earlier that

$$\sim (\forall x \in \mathbb{R}, x^2 = 16) \not\equiv \forall x \in \mathbb{R}, \sim (x^2 = 16)$$

So negation is not a simple matter of propagating the negation symbol through the quantifier. What, then, is the negation of

All men are mortal.

There exists a bag of spinach that is on sale.

What will help us here is to think about what the quantifiers are actually saying. If a predicate is true for all elements in the set, then if we could order those elements, it would be true for the first one, and the next one, and one after that. In other words, we can think of universal quantification as an extension of conjunction. If there merely exists an element for which the predicate is true, then it is true for the first, or the next one, or one of the elements after that. Hence if $D = \{d_1, d_2, d_3, \dots\}$,

$$\forall x \in D, P(x) \quad \equiv \quad P(d_1) \wedge P(d_2) \wedge P(d_3) \dots$$

and

$$\exists x \in D \mid P(x) \quad \equiv \quad P(d_1) \vee P(d_2) \vee P(d_3) \dots$$

Now, we can apply an extended version of DeMorgan's laws.

$$\sim (\forall x \in D, P(x)) \quad \equiv \quad \sim (P(d_1) \wedge P(d_2) \wedge P(d_3) \dots) \quad \equiv \quad \sim P(d_1) \vee \sim P(d_2) \vee \sim P(d_3) \dots \quad \equiv \quad \exists x \in D, \sim P(x)$$

and

$$\sim (\exists x \in D \mid P(x)) \quad \equiv \quad \sim (P(d_1) \vee P(d_2) \vee P(d_3) \dots) \quad \equiv \quad \sim P(d_1) \wedge \sim P(d_2) \wedge \sim P(d_3) \dots \quad \equiv \quad \forall x \in D, \sim P(x)$$

Hence the negation of a universal proposition is an existential proposition, and the negation of an existential proposition is a universal proposition. To negate our examples above, we would say

There exists a man who is not mortal.

All bags of spinach are not on sale.

Exercises

Let S be the set of bags of spinach, $g(x)$ be the predicate where x is green, and $s(x)$ be the predicate where x is on sale. Write the following symbolically, then negate them, then express the negations in English.

1. All bags of spinach are on sale.
2. Some bags of spinach are on sale.
3. All bags of spinach that are not green are on sale.
4. Every bag of spinach is green.
5. Some bags of spinach are not green.
6. Any bag of spinach is green and on sale.

Chapter 9

Multiple quantification; representing predicates

In this chapter, we have two separate concerns, both of which extend the concepts of predication from the previous chapter. First, we consider how to interpret propositions that are *multiply quantified*, that is, have two (or more) quantifiers. Second, we consider how to represent predicates in ML.

9.1 Multiple quantification

Soon we will take up the game of writing mathematical proofs. The most important consideration for determining a strategy in a given proof is how the proposition to be proved is quantified. You will be trained to think, “What would it take to prove to someone that this proposition is true?” or, equivalently, “How would a skeptic *challenge* this proposition?” A proof is a ready-made answer to a possible challenge.

Suppose we have the (fairly banal) proposition

There exists an integer greater than 5.

Written symbolically, this is $\exists x \in \mathbb{Z} \mid x > 5$. A doubter would have to challenge this by saying “So, you think there is a number greater than five, do you? Well then, show me one.” To satisfy this doubter, your response would be to name any number that is both an integer and greater than 5—6 will do fine. On the other hand, suppose you were asserting that

Any integer is a rational.

or, $\forall x \in \mathbb{Z}, x \in \mathbb{Q}$. It would be unfair for a challenger to ask you to verify this proposition for every element in \mathbb{Z} . Even the most hardened skeptic would not have the patience to hear you enumerate an infinite number of possibilities. However, this does demonstrate the futility of *proving by example*:

Doubter: Prove to me that any integer is rational.

Prover: Well, 0 is an integer and it is rational.

Doubter: Ok, one down, infinitely many to go.

Prover: Also, 1 is an integer and it is rational.

...

Prover: And furthermore, 1,405,341 is an integer and it is rational.

Doubter: Ok, 1,405,342 down, infinitely many to go. And you haven’t even begun to talk about negative integers.

What makes proving by example unconvincing is that *the prover is picking the test cases*. What if he is picking only the ones on which the proposition seems to be true, and ignoring the counterexamples? It is like a huckster picking his own shills out of the crowd on whom to demonstrate

his snake oil. Instead, to make the game both *fair* and *finite*, it must be the *doubter* who picks the sample from the domain and challenges the prover to demonstrate the predicate is true for it.

Thus an existentially quantified proposition is proven by providing an example, and a universally quantified proposition is proven by providing a way to confirm any given example.

All this serves to build intuition for how to interpret propositions with nested levels of quantification. How would we symbolically represent the proposition

Every integer has an opposite.

To work this out, let us follow the kind of reasoning above in reverse. Suppose you were to prove this. What kind of challenge would you expect? The idea here is that the integer 5 has for its opposite -5, -10 has 10, 0 has 0, and so on. Since you want to show this pattern holds for *every* integer, it makes sense that challenger gets to pick the integer on which to argue. However, once that integer is picked, how is the rest of the game played? You, the prover, must come up with an opposite to match that integer. Hence the game has two steps: the doubter picks an item to challenge you, and you counter that challenge with another item. The two steps correspond to two levels of quantification. First, you are claiming that some predicate is true for all integers, so we have something in the form

$$\forall x \in \mathbb{Z}, P(x)$$

But what is $P(x)$? What are we claiming about all integers? We claim that something exists corresponding to it, namely an opposite. $P(x) = \exists y \in \mathbb{Z} \mid y = -x$. All together,

$$\forall x \in \mathbb{Z}, \exists y \in \mathbb{Z} \mid y = -x$$

multiply quantified

This is a *multiply quantified* proposition, meaning that the predicate of the proposition is itself quantified. This is more specific than that the proposition merely has more than one quantifier. The proposition “Every frog is green, and there exists a brown toad” has more than one quantifier, but this is not what we have in mind by multiple quantification, because one quantified subproposition is not nested within the other.

Do not fail to notice that quantifiers are not commutative. We would have a very different (and false) proposition if we were to say

$$\exists y \in \mathbb{Z} \mid \forall x \in \mathbb{Z}, y = -x$$

or

There is an integer such that every integer is its opposite.

Also notice that the innermost predicate ($y = -x$) has two independent variables. The general form is

$$\forall x \in D, \exists y \in E \mid P(x, y)$$

where P is a two-argument predicate, with arguments of domains D and E , respectively; or, equivalently, P is a single-argument predicate with domain $D \times E$.

Let us try another example. How would you translate to symbols the proposition

There is no greatest prime number.

To make this process easier, let us temporarily ignore the negation.

There is a greatest prime number.

Obviously the outer proposition is existential. Picking \mathbb{P} to stand for the set of prime numbers and writing half-symbolically we have

$$\exists x \in \mathbb{P} \mid x \text{ is the greatest prime number}$$

What does it mean to be the greatest prime number? It means that all other prime numbers must be less than x .

$$\exists x \in \mathbb{P} \mid \forall y \in \mathbb{P}, y \leq x$$

(Why did we say “ \leq ” rather than “ $<$ ”? Now we negate this.

$$\sim \exists x \in \mathbb{P} \mid \forall y \in \mathbb{P}, y \leq x$$

Evaluate the negation of the existential quantifier.

$$\forall x \in \mathbb{P}, \sim \forall y \in \mathbb{P}, y \leq x$$

Evaluate the negation of the universal quantifier.

$$\forall x \in \mathbb{P}, \exists y \in \mathbb{P} \mid y > x$$

or

For every prime, there exists a greater prime.

As a final example, think back to the beginning of calculus when you first encountered the formal definition of a limit. You may remember it as a less than pleasant experience. It is likely that one of the main frustrations was that it is multiply quantified. $\lim_{x \rightarrow a} f(x) = L$ means

$$\forall \epsilon \in \mathbb{R}^+, \exists \delta \in \mathbb{R}^+ \mid 0 < |x - a| < \delta \rightarrow |f(x) - L| < \epsilon$$

9.2 Ambiguous quantification

The use of symbols becomes increasingly critical when multiple quantification is involved because natural language becomes desperately ambiguous at this point. Consider the sentence

There is a professor teaching every class.

This could mean

$$\exists x \in (\text{The set of professors}) \mid \forall y \in (\text{The set of classes}), x \text{ teaches } y$$

A very busy professor indeed. More likely, this is meant to indicate an adequate staffing situation, that is,

$$\forall y \in (\text{The set of classes}), \exists x \in (\text{The set of professors}) \mid x \text{ teaches } y$$

Similarly, when someone says

A man loves every woman.

does it indicate a particularly promiscuous fellow

$$\exists x \in (\text{Men}) \mid \forall y \in (\text{Women}), x \text{ loves } y$$

or a Hollywood ending?

$$\forall y \in (\text{Women}), \exists x \in (\text{Men}) \mid x \text{ loves } y$$

If we say

Every man loves a woman.

do we mean

$$\forall x \in (\text{Men}), \exists y \in (\text{Women}) \mid x \text{ loves } y$$

that every guy has a star after which he pines, or

$$\exists y \in (\text{Women}) \mid \forall x \in (\text{Men}), x \text{ loves } y$$

some gal will be a shoe-in for homecoming queen?

9.3 Predicates in ML

We have already seen how to write boolean expressions in ML—expressions that represent propositions. However, ML understandably rejects an expression containing an undefined variable.

```
- x < 15.3;
```

```
stdIn:1.7 Error: unbound variable or constructor: x
```

However, we can capture the independent variable by giving a name to the expression, that is, defining a predicate. In ML, we define predicates using the form

```
fun <identifier> ( <identifier> ) = <expression>
```

The keyword `fun` is similar to `val` in that it assigns a meaning to an identifier (namely the first identifier), which is the name of the predicate. The second identifier, enclosed in parentheses, is the independent variable.

```
- fun P(x) = x < 15.3;
```

```
val P = fn : real -> bool
```

```
- P(1.5);
```

```
val it = true : bool
```

```
- P(15.3);
```

```
val it = false : bool
```

```
- P(27.4);
```

```
val it = false : bool
```

Let us compose predicates to test if a real number is within the range $[-3.4, 47.3)$ and if an integer is divisible by three.

```
- fun Q(x) = x >= 3.4 andalso x < 47.3;
```

```
val Q = fn : real -> bool
```

```
- Q(16.5);
```

```

val it = true : bool

- Q(0.1);

val it = false : bool

- Q(57.9);

val it = false : bool

- fun R(n) = n mod 3 = 0;

val R = fn : int -> bool

- R(2);

val it = false : bool

- R(4);

val it = false : bool

- R(6);

val it = true : bool

```

Notice several things. First, the keyword `fun` is used because we are defining a function—specifically, a function whose co-domain is `bool`. Next, note ML’s response to the definition of a predicate, for example `val Q = fn : real -> bool`. This means that the variable `Q` is assigned a certain value. Indeed, `Q` is a variable, but not a variable that stores a `int` or `bool` value, but one that stores a function value. The value is not printed, but `fn` (another keyword based on an abbreviation for “function”) stands in for it. The type of the value is `real -> bool`, which essentially means a function whose domain is `real` and whose co-domain is `bool`.

Remember that `*` corresponds to \times in mathematical notation for Cartesian products. In mathematical notation, we would say that a predicate like `Q` is a function $\mathbb{R} \rightarrow \{\text{true}, \text{false}\}$. You should use your knowledge of functions from earlier study in mathematics to understand this for now, but general functions and the concept of function types will be examined more carefully in a later chapter. The function is quite the crucial concept in ML programming.

9.4 Pattern-matching

As a finale for this chapter, we consider writing predicates for non-numerical data. In an earlier chapter, we learned how to create a new datatype. For example, we considered the set of trees (here expanded):

```

- datatype tree = Oak | Maple | Pine | Elm | Spruce | Fir | Willow;

datatype tree = Elm | Fir | Maple | Oak | Pine | Spruce | Willow

```

We know that equality is defined automatically. By chaining comparisons using `orelse`, we can create more complicated statements and encapsulate them in predicates.

```

- fun isConiferous(tr) =
=   tr = Pine orelse tr = Spruce orelse tr = Fir;

val isConiferous = fn : tree -> bool

- isConiferous(Willow);

val it = false : bool

- isConiferous(Pine);

val it = true : bool

```

Notice that we never explicitly consider the cases for non-coniferous trees. An equivalent way of writing this is

```

- fun isConiferous2(tr) =
=   if (tr = Pine)
=     then true
=     else if (tr = Spruce)
=           then true
=           else if (tr = Fir)
=                 then true
=                 else false;

val isConiferous2 = fn : tree -> bool

- isConiferous2(Oak);

val it = false : bool

- isConiferous2(Spruce);

val it = true : bool

```

Although this is much more verbose, a pattern like this would be necessary if such a predicate (or, more generally, function) needed to perform more computation to determine its result (as opposed to merely returning literals `true` and `false`). ML does, however, provide a cleaner way of using a pattern conceptually the same as that used above. Instead of naming a variable for the predicate to receive, we write a series of expressions for explicit input values:

```

- fun isConiferous3(Pine) = true
=   | isConiferous3(Spruce) = true
=   | isConiferous3(Fir) = true
=   | isConiferous3(x) = false;

val isConiferous3 = fn : tree -> bool

- isConiferous3(Maple);

val it = false : bool

- isConiferous3(Fir);

```

```
val it = true : bool
```

This is referred to as *pattern matching* because the predicate is evaluated by finding the definition that matches the input. Patterns can become more complicated than we have seen here. Notice also that we still use a variable in the last line of the definition of `isConiferous3` as a default case.

It is legal to define a predicate so that it leaves some cases undefined. However, that will generate a warning when you define it and an error message if you try to use it on a value for which it is not defined.

```
- fun isConiferous4(Pine) = true
= |   isConiferous4(Spruce) = true
= |   isConiferous4(Fir) = true;

stdIn:38.1-40.31 Warning: match nonexhaustive
    Pine => ...
    Spruce => ...
    Fir => ...
```

```
val isConiferous4 = fn : tree -> bool
```

```
- isConiferous4(Spruce);
```

```
val it = true : bool
```

```
- isConiferous4(Pine);
```

```
val it = true : bool
```

```
- isConiferous4(Elm);
```

```
uncaught exception nonexhaustive match failure
```

As an example of a more complicated pattern, consider a 2-place predicate. Suppose you are having guests over and want to serve food that will not violate any of your guests' dietary restrictions. Arwen tells you that she is a vegetarian, Luca says he is on a low-sodium diet, and Jael tells you that she eats kosher. Estella says she will eat anything, and Bogdan claims he eats nothing, but you figure that everyone will eat ice cream. To test what combination of foods would be palatable, you use the following ML system:

```
- datatype guests = Arwen | Luca | Jael | Estella | Bogdan;

datatype guests = Arwen | Bogdan | Estella | Jael | Luca

- datatype foods = FrenchFries | Chicken | Bacon | Spinach | IceCream;

datatype foods = Bacon | Chicken | FrenchFries | IceCream | Spinach

- fun isKosher(Bacon) = false
= |   isKosher(x) = true;

val isKosher = fn : foods -> bool
```

```
- fun isMeat(Chicken) = true
= |   isMeat(Bacon) = true
= |   isMeat(x) = false;

val isMeat = fn : foods -> bool

- fun isSalty(Bacon) = true
= |   isSalty(FrenchFries) = true
= |   isSalty(x) = false;

val isSalty = fn : foods -> bool

- fun eats(x, IceCream) = true
= |   eats(Estella, y) = true
= |   eats(Arwen, x) = not (isMeat(x))
= |   eats(Luca, x) = not (isSalty(x))
= |   eats(Jael, x) = isKosher(x)
= |   eats(Bogdan, x) = false;

val eats = fn : guests * foods -> bool
```


Exercises

Let M be the set of men, U be the set of unicorns, $h(x, y)$ be the predicate that x hunts y , and $s(x, y)$ be the predicate that x is smarter than y . Write the following symbolically, then negate them, and then write the negation in English.

1. A certain man hunts every unicorn.
2. Every man is smarter than every unicorn.
3. Any man is mortal if he hunts a unicorn.
4. There is a smartest unicorn.
5. No man hunts every unicorn.

Evaluate the negation

6. $\sim \forall x \in D, \exists y \in E | P(x, y)$.
7. $\sim \exists x \in D | \forall y \in E, P(x, y)$.

After your dinner party you and your guests (from the earlier example) will be watching a movie. Nobody wants to watch *Titanic*. Luca, a Californian, wants to see the Governorator in *Terminator*. Jael is up for anything but a drama. Estella is in the mood for a good comedy. Bogdan has a crush on Estella and wants to watch whatever she wants.

8. Create a movie genre datatype with elements Drama, Comedy, and Action. Create a movie datatype with elements *Terminator*, *Shrek*, *Titanic*, *Alien*, *GoneWithTheWind*, and *Airplane*. Copy the guest datatype from the example.
9. Write three predicates using pattern-matching to determine the genre of a movie (i.e. isDrama, isComedy, isAction).
10. Write a predicate wantsToWatch(guest, movie) that determines if a guest will watch a particular movie.
11. Will Bogdan watch *Shrek*? Will Jael watch *Gone With The Wind*?

Part III

Proof

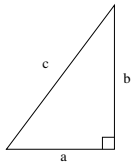
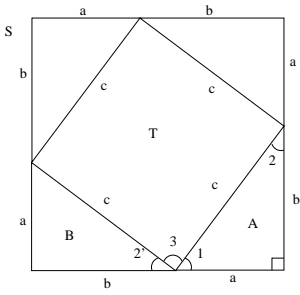
Chapter 10

Subset proofs

10.1 Introductory remarks

There are two principle goals for this course: writing proofs and writing programs. Writing programs develops as a crescendo in that you gather pieces throughout this book until their grand assembling in Part VII. Writing proofs, on the other hand, is a skill you will practice over and over, as a figure skater does a triple axel, for the entire course. It is therefore difficult to overstate the importance of these next few chapters.

A traditional high school mathematics curriculum teaches proof writing in a course on geometry. You may remember the simple two-column format for proof-writing, with succeeding propositions in the left column and corresponding justifications in the right (this is not too different conceptually from what we did in Section 7.3). Let us apply this method to proving the Pythagorean Theorem, which states that the square of the hypotenuse of a right triangle is equal to the sum of the squares of the other two sides, or, for the triangle below left, $a^2 + b^2 = c^2$. One way to prove this is to reduplicate the triangle, rotating and arranging as shown below center, where T is the inner square and S is the outer square. We then reason as we see below right (using charming high school abbreviations and symbols).

$\triangle A \cong \triangle B$ $\angle 1 + \angle 2 = 90^\circ$ $\angle 1 + \angle 2' = 90^\circ$ $\angle 3 = 90^\circ$ T is a square Area of $T = c^2$ Area of $S = (a + b)^2$ Area of each $\triangle = \frac{ab}{2}$ $(a + b)^2 = c^2 + 4\frac{ab}{2}$ $a^2 + 2ab + b^2 = c^2 + 2ab$ $\therefore c^2 = a^2 + b^2$	SSS \triangle angles sum to 180° $\angle 2 \cong \angle 2'$ Supplementary \angle s Equal sides, 90° \angle s Area of \square Area of \square Area of \triangle Sum of areas Algebra (FOIL, simplification) Subtract $2ab$ from both sides.
---	--

Proofs in real mathematics, however, require something more professional. Proofs should be written as paragraphs of complete English sentences—though mathematical symbolism is often useful for conciseness and precision.

10.2 Forms for proofs

A *theorem* is a proposition that is proven to be true. Thus a paradox (or any other non-proposition) cannot be a theorem, because a non-proposition is neither true nor false (or it is both). A false proposition also cannot be a theorem because it is false. Like a theorem, an axiom is true, but unlike a theorem, an axiom is assumed to be true rather than proven true. Finally, a theorem is different from a conjecture, even one that happens to be true, in that a theorem is proven to be true, whereas a conjecture is not proven (and therefore not known for certain) to be true. Our business is

theorem

the promotion of conjectures to theorems by writing proofs for them. However, we will often speak of this as “to prove a theorem,” since all propositions you will be asked to prove will have been proven before.

Basic theorems take on one of three General Forms:

1. Facts. p
2. Conditionals. If p then q .
3. Biconditionals. p iff q .

Of these, General Form 2 is the most important, since facts can often be restated as conditionals, and biconditionals are actually just two separate conditionals. The theorems we shall prove in this part of the book all come out of set theory, and basic facts in set theory take on one of three Set Proposition Forms:

1. Subset. $X \subseteq Y$.
2. Set equality. $X = Y$.
3. Set empty. $X = \emptyset$.

In this chapter, we will work on proving the simplest kinds of theorems: those that conform to General Form 1 and Set Form 1. In the next chapter, we will consider theorems of General Form 1 and Set Forms 2 and 3, and the chapter after that will cover General Forms 2 and 3.

Let A and B be sets, subsets of the universal set U . An example of a proposition in General Form 1, Set Form 1 is

Theorem 10.1 $A \cap B \subseteq A$

This is a simple fact (not modified by a conditional) expressing a subset relation, that one set ($A \cap B$) is a subset of another (A). Our task is to prove that this is always the case, no matter what A and B are. To prove that, we need to ask ourselves *What does it mean for one set to be a subset of another?* and *Why is it the case that these two sets are in that relationship?*

The first question appeals to the definition of subset. Formal, precise definitions are extremely important for doing proofs. Chapter 1 gave an informal definition of the subset relation, but we need a formal one in order to reason precisely. Our knowledge of quantified logic allows us to define

$$X \subseteq Y \text{ if } \forall x \in X, x \in Y$$

(Observe how this fact now breaks down into a conditional. “ $A \cap B \subseteq A$ ” is equivalent to “if $a \in A \cap B$ then $a \in A$.” This observation will make proving conditional propositions more familiar when the time comes. More importantly, you should notice that definitions, although expressed merely as conditionals, really are biconditionals; the “if” is an implied “iff.”)

The burden of our proof to show $A \subseteq B$ is, then, to show that

$$\forall a \in A, a \in B$$

Notice that this is a special case of the more general form

$$\forall a \in A, P(a)$$

letting $P(a) = a \in B$. Think back to the previous chapter. How would you persuade someone that this is the case? You allow the doubter to pick an element of A and then show that that element makes the predicate true. The way we express an invitation to the doubter to pick an element is with the word *suppose*. “Suppose $a \in A \dots$ ” is math-speak for “choose for yourself an element of A , and I will tell you what to do with it, which will persuade you of my point.”

In our case, the set that is a subset is $A \cap B$. The formal definition of intersect is

$$X \cap Y = \{z \mid z \in X \wedge z \in Y\}$$

Now follow the proof:

Proof. Suppose $a \in A \cap B$.

By definition of intersection, $a \in A$ and $a \in B$.

$a \in A$ *by specialization*.

Therefore, by definition of subset, $A \cap B \subseteq A$. \square

One line is italicized because it really could be omitted for the sake of brevity (not that this particular proof needs to be more brief). The proofs in this text frequently will have italicized sentences which will add clarity (but also clutter) to proofs; you may omit similar sentences when you write proofs yourself. Specialization is the sort of logical step that it is fair to assume your audience will perform automatically, as long as you recognize that a real logical operation is indeed happening. Notice that

- Our proof begins with *Suppose...*
- Every other sentence is a proposition joined with a prepositional phrase governed by *by*.
- The last sentence begins with *therefore* and, except for the *by* part, is the proposition we are proving.
- The proof is terminated by the symbol \square , a widely used end-of-proof marker. You will sometimes see proofs terminated with *QED*, an older convention from the Latin *quod erat demonstrandum*, which means “which was to be proven.”

Compare that with our arguments in Section 7.3. Our overall strategy in this case is what is called the *element argument* for proving facts of set form 1:

element argument

To prove $A \subseteq B$

say Suppose $a \in A$.

followed by ... a sequence of propositions that logically follow each other ...

with second-to-last sentence $a \in B$ *by* ...

and last sentence Therefore, $A \subseteq B$ by the definition of subset. \square

Proofs at the beginning level are all about *analysis* and *synthesis*. Analysis is the taking apart of something. Break down the assumed or proven propositions by applying definitions, going from *term* to *meaning*. Synthesis is the putting together of something. Assemble the proposition to be proven by applying the definition in the other direction, going from *meaning* to *term*. Here is a summary of the formal definitions of set operations we have seen before.

$$\begin{array}{ll} X \cup Y &= \{z \mid z \in X \wedge z \in Y\} & X - Y &= \{z \mid z \in X \wedge z \notin Y\} \\ X \cap Y &= \{z \mid z \in X \wedge z \in Y\} & X \times Y &= \{(x, y) \mid x \in X \wedge y \in Y\} \\ \overline{X} &= \{z \mid z \notin X\} \end{array}$$

10.3 An example

Now we consider a more complicated example. The same principles apply, only with more steps along the way. Let A, B , and C be sets, subsets of U . Prove

$$A \times (B \cup C) \subseteq (A \times B) \cup (A \times C)$$

Immediately we notice that this fits our form, so we know our proof will begin with

Proof. Suppose $x \in A \times (B \cup C)$.

and end with

$x \in (A \times B) \cup (A \times C)$ by Therefore, $A \times (B \cup C) \subseteq (A \times B) \cup (A \times C)$. \square

Our proof is going to be a journey from $x \in A \times (B \cup C)$ to $x \in (A \times B) \cup (A \times C)$. Most steps will be the application of a definition. To trace out a route, we need to consider what paths lead from $x \in A \times (B \cup C)$ and which paths lead to $x \in (A \times B) \cup (A \times C)$, and to find a way to connect them in the middle.

What has the definition of Cartesian product to say about $x \in A \times (B \cup C)$? x must be an ordered pair, having the form, say, (a, d) . Having picked the symbols a and d , we know that $a \in A$ and $d \in B \cup C$. What we know about d can be broken down further, observing that $d \in B$ or $d \in C$. This is the analysis of our supposition, broken down to individual facts.

What about $x \in (A \times B) \cup (A \times C)$? What would this mean, if it were true? x would have to be an element of $A \times B$ or of $A \times C$. In the first case, it would have to have the form (a, d) where $a \in A$ and $d \in B$; in the other case, it would still have to have the form (a, d) where $a \in A$, but instead where $d \in C$. This is a hypothetical analysis of our destination. Notice our use of the subjunctive throughout, since we do not know that these parts are true, only that they would be true if the destination were true. However, if we proved all these parts by some other means, then retracing our steps would lead to a synthesis of our destination. In this instance, the glue that connects these with the pieces of the previous analysis is the argument form we have learned called *division into cases*.

Proof. Suppose $x \in A \times (B \cup C)$. By the definition of Cartesian product, $x = (a, d)$ for some $a \in A$ and some $d \in B \cup C$. Then $d \in B$ or $d \in C$, by definition of union.

Case 1: Suppose $d \in B$. Then, by definition of Cartesian product, $(a, d) \in A \times B$. Moreover, $x \in A \times B$ by substitution. Finally, by the definition of union, $x \in (A \times B) \cup (A \times C)$.

Case 2: Suppose $d \in C$. Then, by definition of Cartesian product, $(a, d) \in A \times C$. Moreover, $x \in A \times C$ by substitution. Finally, by the definition of union, $x \in (A \times B) \cup (A \times C)$.

So $x \in (A \times B) \cup (A \times C)$ by division into cases. Therefore, $A \times (B \cup C) \subseteq (A \times B) \cup (A \times C)$. \square

Notice several things. First, instead of writing each sentence on its own line, we combined several sentences into paragraphs. This manifests the general divisions in our proof technique. The first paragraph did the analysis. The next two each dealt with one case, also beginning the synthesis. The last paragraph completed the synthesis.

Second, division into cases was turned into prose and paragraph form by highlighting each case, with each case coming from a clause of a disjunction (" $d \in B$ or $d \in C$ "), and each case requires another supposition. We will see this structure again later, and take more careful note of it then.

Third, we have peppered this proof with little words like "then," "moreover," "finally," and "so." These do not add meaning, but they make the proof more readable.

Finally, we have made use of one extra but very important mathematical tool, that of substitution. If two expressions are assumed or shown to be equal, we may substitute one for the other in another expression. In this case, we assumed x and (a, d) were equal, and so we substituted x for (a, d) in $(a, d) \in A \times B$. (Or, we *replaced* (a, d) with x . Do not say that we substituted (a, d) with x . See Fowler's *Modern English Usage* on the matter [7].)

10.4 Closing remarks

Writing proofs is writing. Your English instructors have long forewarned that good writing skills are essential no matter what field you study. The day of reckoning has arrived. When you write proofs,

you will write in grammatical sentences and fluent paragraphs. Two-column grids with non-parallel phrases are for children. Sentences and paragraphs are for adults. However, you are by no means expected to write good proofs immediately. Proof-writing is a fine skill that takes patience and practice to acquire. Do not be discouraged when you write pieces of rubbish along the way. Being able to write proofs is a goal of this course, not a prerequisite.

More specifically, writing proofs is *persuasive* writing. You have a proposition, and you want your audience to believe that it is true. Yet mathematical proofs have a character of their own among other kinds of persuasive writing. We are not about the business of amassing evidence or showing that something is probable—mathematics stands out even among the other sciences in this regard. One proof cannot merely be stronger than another; a proof either *proves* its theorem absolutely or not at all. A mathematical proof of a theorem may in some ways be less weighty than, say, an argument for a theological position. On the other hand, a mathematical proof has a level of precision that no other discourse community can approach. This is the community standard to which you must live up; when you write proofs, *justify everything*.

You may imagine the stereotypical drill sergeant telling recruits, “When you speak to me, the first and last word out of your mouth better be ‘sir.’ ” You will notice an almost militaristic rigidity in the proof-writing instruction in this book. Every proof must begin with *suppose*. Every other sentence must contain *by* or *since* or *because* (with a few exceptions, which will generally contain another *suppose* instead). Your last sentence must begin with *therefore*. Your proofs will conform to a handful of patterns like the element argument we have seen here. However, this does not represent the whole of mathematical proofs. If you go on in mathematical study, you will see that the structure and phrasing of proofs can be quite varied, creative, and even colorful. Steps are skipped, mainly for brevity. Phrasing can be less precise, as long as it is believable that they could be tightened up. However, this is not yet the place for creativity, but for fundamental training. We teach you to march now. Learn it well, and someday you will dance.

Exercises

Let A, B , and C be sets, subsets of the universal set. Prove.

1. $A \subseteq A \cup B$.
2. $A - B \subseteq \overline{B}$.
3. $A \cap \overline{B} \subseteq A - B$.
4. $\overline{(A \cup B)} \subseteq \overline{A} \cap \overline{B}$.
5. $A \cup (B \cap C) \subseteq (A \cup B) \cap (A \cup C)$.
6. $(A \times B) \cup (A \times C) \subseteq A \times (B \cup C)$.
7. $A \times (B - C) \subseteq (A \times B) - (A \times C)$.

Chapter 11

Set equality and empty proofs

11.1 Set equality

We now concern ourselves with proving facts of Set Form 2, that is, propositions in the form $A = B$ for some sets A and B . As with subsets, proofs of set equality must be based on the definition, what it means for two sets to be equal. Informally we understand that two sets are equal if they are made up of all the same elements, that is, if they completely overlap. In other words, two sets are equal if they are subsets of each other. Formally:

$$X = Y \text{ if } X \subseteq Y \wedge Y \subseteq X$$

This means you already know how to prove propositions of set equality—it is the same as proving subsets, only it needs to be done twice (once in each direction of the equality). Observe this proof of $A - B = A \cap \overline{B}$:

Proof. First, suppose $x \in A - B$. By the definition of set difference, $x \in A$ and $x \notin B$. By the definition of complement, $x \in \overline{B}$. Then, by the definition of intersection, $x \in A \cap \overline{B}$. Hence, by the definition of subset, $A - B \subseteq A \cap \overline{B}$.

Next, suppose $x \in A \cap \overline{B}$ *Fill in your proof from Chapter 10, Exercise 3.* ... Hence, by the definition of subset, $A \cap \overline{B} \subseteq A - B$.

Therefore, by the definition of set equality, $A - B = A \cap \overline{B}$. \square

Notice that this proof required two parts, highlighted by “first” and “next,” each part with its own supposition and each part arriving at its own conclusion. We used the word “hence” to mark the conclusion of a part of the proof and “therefore” to mark the end of the entire proof, but they mean the same thing. Notice also the general pattern: suppose an element in the left side and show that it is in the right; suppose an element in the right side and show that it is in the left.

To avoid redoing work (and making proofs unreasonably long), you may use previously proven propositions as justification in a proof. A theorem that is proven only to be used as justification in a proof of another theorem is called a *lemma*. Lemmas and theorems are either identified by name or by number. For our purposes, we can also refer to theorems by their exercise number. Here is a re-writing of the proof:

lemma

Lemma 11.1 $A - B \subseteq A \cap \overline{B}$.

Proof. Suppose $x \in A - B$. By the definition of set difference, $x \in A$ and $x \notin B$. By the definition of complement, $x \in \overline{B}$. Then, by the definition of intersection, $x \in A \cap \overline{B}$. Therefore, by the definition of subset, $A - B \subseteq A \cap \overline{B}$. \square

Now the proof of our theorem becomes a one-liner (incomplete sentences may be countenanced when things get this simple):

Proof. By Lemma 11.1, Exercise 10.3, and the definition of set equality. \square

11.2 Set emptiness

Suppose we are to prove

$$A \cap \bar{A} = \emptyset$$

To address Set Form 3, we must consider what it means for a set to be empty; though this may seem obvious, a precise proof cannot be written if it does not have a precise definition for which to aim. A set X is empty if

$$\sim \exists x \in U \mid x \in X$$

It is frequently useful to prove that a certain sort of thing does not exist. The doubter's objection that "just because you have not found one does not mean they do not exist" is quite a high hurdle for the prover. We do, however, have a weapon for propositions of this sort—the proof by contradiction syllogism we learned in Chapter 7. We suppose the opposite of what we are trying to prove

$$\exists x \in U \mid x \in X$$

show that this leads to a contradiction, and then conclude what we were trying to prove. This is indeed one of the most profound techniques in mathematics. G.H. Hardy remarked, "It is a far finer gambit than any chess gambit: a chess player may offer the sacrifice of a pawn or even a piece, but the mathematician offers the game."

Proof. Suppose $a \in A \cap \bar{A}$. Then, by definition of intersection, $a \in A$ and $a \in \bar{A}$. By the definition of complement, $a \notin A$. $a \in A$ and $a \notin A$ is a contradiction. *This proves that the supposition $a \in A \cap \bar{A}$ is false.* Therefore, $A \cap \bar{A} = \emptyset$. \square

11.3 Remarks on proof by contradiction

The more powerful the tool, the more easily it can be misused. So it is with proof by contradiction. Since we can prove that a proposition is *false* by deriving a known false proposition from it, the novice prover is often tempted to prove that a proposition is *true* by deriving a known true proposition from it. This is a logical error of the most magnificent kind, which you must be careful to avoid. Let this example, to prove $A = \emptyset$ (that is, all sets are empty), demonstrate the folly to would-be trespassers:

Proof. Suppose $A = \emptyset$. Then $A \cup U = \emptyset \cup U$ by substitution. By Exercise 11, $A \cup U = U$ and also $\emptyset \cup U = U$. By substitution, $U = U$, an obvious fact. Hence $A = \emptyset$. \square

In other words, there is no *proof by tautology*. The truth table below presents another way to see why not—in the second critical row, the conclusion is false.

$p \rightarrow T$	p	T	$p \rightarrow T$	p	
$\therefore p$	T	T	T	T	\leftarrow critical row
	F	T	T	F	\leftarrow critical row

Exercises

Let A, B , and C be sets, subsets of the universal set U . Prove. You may use exercises from the previous chapter in your proofs.

1. $A \cup \emptyset = A$.
2. $A \cup (A \cap B) = A$.
3. $A \times (B \cup C) = (A \times B) \cup (A \times C)$.
4. $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.
5. $A \cup \overline{A} = U$.
6. $A \times (B - C) = (A \times B) - (A \times C)$.
7. $A \cup B = B \cup A$.
8. $(A \cup B) \cup C = A \cup (B \cup C)$.
9. $(A \cap B) \cap C = A \cap (B \cap C)$.
10. $\overline{\overline{A}} = A$.
11. $A \cup U = U$.
12. $\overline{(A \cup B)} = \overline{A} \cap \overline{B}$.
13. $\overline{(A \cap B)} = \overline{A} \cup \overline{B}$.
14. $A \cup (A \cap B) = A$.
15. $A \cup B = A \cup (B - (A \cap B))$.
16. $A \cap \emptyset = \emptyset$.
17. $A - \emptyset = A$.
18. $A \cap \overline{A} = \emptyset$.
19. $A \times \emptyset = \emptyset$.
20. $A - A = \emptyset$.
21. $(A - B) \cap (A \cap B) = \emptyset$.
22. $(A - B) \cap B = \emptyset$.
23. $A \cup (B - (A \cap B)) = \emptyset$.

Chapter 12

Conditional proofs

12.1 Worlds of make believe

In this chapter we concern ourselves with proofs of General Forms 2 and 3, conditional and biconditional propositions. As an initial example, consider the proposition

If $A \subseteq B$, then $A \cap B = A$.

Proof. Suppose $A \subseteq B$.

Further suppose that $x \in A \cap B$. By definition of intersection, $x \in A$. Hence $A \cap B \subseteq A$ by definition of subset.

Proof of
 $A \cap B \subseteq A$.

Now suppose that $x \in A$. Since $A \subseteq B$, then $x \in B$ as well by definition of subset. Then by definition of intersection, $x \in A \cap B$. Hence $A \subseteq A \cap B$ by definition of subset.

Proof of
 $A \subseteq A \cap B$.

Proof of
 $A \cap B = A$.

Entire proof.

Therefore, by definition of set equality, $A \cap B = A$.
□

We see from this that the proof of this slightly more sophisticated proposition is really composed of smaller proofs with which we are already familiar. At the innermost levels, we have subset proofs, two of them. Together, they constitute a proof of set equality, as we saw in the previous chapter. We are now merely wrapping that proof in one more layer to get a proof of a conditional proposition. That “one more layer” is another supposition. Take careful stock in how the word *suppose* is used in proof above.

The sub-proof of $A \subseteq A \cap B$ makes no sense out of context. Certainly a set A is not in general a subset of its intersection with another set. What makes that true (as we say in the proof) is that we have supposed a restriction, namely $A \subseteq B$. The wrapper provides a context that makes the proposition $A \subseteq A \cap B$ true.

When we say *suppose*, we are boarding Mister Rogers’s trolley to the Neighborhood of Make-Believe. We are creating a fantasy world in which our supposition is true, and then demonstrating that something else happens to be true in the world we are imagining. The imaginary world must obey all mathematical laws plus the laws we postulate in our supposition. Sometimes, it is useful to make another supposition, in which case we enter a fantasy world *within* the first fantasy world—that world must obey all the laws of the outer world, plus whatever is now supposed.

Trace our journey through the proof above. We begin in the real world (or at least the world of mathematical set theory), World 0. We imagine a world in which $A \subseteq B$, World 1. From that world

we imagine yet another world, World 2, in which $x \in A \cap B$, and show that in World 2, $x \in A$. This proves that *any* world like World 2 within World 1 will behave that way, and this proves that within World 1, $A \cap B \subseteq A$. (It happens that this is true in World 0 as well, but we do not prove it.) We exit World 2. We then imagine a World 3 within World 1 in which $x \in A$, and in a way similar to what we had before, show that $A \subseteq A \cap B$. Together, these things about World 1 also show that $A \cap B = A$ in World 1, and our proof is complete. Notice that our return to World 0 is not explicit. This is because the proposition we are proving does not ask us to prove anything directly about the real world (as propositions in General Form 1 do). Rather, it asks us to prove something about all worlds in which $A \subseteq B$.

12.2 Integers

For variety, let us try out these proof techniques in another realm of mathematics. Here we will prove various propositions about integers, particularly about what facts depend upon them being even or odd. These proofs will rely on formal definitions of even and odd: An integer x is *even* if

$$\exists k \in \mathbb{Z} \mid x = 2k$$

and an integer x is *odd* if

$$\exists k \in \mathbb{Z} \mid x = 2k + 1$$

We will take as axioms that integers are closed under addition and multiplication, and that all integers are either even or odd and not both.

Axiom 3 If $x, y \in \mathbb{Z}$, then $x + y \in \mathbb{Z}$.

Axiom 4 If $x, y \in \mathbb{Z}$, then $x \cdot y \in \mathbb{Z}$.

Axiom 5 If $x \in \mathbb{Z}$, then x is even iff x is not odd.

You may also use basic properties of arithmetic and algebra in your proofs. Cite them as “by rules of arithmetic” or “by rules of algebra,” although if you recall the name of the specific rule or rules being used, your proof will be better if you cite them. We begin with

If x and y are even integers, then $x + y$ is even.

Proof. Suppose x and y are even integers. By the definition of even, there exist $j, k \in \mathbb{Z}$ such that $x = 2j$ and $y = 2k$. Then

$$\begin{aligned} x + y &= 2j + 2k && \text{by substitution} \\ &= 2(j + k) && \text{by distribution} \end{aligned}$$

Further, $j + k \in \mathbb{Z}$ because integers are closed under addition. *Hence there is an integer, namely $j + k$, such that $x + y = 2(j + k)$.* Therefore $x + y$ is an even integer by the definition of even. \square

Notice how we brought in the variables j and k . By saying “...there exist j, k ...”, we have made an implicit supposition about what j and k are. The definition of even establishes that this supposition is legal. Notice also how we structured the steps from $x + y$ to $2(j + k)$. This is a convenient shorthand for dealing with long chains of equations. Were we to write this out fully, we would say $x + y = 2j + 2k$ by substitution, $2j + 2k = 2(j + k)$ by distribution, and $x + y = 2(j + k)$ by substitution again (or by the transitivity of equals). This is not so bad when we are juggling only three expressions, but a larger number would become unreadable without chaining them together. Finally, the second to last sentence is italicized because it merely rephrases what the previous two sentences gave us. It is included here for explicitness, but you may omit such sentences.

12.3 Biconditionals

Biconditional propositions (those of General Form 3) stand toward conditional propositions in the same relationship as proofs of set equality stand toward subset proofs. A biconditional is simply two conditionals written as one proposition; one merely needs to prove both of them.

$$A - B = \emptyset \text{ iff } A \subseteq B$$

Proof. First suppose $A - B = \emptyset$. *We will prove that $A \subseteq B$.* Suppose $x \in A$. Since $A - B = \emptyset$, it cannot be that $x \in A - B$, by definition of empty set. By definition of set difference, it is not true that $x \in A$ and $x \notin B$, and hence by DeMorgan's law, it is true that either $x \notin A$ or $x \in B$. Since $x \in A$, then by elimination, $x \in B$. Hence $A \subseteq B$.

Conversely, suppose $A \subseteq B$. *We will prove that $A - B = \emptyset$.* Suppose $x \in A - B$. By definition of set difference, $x \in A$ and $x \notin B$. By definition of subset, $x \in B$, contradiction. Hence $A - B = \emptyset$. \square

Notice how many suppositions we have scattered all over the proof—and we are still proving fairly simple propositions. To avoid confusion you should use paragraph structure and transition words to guide reader around the worlds you are moving in and out of. The word *conversely*, for example, indicates that we are now proving the second direction of a biconditional proposition (which is the *converse* of the first direction).

12.4 Warnings

We conclude this chapter—and the part of this book explicitly about proofs—with the exposing of logical errors particularly seductive to those learning to prove. Let not your feet go near their houses.

Arguing from example.

$2 + 6 = 8$. Therefore, the sum of two even integers is an even integer. \square

If this reasoning were sound, then this would also prove that the sum of any two even integers is 8.

Reusing variables.

Suppose x and y are even integers. By the definition of even, there exists $k \in \mathbb{Z}$ such that $x = 2k$ and $y = 2k$.

Since x and y are even, each of them is twice some other integer—but those are *different* integers. Otherwise, we would be proving that all even integers are equal to each other. What is confusing about the correct way we wrote this earlier, “there exist $j, k \in \mathbb{Z}$ such that $x = 2j$ and $y = 2k$,” is that we were contracting the longer phrasing, “there exists $j \in \mathbb{Z}$ such that $x = 2j$ and there exists $k \in \mathbb{Z}$ such that $y = 2k$.” Had we reused the variable k in the longer version, it would be clear that we were trying to reuse a variable we had already defined. This kind of mistake is extremely common for beginners.

Begging the question.

Suppose x and y are even. Then $x = 2j$ and $y = 2k$ for some $j, k \in \mathbb{Z}$. Suppose $x + y$ is even. Then $x + y = 2m$ for some $m \in \mathbb{Z}$. By substitution, $2j + 2k = 2m$, which is even by definition, and which we were to show. \square

This nonsense proof tries to postulate a world in which the proposition to be proven is already true, followed by irrelevant manipulation of symbols. You cannot make any progress by supposing what you mean to prove—this is merely a subtle form of the “proof by tautology” we repudiated in the previous chapter.

Substituting “if” for “because” or “since.”

First suppose $A - B = \emptyset$. Suppose $x \in A$. If $A - B = \emptyset$, then it cannot be that $x \in A - B$.
By definition of set difference, it is not true that $x \in A$ and $x \notin B \dots$

This is more a matter of style and readability than logic. Remember that each step in a proof should yield a new known proposition, justified by previously known facts. “If $A - B = \emptyset$, then it cannot be that $x \in A - B$ ” does no such thing, only informing us that $x \in A - B$, contingent upon $A - B = \emptyset$ being true. Instead, this part of the proof should assert that $x \notin A - B$ *because* $A - B = \emptyset$.

Exercises

Let A, B , and C be sets, subsets of the universal set U . For integers x and y to be consecutive means that $y = x + 1$. Prove. You may use exercises from the previous chapter in your proofs.

1. If $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$.
2. If $A \cap B = A \cap C$ and $A \cup B = A \cup C$, then $B = C$.
3. If $a \notin A$, then $A \cap \{a\} = \emptyset$. (This is the same as saying that if $a \notin A$, then A and $\{a\}$ are disjoint.)
4. $B \subseteq A$ iff $(A - B) \cup B = A$.
5. If $a \in A$ then $\mathcal{P}(A) = \mathcal{P}(A - \{a\}) \cup \{a \cup A' \mid A' \in \mathcal{P}(A - \{a\})\}$. Hint: First review what a powerset is. This is a complicated expression and consequently a difficult proof, but do not be intimidated. The idea is that we are splitting the subsets of A into two groups: those that contain a and those that do not. You need to show that these two groups comprise the entire powerset.
6. If $a \in A$ then $\mathcal{P}(A - \{a\}) \cap \{ \{a\} \cup A' \mid A' \in \mathcal{P}(A - \{a\}) \} = \emptyset$. (This also is a proof that two sets are disjoint. This together with the previous exercise show that these two sets make a partition of $\mathcal{P}(A)$.)
7. If x and y are odd integers, then $x + y$ is even.
8. If x and y are consecutive integers, then $x + y$ is odd.
9. If n^2 is odd, then n is odd. (Hint: try a proof by contradiction using Axiom 5.)
10. $x \cdot y$ is odd iff x and y are both odd.

Chapter 13

Special Topic: Russell's paradox

The usefulness of the set concept is its simpleness and its flexibility. For this reason set theory is a core component of the foundations of mathematics. An example of the concept's flexibility is that we can talk sensibly about sets of sets—for example, powersets. Reasoning becomes more subtle, however, if we speak of sets that even contain themselves. For example, the set X of all sets mentioned in this book is hereby mentioned in this book, and so $X \in X$. Bertrand Russell called for caution when playing with such ideas with the following paradox.

Let X be the set of all sets that do not contain themselves, that is, $X = \{Y \mid Y \notin Y\}$. Does X contain itself?

First, suppose it does, that is $X \in X$. However, then the definition of X states that only those sets that do not contain themselves are elements of X , so $X \notin X$, which is a contradiction. Hence $X \notin X$. But wait—the same definition of X now tells us that $X \in X$, and we have another contradiction.

A well-known puzzle, also attributed to Russell, presents the same problem. Suppose a certain town has only one man who is a barber. That barber shaves only every man in the town who does not shave himself. Does the barber shave himself? If he does, then he doesn't; if he doesn't, then he does.

We can conclude from this only that the setup of the puzzle is an impossibility. There could not possibly be a man who shaves only every man who does not shave himself. Likewise, the set of all sets that do not contain themselves must not exist. This is why rigorous set theory must be built on axioms. Although we have not presented a complete axiomatic foundation for set theory here, we have assumed that at least one set exists (namely, the empty set), and we have assumed a notion of what it means for sets to be equal. We have not assumed that any set that can be described necessarily exists.

For this reason, when we name sets in a proof (“let X be the set. . .”), we really are doing more than assigning a name to some concept; we are jumping to the conclusion that the concept exists. Therefore if we are defining sets in terms of a property (“let $X = \{x \mid x < 3\}$ ”), it is more rigorous to make that set a subset of a known (or postulated) set merely *limited* by that property (“let $X = \{x \in \mathbb{Z} \mid x < 3\}$ ”).

This clears away the paradox nicely. Since it makes sense only to speak about things in the universal set, we will assume that $X \subset U$, that is, $X = \{Y \in U \mid Y \notin Y\}$. Now the question, Does X contain itself?, becomes, Is it true that $X \in U$ and $X \notin X$? First suppose $X \in U$. Then either $X \notin X$ or $X \in X$. As we saw before, both of those lead to contradictions. Hence $X \notin U$. In other words, X does not exist.

Interestingly, this leaves powerset without a foundation, since we cannot define it as a subset of something else. Since we do not want to abandon the idea altogether, we place it on firm ground with its own axiom.

Axiom 6 (Powerset.) *For any set X , there exists a set $\mathcal{P}(X)$ such that $Y \in \mathcal{P}(X)$ if and only if $Y \subseteq X$.*

We have declared it O.K. to speak of powersets. However, we can prove that no set contains its own powerset.

Theorem 13.1 *If A is a set, then $\mathcal{P}(A) \not\subseteq A$.*

Proof. Suppose A is a set. Further suppose that $\mathcal{P}(A) \subseteq A$. Let $B = \{b \in A \mid b \notin b\}$. Since $B \subseteq A$, $B \in \mathcal{P}(A)$ by definition of powerset. By definition of subset, $B \in A$. Now we have two cases:

Case 1: Suppose $B \in B$. Then $B \notin B$, contradiction; this case is impossible.

Case 2: Suppose $B \notin B$. Then $B \in B$, contradiction, this case is impossible.

Either case leads to a contradiction. Hence $\mathcal{P}(A) \not\subseteq A$. \square

Moreover, this shows that the idea of a “set of all sets” is downright impossible in our axiomatic system.

Corollary 13.1 *The set of all sets does not exist.*

Proof. Suppose A were the set of all sets. By Axiom 6, $\mathcal{P}(A)$ also exists. Since $\mathcal{P}(A)$ is a set of sets and A is the set of all sets, $\mathcal{P}(A) \subseteq A$. However, by the theorem, $\mathcal{P}(A) \not\subseteq A$, a contradiction. Hence A does not exist. \square

This chapter draws heavily from Epp[5] and Hrbacek and Jech[9].

Part IV

Algorithm

Chapter 14

Algorithms

14.1 Problem-solving steps

So far we have studied *sets*, a tool for categorization; *logic*, rules for deduction; and *proof*, the application of these rules. We proceed now to another thinking skill, that of ordering instructions to describe a process of solving a problem. The importance of this to computer science is obvious, because programming is merely the ordering of instructions for a computer to follow. We will see if also as a mathematical concept.

An *algorithm* is a series of steps to solving a problem or discovering a result. An algorithm must have some raw material to work on (its *input*), and there must be a result that it produces (its *output*). The fact that it has input implies that the solution is *general*; you would write an algorithm for finding $n!$, not one for finding $5!$. An algorithm must be expressed in a way that is *precise* enough to be followed correctly.

algorithm

input

output

Much of the mathematics you learned in the early grades involved mastering algorithms—how to add multi-digit numbers, how to do long division, how to add fractions, etc. We also find algorithms in everyday life in the form of recipes, driving directions, and assembly instructions.

Let us take multi-digit addition. The algorithm is informed by how we arrange the materials we work on. The two numbers are analyzed by “columns,” which represent powers of ten. You write the two addends, one above the other, aligning them by the one’s column. The answer will be written below the two numbers, and carry digits will be written above. At every point in the process, there is a specific column that we are working on (call it the “current column”). Our process is

1. Start with the one’s column.
2. While the current column has a number for either of the addends, repeatedly
 - (a) Add the two numbers in the current column plus any carry from last time; let’s call this result the “sum.”
 - (b) Write the one’s column of the sum in the current column of the answer.
 - (c) Write the ten’s column of the sum in the carry space of the next column to the left.
 - (d) Let the next column to the left become our new current column.
3. If the last round of the repetition had a carry, write it in a new column for the answer.

Notice how this algorithm takes two addends as its input, produces an answer for its output, and uses the notions of sum and current column as temporary scratch space. It is particularly important to notice that the sum and the current column keep changing.

A grammatical analysis of the algorithm is instructive. All the sentences are in the imperative mood¹. Contrast this with *propositions*, which were all indicative. The algorithm does contain

¹It is a convenient aspect of English, however, that the “to” at the end of the previous paragraph makes them infinitives.

propositions, however: the independent clauses “the current column has a number for either of the addends” and “the last round of the repetition has a carry” are either true or false. Moreover, those propositions are used to guard steps 2 and 3; the words “if” and “while” guide decisions about whether or how many times to execute a certain command. Finally, note that step 2 is actually the repetition of four smaller steps, bound together as if they were one. This is similar to how we use simple expressions to build more complex ones.

We already know how to do certain pieces of this in ML: Boolean expressions represent propositions and `if` expressions make decisions. What we have not yet seen is how to *repeat*, how to *change* the value of a variable, how to *compound* steps together to be treated as one, or generally how to do anything explicitly imperative. ML’s tools for doing these things will seem clunky because it goes against the grain of the kind of programming for which ML was designed. It is still worth our time to consider imperative algorithms; bear with the unusual syntax for the next few chapters, and after that ML will appear elegant again.

14.2 Repetition and change

statement

A *statement* is a complete programming construct that does not produce a value (and so differs from an expression). If you evaluate a statement in ML, it will respond with

```
val it = () : unit
```

As it appears, this literally is an empty tuple, and it is ML’s way of expressing “nothing” or “no result.” `()` is the only value of the type `unit`.

statement list

The simplest way to write statements is to use them in a *statement list*, which is a list of expressions separated by semicolons and enclosed in parentheses (thus it looks like a tuple except with semicolons instead of commas). A statement list is evaluated by evaluating each expression in order and throwing away the result except for the last, which it evaluates and returns as the value of the entire statement list. For example,

```
- (7 + 3; 15.6 / 34.7; 4 < 17; Dove);
```

```
val it = Dove : Bird
```

The results of `7 + 3`, `15.6 / 34.7`, and `4 < 17` are ignored, turning them into statements. A statement list itself, however, has a value and thus is an expression and not a statement. Statements are used for their *side effects*, changes they make to the system that will affect expressions later. We do not yet know anything that has a side effect.

side effects

while statement

A *while statement* is a construct which will evaluate an expression repeatedly as long as a given condition is true. Its form is

```
while <expression> do <expression>
```

Since it is a statement, it will always result in `()` so long as it does indeed finish. If we try

```
- val x = 4;
```

```
val x = 4 : int
```

```
- while x < 5 do x - 3;
```

ML gives no response, because we told it to keep subtracting 3 from `x` as long as `x` is less than 5. Since `x` is 4, it is and always will be less than 5, and so the execution of the statement goes on forever. (Press control-C to make it stop.) The algorithm for adding two multi-digit numbers has a concrete stopping point: when you run out of columns in the addends. The problem here is that since variables do not change during the evaluation of an expression, there is no way that the boolean expression guarding the loop will change either. Trying

```
- while x < 5 do x = x - 3;
```

will merely compare x with $x - 3$ forever. The attempt

```
- while x < 5 do val x = x - 3;
```

```
stdIn:11.13-11.21 Error: syntax error: deleting DO VAL ID
```

is rejected because `val x = x - 3` is not an expression.

It turns out that in ML, variables are like the laws of the Medes and the Persians, which, once instated, cannot be repealed (Daniel 6:15, Esther 1:19). This seems to contradict your experience because you probably have reused variables in the same session.

```
- val y = 5;
```

```
val y = 5 : int
```

```
- val y = 16;
```

```
val y = 16 : int
```

```
- val y = Owl;
```

```
val y = Owl : Bird
```

What is actually happening here is like when King Xerxes, realizing he could not revoke an old decree, instead issued a new decree that counteracted the old one. Technically, when you reuse an identifier in ML, you are not changing the value of the old variable but making a new variable of the same name that shadows the old one. This is a technical detail that is transparent to most ML programming (you need not understand or remember it), but it makes a difference in writing imperative algorithms because you cannot redeclare a variable in the middle of an expression or statement.

To deal with this, ML allows for a different kind of variable, called a *reference variable*, which can change value. Three rules distinguish reference variables from ordinary ones.

reference variable

- To *declare* a reference variable, precede the expression assigned with the keyword `ref`.

```
- val x = ref 5;
```

```
val x = ref 5 : int ref
```

- To *use* a reference variable, precede the variable with `!`.

```
- !x;
```

```
val it = 5 : int
```

- To *set* a reference variable, use an *assignment statement* in the form

assignment statement

`<identifier> := <expression>`

```
- x := !x + 1;
```

```
val it = () : unit
```

```
- !x;
```

```
val it = 6 : int
```

Notice how types work here. The type of the variable `x` is `int ref`, and the type of the expression `!x` is `int`. The operator `!` always takes something of type `a' ref` for some type `a'` and returns something of type `a'`.

As an example, consider computing a factorial. Using product notation, we define

$$n! = \prod_{i=1}^n i$$

For instance,

$$5! = \prod_{i=1}^5 i = 1 \times 2 \times 3 \times 4 \times 5 = 125$$

Notice the algorithm nature of this definition (or just the product notation). It states to keep a running product while repeated multiplying by a multiplier, incrementing the multiplier by one at each step, starting at one, until a limit for the multiplier is reached. With reference variables in our arsenal, we can do this in ML:

```
- val i = ref 0;

val i = ref 0 : int ref

- val fact = ref 1;

val fact = ref 1 : int ref

- while !i < 5 do
=   (i := !i + 1;
=   fact := !fact * !i);

val it = () : unit

- !fact;

val it = 120 : int
```

The while statement produces only `()`, even though it computes `5!`. It does not report that answer (because it is not an expression), but instead changes the value of `fact` and `i`—a good example of a side effect. Thus we need to evaluate the expression `!fact` to discover the answer.

14.3 Packaging and parameterization

We used a statement list to compound the assignments to `i` and `fact`. Recall that the value of a statement list is the value of the last expression in the list. This way, we can compound the two last steps above into one step. This will give us a better sense of packaging, since the computation and the reporting of the result are really one idea.

```
- i := 0;

val it = () : unit

- fact := 1;
```

```

val it = () : unit

- (while !i < 5 do
=   (i := !i + 1;
=   fact := !fact * !i);
=   !fact);

val it = 120 : int

```

However, the variables `i` and `fact` will still exist after the computation finishes, even though they no longer have a purpose. The duration of a variable's validity is called its *scope*. A variable declared in the prompt has scope starting then and continuing until you exit ML. We would like instead to have variables that are *local* to the expression, that is, variables that the expression alone can use and that disappear when the expression finishes executing. We can make local variables by using a `let` expression with form

scope

```
let <var declaration>1; <var declaration>2; ... <var declaration>n in <expression> end
```

The value of the last expression is also the value of the entire `let` expression. Now we rewrite factorial computation as a single, self-contained expression:

```

- let
=   val i = ref 0;
=   val fact = ref 1;
= in
=   (while !i < 5 do
=     (i := !i + 1;
=     fact := !fact * !i);
=     !fact)
= end;

val it = 120 : int

```

The only thing deficient in our program is that it is not reusable. Why would we bother, after all, with a 9-line algorithm for computing 5! when the one line `1 * 2 * 3 * 4 * 5` would have produced the same result? The value of an algorithm is its generality. This is the same algorithm we would use to compute any other factorial $n!$, only that we would replace 5 in the fifth line with n . In other words, we would like to *parameterize* the expression, or wrap it in a package that accepts n as input and produces the factorial for any n . Just as we parameterized statements with `fun` to make predicates, so we can here.

```

- fun factorial(n) =
=   let
=     val i = ref 0;
=     val fact = ref 1;
=   in
=     (while !i < n do
=       (i := !i + 1;
=       fact := !fact * !i);
=       !fact)
=   end;

val factorial = fn : int -> int

- factorial(5);

```

```

val it = 120 : int

- factorial(8);

val it = 40320 : int

- factorial(10);

val it = 3628800 : int

```

For a second time, you are informally introduced to the notion of a function. We see it here as a way to package an algorithm so that it can be used as an expression when given an input value. Later we will see how to knit functions together to eliminate the need for while loops and reference variables in almost all cases.

14.4 Example

In Chapter 4 introduced arrays as a type for storing finite, uniform, and random accessible collections of data. You may recall that when you performed array operations that the interpreter responded with `() : unit`, which we now know indicates that these are statements. The following displays and uses an algorithm for computing long division. The function takes a divisor (as an integer) and a dividend (as an array of integers, each position representing one column), and it returns a tuple standing for the quotient and the remainder. Study this carefully.

```

- fun longdiv(divisor, dividend, len) =
=   let
=     val i = ref 0;
=     val result = ref 0;
=     val remainder = ref 0;
=   in
=     (while !i < len do
=       (remainder := !remainder * 10 + sub(dividend, !i);
=       let
=         val quotient = !remainder div divisor;
=         val product = quotient * divisor;
=       in
=         (result := !result * 10 + quotient;
=         remainder := !remainder - product)
=       end;
=       i := !i + 1);
=     (!result, !remainder))
= end;

val longdiv = fn : int * int array * int -> int * int

- longdiv(8,A,4);

val it = (440,1) : int * int

- val A = array(4, 0);

val A = [|0,0,0,0|] : int array

```

```
- update(A, 0, 3);  
  
val it = () : unit  
  
- update(A, 1, 5);  
  
val it = () : unit  
  
- update(A, 2, 2);  
  
val it = () : unit  
  
- update(A, 3, 1);  
  
val it = () : unit  
  
- A;  
  
val it = [|3,5,2,1|] : int array
```

Exercises

1. Suppose ML did not have a multiplication operator defined for integers. Write a function `multiply` which takes two integers and produces their product, without using direct multiplication (that is, using repeated addition).
2. A series in the form $\sum_{i=0}^n ar^i$ is called a *geometric series*. That is, given a value r and a coefficient a , find the sum of $a + ar + ar^2 + ar^3 + \dots + ar^n$. Write a function that computes a geometric series, taking n , a , and r as parameters.
3. Rewrite the long division function so that it receives the dividend as a normal integer rather than an array. (Hint: use `div` and `mod` to cut the dividend up into digits)
4. The Fibonacci sequence is defined by repeatedly adding the two previous numbers in the sequence (starting with 0 and 1) to obtain the next number, i.e.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Write a function to compute the n th Fibonacci number, given n .

(Hint: You will need four reference variables local to the function: one to serve as a counter, one to hold the current value, one to hold the previous value, and one to hold a temporary value as you shuffle the current value to the previous:

```
temp := ...
previous := !current
current := !temp
```

Then think about how you can rewrite this so that the temporary value can be local to the while statement, and a regular variable instead of a reference variable.)

Chapter 15

Induction

15.1 Calculating a powerset

Recall that the *powerset* of a set X is the set of all its subsets.

$$\mathcal{P}(X) = \{Y \mid Y \subseteq X\}$$

For example, the powerset of $\{1, 2, 3\}$ is $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Here is an algorithm for computing the powerset of a given set (represented by a list), anoted at the side.

```
- fun powerset(set) =
=   let
=     val remainingSet = ref set;
=     val powSet = ref ([[]] : int list list);
=   in
=     (while not (!remainingSet = nil) do
=       let
=         val powAddition = ref ([] : int list list);
=         val remainingPowSet = ref (!powSet);
=         val currentElement = hd(!remainingSet);
=       in
=         (while not (!remainingPowSet = nil) do
=           let
=             val currentSubSet = hd(!remainingPowSet);
=           in
=             (powAddition := (currentElement :: currentSubSet)
=              :: !powAddition;
=              remainingPowSet := tl(!remainingPowSet))
=           end;
=           powSet := !powSet @ !powAddition;
=           remainingSet := tl(!remainingSet))
=         end;
=         !powSet)
=     end;
```

remainingSet is the part of set we have not yet processed.
powSet is the powerset as we have calculated it so far.

While there is still some of the set left to process...

powAddition is what we are adding to the power set this time around.
remainingPowSet is the part of powSet we have not yet processed.
currentElement is the element of set we are processing this time around.
...pick the next element...

...and while there is still part of what we have made so far left to process...

currentSubSet is the subset we are processing this time around.
...pick the next subset...

...add currentElement to that subset...

...and add all those new subsets to the powerset.

```
val powerset = fn : int list -> int list list

- powerset([1,2,3]);

val it = [[], [1], [2, 1], [2], [3, 2], [3, 2, 1], [3, 1], [3]] : int list list
```

As we will see in a later chapter, this is far from the best way to compute this in ML. Dissecting it, however, will exercise your understanding of algorithms from the previous chapter. Of immediate interest to us is the relationship between the size of a set and the size of its powerset. Recall that the *cardinality* of a finite set X , written $|X|$, is the number of elements in the set. (Defining the cardinality of infinite sets is a more delicate problem, one we will pick up later.) With our algorithm handy, we can generate the powerset of sets of various sizes and count the number of elements in them.

```
- powerset([]);

val it = [[]] : int list list

- powerset([1]);

val it = [[],[1]] : int list list

- powerset([1,2]);

val it = [[],[1],[2],[1,2]] : int list list

- powerset([1,2,3,4]);

val it = [[],[1],[2],[3],[4],[1,2],[1,3],[1,4],[2,3],[2,4],[3,4],[1,2,3],[1,2,4],[1,3,4],[2,3,4],[1,2,3,4]] : int list list
```

$ A $	$ \mathcal{P}(A) $
0	1
1	2
2	4
3	8
4	> 12

The ellipses on the last result indicate that there are more items in the list, but the list exceeds the length that the ML interpreter normally displays. We summarize our findings in the adjacent table. The pattern we recognize is that $|\mathcal{P}(A)| = 2^{|A|}$ (so the last number is actually 16). An informal way to verify this hypothesis is to think about what the algorithm is doing. We start out with the empty set—all sets will at least have that as a subset. Then, for each other element in the original set, we add it to each element in our powerset so far. Thus, each time we process an element from the original set, we double the size of the powerset. So if the set has cardinality n , we begin with a set of cardinality 1 and double its size n times; the resulting set must have cardinality 2^n .

This makes sense, but how do we prove it formally? For this we will use a new proving technique, proof by mathematical induction.

15.2 Proof of powerset size

Our theorem is

Theorem 15.1 $|\mathcal{P}(A)| = 2^{|A|}$

This is in General Form 1 (straight facts, without explicit condition). However, restating this will make the proof easier. First, we define the predicate

$$I(n) = \text{If } A \text{ is a set and } |A| = n \text{ then } |\mathcal{P}(A)| = 2^n$$

invariant

The I is for *invariant*, since this is a condition that remains the same regardless of how we adjust n . Our theorem essentially says

$$\forall n \in \mathbb{W}, I(n)$$

What this gives us is that now we are predicating our work on whole numbers (“prove this for all n ”) rather than sets (“prove this for all A ”).

Before we go further, an important side note: The definition of cardinality we presented in Chapter 3 is informal, and we will not be able to define it more precisely until Chapter 25. For this reason, and also to reduce the complexity of the present proof, we will be appealing to various results proven elsewhere in the book.

*Result**Where found*

If A and B are finite sets and $A \subseteq B$, then $|B - A| = |B| - |A|$.

Chapter 25, Exercise 10.

If $a \in A$ then $\mathcal{P}(A) = \mathcal{P}(A - \{a\}) \cup \{a \cup A' \mid A' \in \mathcal{P}(A - \{a\})\}$.

Chapter 12, Exercise 5.

If $a \in A$ then $\mathcal{P}(A - \{a\}) \cap \{ \{a\} \cup A' \mid A' \in \mathcal{P}(A - \{a\}) \} = \emptyset$.

Chapter 12, Exercise 6.

If A is a finite set and $a \in A$, then $|\{ \{a\} \cup A' \mid A' \in \mathcal{P}(A - \{a\}) \}| = |\mathcal{P}(A - \{a\})|$.

Chapter 25, Exercise 11.

If A and B are finite, disjoint sets, then $|A \cup B| = |A| + |B|$.

Theorem 25.2.

Proof. Suppose $|A| = 0$. By definition of the empty set, $A = \emptyset$. Moreover, by definition of powerset, $\mathcal{P}(A) = \mathcal{P}(\emptyset) = \{\emptyset\}$, since \emptyset is the only subset of \emptyset . By definition of cardinality, $|\{\emptyset\}| = 1 = 2^0$, so $|\mathcal{P}(A)| = 2^0$. Hence $I(0)$.

Remember, we want to prove $I(n)$ for all n . We have proven it only for $n = 0$, leaving infinitely many possibilities to go. So far, this approach looks frighteningly like a proof by exhaustion with an infinity of cases. However, this apparently paltry result allows us to say one thing more.

Moreover, $\exists N \geq 0$ such that $I(N)$.

We know that this is true because it is at least true for $n = 0$. Possibly it is true for greater values of n also. But this foot in the door now lets us take all other cases in a giant leap.

Now, suppose $|A| = N + 1$. Let $a \in A$. (Since $N \geq 0$, $N + 1 \geq 1$, so we know A must have at least one element.) Note that $\{a\} \subseteq A$ by definition of subset and $|\{a\}| = 1$ by definition of cardinality.

$$\begin{aligned} |A - \{a\}| &= |A| - |\{a\}| && \text{by Exercise 10 of Chapter 25} \\ &= (N + 1) - 1 && \text{by substitution} \\ &= N && \text{by arithmetic.} \end{aligned}$$

By Exercises 5 and 6 of Chapter 12, $\mathcal{P}(A - \{a\})$ and $\{ \{a\} \cup A' \mid A' \in \mathcal{P}(A - \{a\}) \}$ are a partition of $\mathcal{P}(A)$.

Do not let the complicated notation intimidate you. All we are doing is splitting $\mathcal{P}(A)$ into the subsets that contain a and those that do not. Review the relevant exercises of Chapter 12 if necessary.

Since $I(N)$, $|\mathcal{P}(A - \{a\})| = 2^N$. By Exercise 11 of Chapter 25, $|\{ \{a\} \cup A' \mid A' \in \mathcal{P}(A - \{a\}) \}| = |\mathcal{P}(A - \{a\})|$. So,

$$\begin{aligned} |\mathcal{P}(A)| &= |\mathcal{P}(A - \{a\})| + |\{ \{a\} \cup A' \mid A' \in \mathcal{P}(A - \{a\}) \}| && \text{by Theorem 25.2} \\ &= 2^N + 2^N && \text{by substitution} \\ &= 2^{N+1} && \text{by exponential addition} \end{aligned}$$

Hence $I(N + 1)$.

Now we have proven (a) $I(N)$ for some N , and (b) if $I(n)$ then $I(n + 1)$. Stated differently, we have shown that it is true for the first case, and that if it is true for one case it is true for the next case. Thus,

By the principle of math induction, $I(n)$ for all $n \in \mathbb{W}$. Therefore, $|\mathcal{P}(A)| = 2^{|A|}$. \square

15.3 Mathematical induction

Induction, broadly, is a form of reasoning where a general rule is inferred from and verified by a set of observations consistent with that rule. Suppose you see five blue jays and notice that all of them are blue. From this you hypothesize that perhaps all blue jays are blue. You then proceed to see fifty more blue jays, all of them also blue. Now you are satisfied and believe that in fact, all blue jays are blue.

This differs from *deduction*, where one knows a general rule beforehand, and from that general rule infers a specific fact. Suppose you accepted the fact that all blue jays are blue as an axiom. If a blue jay flew by, you would have no reason even to look at it: you know it must be blue.

In fields like philosophy and theology, induction is viewed somewhat suspiciously, since it does not prove something beyond any doubt whatsoever. What if blue jay # 56 is red, and you simply quit watching too early? Inductive arguments are still used from time to time, though, because they can be rhetorically persuasive and because for some questions, it is simply the best we can do.

In natural science, in fact, we see inductive reasoning used all the time. The scientific method posits a hypothesis that explains observed phenomena and promotes a hypothesis to a theory only if many experiments (that is, controlled observations) consistently fit the hypothesis. The law of universal gravitation is called a law because so many experiments and observations have been amassed that suggest it to be true. It has never been proven in the mathematical sense, nor could it be.

In formal logic and mathematics, however, inductive reasoning is never grounds for proof, but rather is a gradiose but just as fallacious version of proof by example. Fermat's last theorem had been verified using computers to at least $n = 1000000$ [14], but it still was not considered proven until Andrew Wiles's proof. All this is to point out that math induction is *not* induction in the rhetorical sense. It is still deduction. It bears many surface similarities to inductive reasoning since it considers at least two examples and then generalizes, but that generalization is done by deduction from the following principle.

Axiom 7 (Mathematical Induction) *If $I(n)$ is a predicate with domain \mathbb{W} , then if $I(0)$ and if for all $N \geq 0$, $I(N) \rightarrow I(N + 1)$, then $I(n)$ for all $n \in \mathbb{W}$.*

(The principle can also be applied where $I(n)$ has domain \mathbb{N} , and we prove $I(1)$ first; or, any other integer may be used as a starting point. Assuming zero to be our starting point allows us to state the principle more succinctly.)

Even though we take this as an axiom for our purposes, it can be proven based on other axioms generally taken in number theory. Intuitively, an inductive proof is like climbing a ladder. The first step is like getting on the first rung. Every other step is merely a movement from one rung to the next. Thus you prove two things: you can get on the first rung, and, supposing that you reach the n th rung at some point, you can move to the $n + 1$ st rung. These two facts taken together prove that you eventually can reach any rung.

Recall the dialogue between the doubter and the prover in Chapter 9. In this case, the prover is claiming $I(n)$ for all n . When the doubter challenges this, the prover says, "Very well, you pick an n , and I will show it works." When the doubter has picked n , the prover then proves $I(0)$; then, using $I(0)$ as a premise, proves $I(1)$; uses $I(1)$ to prove $I(2)$; and so forth, until $I(n)$ is proven. However, all of these steps except the first are identical. A proof using mathematical induction provides a recipe for generating a proof up to any n , should the doubter be so stubborn.

Therefore, a proof by induction has two parts, a *base case* and an *inductive case*. For clarity, you should label these in your proof. Thus the format of a proof by induction should be

Proof. *Reformulate the theorem as a predicate with domain \mathbb{W} . By induction on n .*

Base case. *Suppose some proposition equivalent to $n = 0$ Hence $I(0)$. Moreover $\exists N \geq 0$ such that $I(N)$.*

Inductive case. *Suppose some proposition equivalent to $n = N + 1$ Hence $I(N + 1)$.*

Therefore, by math induction, $I(n)$ for all $n \in \mathbb{W}$ and the original proposition. \square

base case
inductive case

You may notice that with our formulation of the axiom, the proposition $\exists N \geq 0$ such that $I(N)$ is technically unnecessary. We will generally add it for clarity.

15.4 Induction gone awry

When being exposed to mathematical induction for the first time, many students are initially incredulous that it really works. This may be because students imagine an incorrect version of math induction being used to prove manifestly false propositions. Consider the following “proof” that all cows have the same color¹, and try to determine where the error is. This will test how well you have understood the principle, and, we hope, convince you that the problem lies somewhere besides the principle itself.

Proof. Let the predicate $I(n)$ be “for any set of n cows, every cow in that set has the same color.” We will prove $I(n)$ for all $n \geq 1$ by induction on n .

Base case. Suppose we have a set of one cow. Since that cow is the only cow in the set, it obviously has the same color as itself. Thus all the cows in that set have the same color. Hence $I(1)$. Moreover, $\exists N \geq 1$ such that $I(N)$.

Inductive case. Now, suppose we have a set, C , of $N + 1$ cows. Pick any cow, $c_1 \in C$. The set $C - \{c_1\}$ has N cows, by Exercise 10 of Chapter 12. Moreover, by $I(N)$, all cows in the set $C - \{c_1\}$ have the same color. Now pick another cow $c_2 \in C$, where $c_2 \neq c_1$. We know c_2 must exist because $|C| = N + 1 \geq 1 + 1 = 2$. By reasoning similar to that above, all cows in the set $C - \{c_2\}$ must have the same color.

Now, $c_1 \in C - \{c_2\}$, and so c_1 must have the same color as the rest of the set (that is, besides c_2). Similarly, $c_2 \in C - \{c_1\}$, so c_2 must have the same color of the rest of the set. Hence c_1 and c_2 also have the same color as each other, and so all cows of C have the same color.

Therefore, by math induction, $I(n)$ for all n , and all cows of any sized set have the same color. \square

The error lurking here is the conclusion that c_1 and c_2 have the same color just because they each have the same color as “the rest of C .” Since we had previously proven only $I(1)$, we cannot assume that C has any more than two elements—so it could be that $C = \{c_1, c_2\}$. In that case, $\{c_1\}$ and $\{c_2\}$ each have cows all of the same color, but that does not relate c_1 to c_2 or prove anything about the color of the cows of C .

The problem is not induction at all, but a faulty implicit assumption that C has size at least three and that $I(2)$ is true. If we had proven $I(2)$ (which of course is ridiculous), then we could indeed prove $I(n)$ for all n . Suppose $C = \{c_1, c_2, c_3\}$. Take c_1 out, leaving $\{c_2, c_3\}$. By $I(2)$, c_2 and c_3 have the same color. Put c_1 back in, take c_2 out, leaving $\{c_1, c_3\}$. By $I(2)$ again, c_1 and c_3 have the same color. By “transitivity of color,” c_1 and c_2 have the same color. Hence $I(3)$, and by induction $I(n)$ for all n . One false assumption can prove anything.

15.5 Example

In Exercises 12 and 13 of Chapter 11, you proved what are known as DeMorgan’s laws for sets (compare them with DeMorgan’s laws from Chapter 5). We can generalize those rules for unions and intersections of more than two sets. First, we define the *iterated union* and *iterated intersection* of the collection of sets A_1, A_2, \dots, A_n :

$$\bigcup_{i=1}^n A_i = A_1 \cup A_2 \cup \dots \cup A_n$$

iterated union

iterated intersection

¹The original formulation, by George Pólya, was a proof that “any n girls have eyes of the same color”[11]. That was in 1954. Cows are not as interesting, but they are a more appropriate subject, given modern sensitivities.

$$\bigcap_{i=1}^n A_i = A_1 \cap A_2 \cap \dots \cap A_n$$

Now we can prove a generalized DeMorgan's law.

Theorem 15.2 $\overline{\bigcup_{i=1}^n A_i} = \bigcap_{i=1}^n \overline{A_i}$ for all $n \in \mathbb{N}$.

Proof. By induction on n .

Base case. Suppose $n = 1$, and suppose A_1 is a (collection of one) set. Then, by definition of iterated union and iterated intersection, $\overline{\bigcup_{i=1}^1 A_i} = \overline{A_1} = \bigcap_{i=1}^1 \overline{A_i}$. Hence there

exists some $N \geq 1$ such that $\overline{\bigcup_{i=1}^N A_i} = \bigcap_{i=1}^N \overline{A_i}$.

Inductive case. Now, suppose A_1, A_2, \dots, A_{N+1} is a collection of $N + 1$ sets. Then

$$\begin{aligned} \overline{\bigcup_{i=1}^{N+1} A_i} &= \overline{A_1 \cup A_2 \cup \dots \cup A_N \cup A_{N+1}} && \text{by definition of iterated union} \\ &= \overline{\left(\bigcup_{i=1}^N A_i \right) \cup A_{N+1}} && \text{also by definition of iterated union} \\ &= \overline{\bigcup_{i=1}^N A_i \cap \overline{A_{N+1}}} && \text{by Exercise 12 of Chapter 11} \\ &= \overline{\left(\bigcap_{i=1}^N \overline{A_i} \right) \cap \overline{A_{N+1}}} && \text{by the inductive hypothesis} \\ &= \overline{\left(\overline{A_1 \cap A_2 \cap \dots \cap A_N} \right) \cap \overline{A_{N+1}}} && \text{by definition of iterated intersection} \\ &= \overline{\bigcap_{i=1}^{N+1} \overline{A_i}} && \text{also by definition of iterated intersection.} \end{aligned}$$

Therefore, by math induction, for all $n \in \mathbb{N}$ and for all collections of sets A_1, A_2, \dots, A_n ,

$$\overline{\bigcup_{i=1}^n A_i} = \bigcap_{i=1}^n \overline{A_i}. \quad \square$$

Notice that in this proof we never gave a name to the predicate (for example, $I(n)$). This means

we could not say “since $I(N)$ ” to justify that $\left(\bigcap_{i=1}^N \overline{A_i} \right) \cap \overline{A_{N+1}}$ is equivalent to the previous expressions.

inductive hypothesis

Instead, we used the term *inductive hypothesis*, which refers to our supposition that the predicate is true for N . Notice also that we never make this supposition explicitly with the word “suppose.” We make it implicitly when we say that it is true for *some* N —we have proven that some N exists (1 at least), but we are still supposing an arbitrary number, and calling it N . For your first couple of tries at math induction, you will probably find it easier to get it right if you name and use the predicate explicitly. Once you get the hang of it, you will probably find the way we wrote the preceding proof more concise.

Exercises

Prove using mathematical induction, for all $n \in \mathbb{N}$.

$$1. \overline{\bigcap_{i=1}^n A_i} = \bigcup_{i=1}^n \overline{A_i}.$$

$$2. \bigcup_{i=1}^n (A \cap B_i) = A \cap \left(\bigcup_{i=1}^n B_i \right).$$

$$3. \bigcup_{i=1}^n (A_i - B) = \left(\bigcup_{i=1}^n A_i \right) - B.$$

$$4. \bigcap_{i=1}^n (A_i - B) = \left(\bigcap_{i=1}^n A_i \right) - B.$$

5. A *summation* is an iterated addition, defined by $\sum_{i=1}^n a_i = a_1 + a_2 + \dots + a_n$ for some formula a_i , depending on i . Using math induction, prove $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ for all $n \in \mathbb{N}$.

6. We say that an integer a is divisible by $b \neq 0$ (or b divides a), written $b|a$, if there exists an integer c such that $a = cb$. Prove that for all $x \in \mathbb{Z}$ and for all $n \in \mathbb{W}$, $x - 1 | x^n - 1$. Hint: first suppose $x \in \mathbb{Z}$. Then use math induction on n . If you do not see how it works at first, pick a specific value for x , say 3, and try it for $n = 0, n = 1, n = 2$, and $n = 3$. Notice the pattern, and then use math induction to prove it for all n , but still assuming $x = 3$. Then rewrite the proof for an arbitrary x .

Chapter 16

Correctness of algorithms

16.1 Defining correctness

This program computes an arithmetic series from 1 to a given N , $\sum_{i=1}^N i$:

```
- fun arithSum(N) =  
= let  
=   val s = ref 0;  
=   val i = ref 1;  
=   in  
=     (while !i <= N do  
=       (s := !s + !i;  
=        i := !i + 1);  
=       !s)  
=   end;
```

But how do we know if this is correct? Or what does it even mean for a program to be correct? Intuitively, a program is correct if, given a certain input, it will produce a certain, desired output. In this case, if the input N is an integer (something the ML interpreter will enforce), then it should evaluate to the desired sum. When a programmer is testing software, he or she runs the program on several inputs chosen to represent the full variety of the input range and compares the result of the program to the expected result. Obviously this approach to correctness is based on experimentation; it is inductive reasoning in the non-mathematical sense, and while it can increase confidence in a program's correctness, it cannot prove correctness absolutely (except in the unrealistic case that every possible input is tested—a proof by exhaustion over the set of possible input). This process also assumes the intended result of the program can be verified conveniently by hand (or by another program)—and if that was truly convenient, we may not have written the program in the first place.

Let us consider this notion of correctness more formally. The correctness of a given program is defined by two sets of propositions: *pre-conditions*, which we expect to be true before the program is evaluated, and *post-conditions* which we expect to hold afterwards. If the post-conditions hold whenever the pre-conditions are met, then we say that the algorithm is correct. This approach is particularly useful in that it scales down to apply to smaller portions of an algorithm; Suppose we have the pre-condition *a is a nonnegative integer* for the statement

pre-conditions

post-conditions

```
b := !a + 1
```

We can think of many plausible post-conditions for this, including *b is a positive integer*, $b > a$, and $b - a = 1$. Whichever of these makes sense, they are things we can prove mathematically, as long as we implicitly take the efficacy of the assignment statement as axiomatic; propositions deduced that way are justified as *by assignment*.

Moreover, the post-conditions of a single statement are then the pre-conditions of the following statement; the pre-conditions of the entire program are the pre-conditions of the first statement; and the post-conditions of the final statement are the post-conditions of the entire program. Thus by inspecting how each statement in turn affects the propositions in the pre- and post-conditions, we can prove an algorithm is correct. Consider this program for computing $a \bmod b$ (assuming we can still use `div`).

```
- fun remainder(a, b) =
=   let
=     val q = a div b;
=     val p = q * b;
=     val r = a - p;
=   in
=     r
=   end;
```

Sometimes we will consider val-declarations merely as setting up initial pre-conditions; since they do all the work of computation in this example, we will consider them statements to be inspected. The pre-condition for the entire program is that $a, b \in \mathbb{Z}^+$. The result of the program should be $a \bmod b$; equivalently, the post-condition of the declarations is $r = a \bmod b$. To analyze what `a div b` does, we rely on the following standard result from number theory, a proof for which can be found in any traditional discrete math text.

Theorem 16.1 (Quotient-Remainder.) *If $n \in \mathbb{Z}$ and $d \in \mathbb{Z}^+$, then there exist unique integers q and r such that $n = d \cdot q + r$ and $0 \leq r < d$.*

This theorem is the basis for our definition of division and modulus. q in the above theorem is called the *quotient*, and $a \text{ div } b = q$. r is called the *remainder*, and $a \bmod b = r$.

quotient
remainder

As a first pass, we intersperse the algorithm with little proofs.

<pre>val q = a div b</pre>	<p>Suppose $a, b \in \mathbb{Z}$.</p> <p>By assignment, $q = a \text{ div } b$. By the Quotient-Remainder Theorem and the definition of division, $a = b \cdot q + R$ for some $R \in \mathbb{Z}$, where $0 \leq R < b$. By algebra, $q = \frac{a-R}{b}$.</p>
<pre>val p = q * b</pre>	<p>$p = q \cdot b$ by assignment. $p = a - R$ by substitution and algebra.</p>
<pre>val r = a - p</pre>	<p>By assignment, $r = a - p$. By substitution and algebra, $r = a - (a - R) = R$. Therefore, by the definition of <code>mod</code>, $r = a \bmod b$. \square</p>

16.2 Loop invariants

The previous example is simple almost to deception. It ignores all things that make an algorithm difficult to reason about: variables that change value, branching, and repetition. We will discover how to reason about the mutation of reference variables and the evaluation of conditionals along the way. We now consider loops, which are more difficult.

Simplistically, a loop can be analyzed for correctness by unrolling it. If a loop were to be executed, say, five times, then we can prove correct the program that would result if we pasted the body of the loop end to end five times. The problem with that approach is that almost always the number of iterations of the loop itself depends on the input, as we see in the arithmetic series example, the

guard is $i \leq N$. (An *iteration* is an execution of the body of a loop; the boolean expression which we test to see if the loop should continue is called the *guard*). We want to prove some proposition to be true for an arbitrary number of iterations. The number of iterations is certainly a whole number. This suggests a proof by induction on the number of iterations.

iteration

guard

We need a predicate, then, whose argument is the number N of iterations, and we need to show it to be true for all $N \geq 0$. This is asking for a lot of flexibility from this predicate: since it must be true for 0 iterations, it is the pre-condition for the entire loop. Since it must be true for N iterations, it is the post-condition for the entire loop. Since it must be true for every value between 0 and N , it is the post-condition and pre-condition for every iteration along the way. Of course, if a statement or code section has identical pre-conditions and post-conditions, that suggests the code does not do anything. That is not what is in view here—these various pre- and post-conditions are not identical to each other, but rather are parameterized. We must formulate this predicate in such a way that the parameterization captures what the loop does. The predicate must state *what the loop does not change*, with respect to the number of iterations.

A *loop invariant* $I(n)$ is a predicate whose argument, n , is the number of iterations of the loop, chosen so that

loop invariant

- $I(0)$ is true (that is, the proposition is true before the loop starts; this must be proven as the base case in the proof).
- $I(n)$ implies $I(n + 1)$ (that is, if the proposition is true before a given iteration, it will still be true after that iteration; this must be proven as the inductive case in the proof).
- If the loop terminates (after, say, N iterations), then $I(N)$ is true (this follows from the two previous facts and the principle of math induction).
- Also if the the loop terminates, then $I(N)$ implies the post-condition of the entire loop.

These four points correspond to four steps in the proof that a loop is correct: we prove that the loop is *initialized* so as to establish the loop invariant; that a given iteration *maintains* the loop invariant; that the loop will eventually *terminate*, that is, that the guard will be false eventually; and that the loop invariant implies the post-condition. The first two steps constitute a proof by induction, on which we focus. The last two complete the proof of algorithm correctness.

initialization

maintenance

termination

16.3 Big example

Let us now apply this intention to the algorithm given at the beginning of this chapter. The post-condition for the entire algorithm is that the variable s should equal the value of the series. Since s is a reference variable, it changes value during the running of the program. Specifically, after n iterations, $s = \sum_{k=1}^n k$. (Why did we replace N and i with n and k ?) It will also be helpful to monitor what the variable i is doing. Since i is initialized to 1 and is incremented by 1 during each iteration, then with respect to n , $i = n + 1$. Thus we have our loop invariant:

$$I(n) = \text{after } n \text{ iterations, } i = n + 1 \text{ and } s = \sum_{k=1}^n k$$

Now we prove

Theorem 16.2 For all $N \in \mathbb{W}$, the program `arithSum` computes $\sum_{i=1}^N i$.

Proof. Suppose $N \in \mathbb{W}$.

Base case / initialization. After 0 iterations, $i = 1$ and $s = 0 = \sum_{k=1}^0 k$ by assignment and the definition of summation. Hence $I(0)$, and so there exists an $n' \geq 0$ such that $I(n')$.

Inductive case / maintenance. Suppose $I(n')$. Let i_{old} be the value of i after the n' th iteration and before the $n' + 1$ st iteration, and let i_{new} be the value of i after the $n' + 1$ st iteration. Similarly define s_{old} and s_{new} . By $I(n')$, $i_{\text{old}} = n' + 1$ and $s_{\text{old}} = \sum_{k=1}^{n'} k$.

By assignment and substitution, $s_{\text{new}} = s_{\text{old}} + i_{\text{old}} = \sum_{k=1}^{n'} k + (n' + 1) = \sum_{k=1}^{n'+1} k$. Similarly $i_{\text{new}} = i_{\text{old}} + 1 = (n' + 1) + 1$. Hence $I(n' + 1)$.

Hence by math induction, $I(n)$ for all $n \in \mathbb{W}$, and so $I(N)$.

Before we continue, be attentive to the subtle matter of our choice of variables. What is with the N , the n , and the n' ? N is the input to the *program* `arithSum`. n is the argument to (or, independent variable of) the *predicate* $I(n)$ used to analyze `arithSum`. One thing we are trying to prove is $I(N)$ (I is true for $n = N$) in the imaginary world created by supposing N . n' is an arbitrary whole number such that $I(n')$, used inside of our inductive proof that $I(n)$ for all n . Properly disambiguating variables is essential for all proofs and become particularly hard in proofs of algorithm correctness when you are juggling similar variables and proving a property of an object which itself has variables. Failure to do this will lead to equivocal nonsense.

Termination By $I(N)$, after N iterations, $i = N + 1 > N$, and so the guard will fail.

Moreover, by $I(N)$, after N iterations, $s = \sum_{k=1}^N k$. By change of variable, $s = \sum_{i=1}^N i$, which is our post-condition for the program. Hence the program is correct. \square

16.4 Small example

Now consider an algorithm that processes an array. This example is “smaller” than the previous one because we will concern ourselves only with the proof of the loop invariant. This program finds the smallest element in an array.

```
- fun findMin(array) =
=   let
=     val min = ref (sub(array, 0));
=     val i = ref 1;
=   in
=     (while !i < length(array) do
=       (if sub(array, !i) < !min
=         then min := sub(array, !i)
=         else ();
=       i := !i + 1);
=     !min)
=   end;
```

As you will see by inspecting the code, it finds the minimum by considering each element in the array in order. The variable *min* is used to store the “smallest seen so far.” Every time we see an element smaller than the smallest so far (`sub(array, !i) < !min`), we make that element the new smallest so far (`then min := sub(array, !i)`), and otherwise do nothing (`else ()`). At the end of the loop, the “smallest so far” is also the “smallest overall.”

Thus our loop invariant must somehow capture the notion of “smallest so far.” To make this more convenient, we introduce the notation $A[i..j]$, which will stand for the set of values that are elements of the *subarray* of A in the range from i inclusive to j exclusive. That is,

subarray

$$A[i..j] = \{A[k] \mid i \leq k < j\}$$

We say that x is the minimum element in a subset A of integers if $x \in A$ and $\forall y \in A, x \leq y$. Now, our post-condition for the loop is that min is the minimum of $\text{array}[0..N]$, where N is the size of array . Our (proposed) loop invariant is

$$I(n) = \text{after } n \text{ iterations, } i = n + 1 \text{ and } \text{min} \text{ is the minimum of } \text{array}[0..i]$$

We now prove it to be a loop invariant.

Proof. Suppose array is an array of integers.

Base case / initialization. After 0 iterations, $\text{min} = \text{array}[0]$ and $i = 1 = 0 + 1$ by assignment. By the definition of subarray, $\text{array}[0]$ is the only element in $\text{array}[0..1] = \text{array}[0..i]$, and so min is the minimum of $\text{array}[0..i]$ by definition of minimum. Hence $I(0)$, and there exists $n' \geq 0$ such that $I(n')$.

Inductive case / maintenance. Suppose $I(n')$. Let min_{old} be the value of min after the n' th iteration and before the $n' + 1$ st iteration, and let min_{new} be the value of min after the $n' + 1$ st iteration. Similarly define i_{old} and i_{new} . By $I(n')$, min_{old} is the minimum of $\text{array}[0..i_{\text{old}}]$. We now have two cases:

Case 1: Suppose $\text{array}[i_{\text{old}}] < \text{min}_{\text{old}}$. Then, by assignment $\text{min}_{\text{new}} = \text{array}[i_{\text{old}}]$. Now, suppose $x \in \text{array}[0..(i_{\text{old}} + 1)]$. If $x = \text{array}[i_{\text{old}}]$, then $\text{min}_{\text{new}} \leq x$ trivially. Otherwise, $x \in \text{array}[0..i_{\text{old}}]$, and $\text{min}_{\text{new}} = \text{array}[i_{\text{old}}] < \text{min}_{\text{old}} \leq x$, the last step by the definition of minimum. In either case, $\text{min}_{\text{new}} \leq x$, and so min_{new} is the minimum of $\text{array}[0..(i_{\text{old}} + 1)]$ by definition of minimum.

Case 2: Suppose $\text{array}[i_{\text{old}}] \geq \text{min}_{\text{old}}$. Then $\text{min}_{\text{new}} = \text{min}_{\text{old}}$. Now, suppose $x \in \text{array}[0..(i_{\text{old}} + 1)]$. If $x = \text{array}[i_{\text{old}}]$, then $\text{min}_{\text{new}} \leq x$ by substitution. Otherwise, $x \in \text{array}[0..i_{\text{old}}]$, and $\text{min}_{\text{new}} = \text{min}_{\text{old}} \leq x$ by definition of minimum. In either case, $\text{min}_{\text{new}} \leq x$, and so min_{new} is the minimum of $\text{array}[0..(i_{\text{old}} + 1)]$ by definition of minimum.

In either case, min_{new} is the minimum of $\text{array}[0..(i_{\text{old}} + 1)]$. By assignment, $i_{\text{new}} = i_{\text{old}} + 1 = (n' + 1) + 1$ by substitution. Also by substitution, min_{new} is the minimum of $\text{array}[0..i_{\text{new}}]$. Hence $I(n' + 1)$, and $I(n)$ is an invariant for this loop. \square

Notice how the conditional requires a division into cases. Whichever way the condition branches, we end up with min being the minimum of the range so far.

Exercises

In Exercises 1–3, prove the predicates to be loop invariants for the loops in the following programs.

1. $I(n) = x$ is even.

```
- fun aaa(n) =
= let
=   val x = ref 0;
=   val i = ref 0;
= in
=   (while !i < n do
=     (x := !x + 2 * i;
=      i := !i + 1);
=     !x)
= end;
```

2. $I(n) = x + y = 100$.

```
- fun bbb(n) =
= let
=   val x = ref 50;
=   val y = ref 50;
=   val i = ref 0
= in
=   (while !i < n do
=     (x := !x + 1;
=      y := !y - 1);
=     !x + !y)
= end;
```

3. $I(n) = m + n$ is odd.

```
- fun ccc(n) =
= let
=   val x = ref 0;
=   val y = ref 101;
= in
=   (while !x < n do
=     (x := !x + 4;
=      y := !y - 2);
=     !x + !y)
= end;
```

4. Finish the proof of correctness for `findMin`. That is, show that the loop will terminate and that the loop invariant implies the post-condition.
5. Write a program which, given an array of integers, computes the sum of the elements in the array. Write a complete proof of correctness for your program: determine pre-conditions and post-conditions, determine a loop invariant, prove it to be a loop invariant, show that the loop will terminate, and show that the loop invariant implies the post-conditions.

Exercises 2 and 3 are taken from Epp [5].

Chapter 17

From Theorems to Algorithms

17.1 The Division Algorithm

Recall the Quotient-Remainder Theorem from the previous chapter:

Theorem 16.1. *If $n, d \in \mathbb{Z}^+$, then there exist unique integers q and r such that $n = d \cdot q + r$ and $0 \leq r < d$.*

Notice that this result tells us of the existence of something. A proof for such a proposition must either show that it is impossible for the item not to exist (by a proof by contradiction) or give an algorithm for calculating the item. The latter is called a *constructive proof*, and it in fact gives more information than merely the truth of the proposition. The traditional proof of the QRT is non-constructive.

However, the theorem itself, if studied carefully, tells us much about how to build an algorithm for finding the quotient and the remainder, called the *Division Algorithm*. In the previous chapter, we proved results (specifically, correctness) about algorithms. In this chapter, we derive algorithms from results in a process that is essentially the reverse of that of the previous chapter.

Division Algorithm

Consider what the QRT says about q and r . The two assertions (the propositions subordinate to “such that”) can be thought of as restrictions that must be met as we try to find suitable values. We have:

- $n = d \cdot q + r$.
- $r \geq 0$.
- $r < d$.

The restrictions $n = d \cdot q + r$ and $r \geq 0$ are not difficult to satisfy; simply take $q = 0$ and $r = n$ —call this our initial guess. That guess, however, might conflict with the other restriction, $r < d$. Our general strategy, then, is to alter our initial guess, making sure that any changes we make do not violate the earlier restriction, and repeating until we satisfy the other restriction, too.

Note the elements of our algorithm sketch, which you should recognize from studying correctness proofs from the previous chapter. We have:

- Initial conditions: $q = 0$ and $r = n$.
- A loop invariant: $n = d \cdot q + r$.
- A termination condition: $r < d$.

In ML,

```

- fun divisionAlg(n, d) =
=   let
=     val r = ref n;
=     val q = ref 0;
=   in
=     (while !r >= d do
=       ();
=       (!q, !r))
=   end;

```

Notice that we return the quotient and the remainder as a tuple. The only thing missing is the body of the loop—that is, how to mutate q and r so as to preserve the invariant and to make progress towards the termination condition. It may seem backwards to determine the husk of a loop before its body or to set up a proof of correctness before writing what is to be proven correct. However, this way of thinking encourages programming that is amenable to correctness proof and enforces a discipline of programming and proving in parallel.

Since the termination condition is $r < d$, progress is made by making r smaller. If the termination condition does not hold, then $r \geq d$, and so there is some nonnegative integer, say y , such that $r = d + y$. Note that

$$\begin{aligned}
 n &= d \cdot q + r \\
 &= d \cdot q + d + y \\
 &= d \cdot (q + 1) + y
 \end{aligned}$$

Compare the form of the expression at the bottom of the right column with that at the top. As we did in the previous chapter, we will reckon the affect of the change to r by distinguishing r_{old} from r_{new} , and similarly for q . Then let

$$\begin{aligned}
 r_{\text{new}} &= y &= r_{\text{old}} - d \\
 q_{\text{new}} &= q_{\text{old}} + 1
 \end{aligned}$$

In other words, decrease r by d and increase q by one. This preserves the loop invariant (by substitution, $n = d \cdot q_{\text{new}} + r_{\text{old}}$) and reduces r .

```

- fun divisionAlg(n, d) =
=   let
=     val r = ref n;
=     val q = ref 0;
=   in
=     (while !r >= d do
=       (r := !r - d;
=        q := !q + 1);
=       (!q, !r))
=   end;

```

We have a correctness proof ready-made. Furthermore, the correctness of the algorithm proves the theorem (the existence part of it, anyway), since a way to compute the numbers proves that they exist.

17.2 The Euclidean Algorithm

greatest common divisor The *greatest common divisor* (GCD) of two integers a and b is a positive integer d such that

- $d|a$ and $d|b$ (that is, d is a common divisor).
- for all c such that $c|a$ and $c|b$, $c|d$ (that is, d is the greatest of all common divisors).

Recall from the definition of divides that the first item means that there exist integers p and q such that $a = p \cdot d$ and $b = q \cdot d$. The second item says that any other divisor of a and b must also be a divisor of d . The GCD of a and b , commonly written $\gcd(a, b)$, is most recognized for its use in simplifying fractions. Clearly a simple and efficient way to compute the GCD would be useful for any application involving integers or rational numbers. Here we have two lemmas:

Lemma 17.1 *If $a \in \mathbb{Z}$, then $\gcd(a, 0) = a$.*

Lemma 17.2 *If $a, b \in \mathbb{Z}, q, r \in \mathbb{Z}^{\text{nonneg}}$, and $a = b \cdot q + r$, then $\gcd(a, b) = \gcd(b, r)$.*

The proof of Lemma 17.1 is left as an exercise. For the proof of Lemma 17.2, consult a traditional discrete math text. We now follow the same steps as in the previous section

The fact that we have two lemmas of course makes this more complicated. However, we can simplify this by unifying their conclusions. They both provide equivalent expressions for $\gcd(a, b)$, except that Lemma 17.1 addresses the special case of $b = 0$. From this we have the intuition that the loop invariant will probably involve the GCD.

Now consider how the lemmas differ. The equivalent Lemma 17.2 gives is simply a new GCD problem; Lemma 17.1, on the other hand, gives a final answer. Thus we have our termination condition: $b = 0$. Lemma 17.2 then helps us distinguish between what does not change and what does: The parameters to the GCD change; the answer does not. If we distinguish the changing variables a and b from their original values a_0 and b_0 , we have

- Initial conditions: $a = a_0$ and $b = b_0$.
- Loop invariant: $\gcd(a, b) = \gcd(a_0, b_0)$.
- Termination condition: $b = 0$.

All that remains is how to mutate the variables a and b . a takes on the value of the old b . b takes on the value of r from Lemma 17.2.

$$\begin{aligned} a_{\text{new}} &= b_{\text{old}} \\ b_{\text{new}} &= r = a_{\text{old}} \bmod b_{\text{old}} \end{aligned}$$

This is known as the *Euclidean Algorithm*, though it was probably known before Euclid.

Euclidean Algorithm

```
- fun gcd(a0, b0) =
=   let
=     val a = ref a0;
=     val b = ref b0;
=   in
=     (while !b <> 0 do
=       (a := !b;
=        b := !a mod !b);
=     !a)
=   end;

val gcd = fn : int * int -> int

- gcd(21,36);

val it = 36 : int
```

36 clearly is not the GCD of 21 and 36. What went wrong? Our proof-amenable approach has lulled us into a false sense of security, but it also will help us identify the problem speedily. The line $b := !a \bmod !b$ is not $b_{\text{new}} = a_{\text{old}} \bmod b_{\text{old}}$ but rather $b_{\text{new}} = a_{\text{new}} \bmod b_{\text{old}}$. We must calculate r before we change a . This can be done with a let expression:

```

- fun gcd(a0, b0) =
=   let
=     val a = ref a0;
=     val b = ref b0;
=   in
=     (while !b <> 0 do
=       let val r = !a mod !b
=       in
=         (a := !b;
=          b := r)
=       end;
=     !a)
=   end;

```

17.3 The Euclidean Algorithm, another way

Considering how we unified Lemmas 17.1 and 17.2, we could have written them as one lemma:

Lemma 17.1–17.2. *If $a, b \in \mathbb{Z}$, then*

$$\gcd(a, b) = \begin{cases} a & \text{if } b = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

Not only is this more concise (partly because we employed the `mod` operation guaranteed by the QRT), but it also expresses the result in an if-then-else format as we might find in an algorithm. Could we use this in constructing an ML function? We have seen predicates and functions that are defined in terms of other predicates and functions, but this is different because the GCD is defined in terms of itself. This invites us to do some wishful thinking: If only the function `gcd` were already defined, then defining it would be so much easier. ML, like most programming languages, allows this sort of wishing:

```

- fun gcd(a, b) = if b = 0 then a else gcd(b, a mod b);

```

Or, using pattern-matching:

```

- fun gcd(a, 0) = a
=   | gcd(a, b) = gcd(b, a mod b);

```

The loop disappears completely. Instead of effecting repetition by a command to iterate (as with a `while` statement), repetition happens implicitly by the repeated calling of the function. Instead of mutating reference variables, we feed different parameters into `gcd` each time. Essentially we have

```

gcd(21, 36)
= gcd(36, 21)
= gcd(21, 15)
= gcd(15, 6)
= gcd(6, 3)
= gcd(3, 0)
= 3

```

recursion

This interaction of a function with itself is called *recursion*, which means self-reference. A simple recursive function has a conditional (or patterns) separating into two cases: a base case, a point at which the recursion stops; and a recursive case, which involves a recursive function call. Notice how the termination condition of the iterative version corresponds to the base case of the recursive version. Also notice the similarity of structure between inductive proofs and recursive functions.

What is most striking is how much more concise the recursive version is. Recursive solutions do tend to be compact and elegant. Something else, however, is also at work here; ML is intended for a programming style that makes heavy use of recursion. More generally, this style is called the *functional* or *applicative* style of programming (as opposed to the *iterative* style, which uses loops), *functional programming* where the central concept of building algorithms is the applying of functions, as opposed to the repeating of loop bodies or the stringing together of statements. This style will be taught for the *applicative style* remainder of this course.

Exercises

1. Write a complete proof of correctness for the function `divisionAlg`.
2. Recall that `divisionAlg` returns a tuple standing for the quotient and the remainder together. Write functions `quot` and `remain` that return only the quotient and remainder, respectively (so they are equivalent to ML's `div` and `mod`). Do not re-type the algorithm for each one; instead, enter it once as we did here and have `remain` and `quot` call `divisionAlg`.
3. Prove Lemma 17.1.

Exercises 4–9 are developed from Abelson and Sussman, *Structure and Interpretation of Computer Programs*, McGraw Hill, 1996, pages 46–47.

4. The following lemmas give no particularly surprising result.

Lemma 17.3 If $a, b \in \mathbb{Z}$, then $a \cdot b^0 = a$.

Lemma 17.4 If $a, b, n \in \mathbb{Z}$, then $a \cdot b^n = (a \cdot b)b^{n-1}$.

However, they can be used to generate an algorithm for exponentiation using repeated multiplication. Write an ML function for such an algorithm, to compute x^y . (Hint: If initially $a = 1$, $b = x$, and $n = y$, then $a \cdot b^n = x^y$, which is also your invariant. Your postcondition is that $a = x^y$, $b = x$, and $n = 0$.)

5. The algorithm you used in Exercise 4 required n multiplication operations. Write a function implementing a faster version of this algorithm by making use of the following additional lemma.

Lemma 17.5 If $a, b, n \in \mathbb{Z}$ and n is even, then $a \cdot b^n = a(b^2)^{\frac{n}{2}}$.

(If you have done this correctly, your function/algorithm should require about $\log_2 n$ multiplications. Can you tell why?)

6. In a similar way, use the following three lemmas to write a function that computes $x \cdot y$ without using multiplication. (To make it fast, you will need to use division, but only dividing by 2.)

Lemma 17.6 If $a, b \in \mathbb{Z}$, then $a + b \cdot 0 = a$.

Lemma 17.7 If $a, b, c \in \mathbb{Z}$, then $a + b \cdot c = a + b + b \cdot (c - 1)$.

Lemma 17.8 If $a, b, c \in \mathbb{Z}$ and c is even, then $a + b \cdot c = a + (b + b)(c \div 2)$.

7. Write a recursive version of your function from Exercise 4.
8. Write a recursive version of your function from Exercise 5.
9. Write a recursive version of your function from Exercise 6.
10. Write a recursive version of `divisionAlg`.

Chapter 18

Recursive algorithms

In the previous chapter we were introduced to recursive functions. We will explore this further here, with a focus on processing sets represented in ML as lists, and also on list processing in general. This focus is chosen not only because these applications are amenable to recursion, but also because of the centrality of sets to discrete mathematics and of lists to functional programming. We divide our discussion into two sections: functions that take lists (and possibly other things as well) as arguments and compute something about those lists; and functions that compute lists themselves.

18.1 Analysis

Suppose we are modeling the courses that make up a typical mathematics or computer science program, and we wish to analyze the requirements, overlap, etc. Datatypes are good for modeling a universal set.

```
- datatype Course = Calculus | Discrete | Programming | LinearAlg
=                  | DiffEq | OperSys | RealAnalysis | ModernAlg | Stats
=                  | Algorithms | Compilers;
```

The most basic set operation is \in , set inclusion. We call it an “operation,” though it can be just as easily be thought of as a predicate, say `isElementOf`. (In Part V, we will see that it is also a relation.) Set inclusion is very difficult to describe formally, in part because sets are unordered. However, any structure we use to represent sets must impose some incidental order to the elements. Computing whether a set contains an element obviously requires searching the set to look for the element, and the incidental ordering must guide our search. If we represented a set using an array, we might write a loop which iterates over the array and update a `bool` variable to keep track of whether we have found the desired item. Compare this with `findMin` from Chapter 16:

```
- fun isElementOf(x, array) =
=   let
=     val len = length(array);
=     val i = ref 0;
=     val found = ref false;
=   in
=     (while not (!found) andalso !i < len do
=       (if sub(array, !i) = x then found := true else ();
=        i := !i + 1);
=     !found)
=   end;

val isElementOf = fn : ''a * ''a array -> bool
```

```

- csCourses;

val it = [|Discrete,Programming,OperSys,Algorithms,Compilers|] : Course array

- isElementOf(OperSys,csCourses);

val it = true : bool

```

(This function would give the same result is we had left out `not (!found)` `andalso`. Why did we include that extra condition?)

Arrays are a cumbersome way to represent sets, and as we will see in the next section, completely useless when we want to derive new sets from old. It is the fixed size and random access of arrays that make them so inconvenient, and this is why lists have been and continue to be our preferred representation of sets. We have a pattern for writing an array algorithm which asks, How does this algorithm on the entire array break down to a step to be taken on each position? The answer to this question become the body of the loop. We must develop a corresponding strategy for lists.

It is always easiest to start with the trivial. If a set is empty, no item is an element of it. Thus we can say

```
- fun isElementOf(x, []) = false
```

The almost-trivial case is if by luck the element we are looking for is the head of the list.

```
- fun isElementOf(x, []) = false
  = | isElementOf(x, a) = if x = hd(a) then true
```

or

```
- fun isElementOf(x, []) = false
  = | isElementOf(x, y::rest) = if x = y then true
```

What if the element is somewhere else in the list? Or what if it is not in the list, but the list is not empty? Both of those questions are subsumed by asking, Is the element in `rest` or not?

```
- fun isElementOf(x, []) = false
  = | isElementOf(x, y::rest) = if x = y then true else isElementOf(x, rest);
```

Do not fail to marvel at the succinctness of this recursive algorithm on a list, contrasted with the iterative algorithm on an array. The key insight is that after checking if the list is empty and whether the first item is what we are looking for, we have reduced the problem (if we have not solved it immediately) to a new problem, identical in structure to the original, but smaller. This is the heart of thinking recursively. It works so well on lists because lists themselves are defined recursively; a list is

- An empty list, or
- an item followed by a list.

Hence our pattern for list operations is

- What do I do with an empty list?
- Given a non-empty list
 - What do I do with the first element?
 - What do I do with the rest of the list?

You will learn to recognize that the answer to the last sub-question is also the solution to the entire problem.

Let us apply this now to a new problem: computing the cardinality of a set. The type of the function `cardinality` will be `'a list -> int`. If a set is empty, then its cardinality is zero. Otherwise, if a set A contains at least one element, x , then we can note

$$A = \{x\} \cup (A - \{x\})$$

and since $\{x\}$ and $A - \{x\}$ are disjoint,

$$|A| = |\{x\}| \cup |A - \{x\}|$$

and so

```
- fun cardinality([]) = 0
=   | cardinality(x::rest) = 1 + cardinality(rest);

val cardinality = fn : 'a list -> int

- val csCourses = [Discrete, Programming, OperSys, Algorithms, Compilers];

val csCourses = [Discrete, Programming, OperSys, Algorithms, Compilers]
  : Course list

- cardinality(csCourses);

val it = 5 : int
```

We can describe the recursive case into three parts. First there is the work done before the recursive call of the function; when working with lists, this work is usually the splitting of the list into its head and tail, which can be done implicitly with pattern matching, as we have done here. Second, there is the recursive call itself. Finally, we usually must do some work after the call, in this case accounting for the first element by adding one to the result of the recursive call.

18.2 Synthesis

Now we consider functions that will construct lists. One drawback of using lists to represent sets is that lists may have duplicate items. Our set operation functions operate under the assumption that the list has been constructed so there happen to be no duplicates. We will get undesired responses if that assumption breaks.

```
- cardinality([RealAnalysis, OperSys, LinearAlg, ModernAlg, OperSys]);

val it = 5 : int
```

It would be useful to have a function that will strip duplicates out of a list, say `makeNoRepeats`. Applying the same strategy as before, first consider how to handle an empty list. Since an empty list cannot have any repeats, this is the trivial case, but keep in mind we are not computing an `int` or `bool` based on this list. Instead we are computing another list, in this case, just the empty list.

```
- fun makeNoRepeats([]) = []
```

Now, what do we do with the case of $x::\text{rest}$? The subproblem that corresponds to the entire problem is removing the duplicates from rest . It is a safe guess that our answer will include $\text{makeNoRepeats}(\text{rest})$. The leaves just x to be dealt with. If we wish to include x in our resulting list, we can use the cons operator to construct that new list, just as we use it to break down a list in pattern matching: $x::\text{makeNoRepeats}(\text{rest})$. However, we should include x in the resulting list only if it does not appear there already (otherwise it would be a duplicate); it will appear in the result of the recursive call if and only if it appears in rest (why?). Thus we have

```
- fun makeNoRepeats([]) = []
=   | makeNoRepeats(x::rest) =
=       if isElementOf(x,rest)
=       then makeNoRepeats(rest)
=       else x::makeNoRepeats(rest);

val makeNoRepeats = fn : ''a list -> ''a list

- makeNoRepeats([RealAnalysis, OperSys, LinearAlg, ModernAlg, OperSys]);

val it = [RealAnalysis,LinearAlg,ModernAlg,OperSys] : Course list
```

To test your understanding of how this works, explain why it was the first occurrence of `OperSys` that disappeared, rather than the second.

We noted in Chapter 4 that the cat operator is a poor way to perform a union on sets represented on lists because any elements in the intersection of the two lists will be included twice. Now we can write a simple union function, making use of `makeNoRepeats`:

```
- fun union(a, b) = makeNoRepeats(a @ b);
```

A final example will reemphasize the importance of types. Recall that one can make a list of any type, including other list types. Now suppose we wanted to take a list, create one-element lists of all its elements, and return a list of those lists, for example

```
- listify([RealAnalysis,Discrete,DiffEq,Compilers]);

val it = [[RealAnalysis],[Discrete],[DiffEq],[Compilers]] : Course list list
```

An empty list is still listified to an empty list. But now the work that needs to be done to x is to make a list out of it.

```
- fun listify([]) = []
=   | listify(x::rest) = [x] :: listify(rest);
```

What is interesting here is an analysis of the types of the subexpressions of the recursive case:

$$\underbrace{\underbrace{[x]}_{\text{'a list}}}_{\text{'a list}} :: \underbrace{\underbrace{\text{listify}(\text{rest})}_{\text{'a list list}}}_{\text{'a list list}}$$

Exercises

1. Write a predicate `isSubsetOf` which takes two sets represented as lists and determines if the first set is a subset of the other. Hint: make the first set vary in the pattern between empty and non-empty, but regard the second as just a generic set, as in

```
- fun isSubsetOf([], b) = ...
=   | isSubsetOf(x::rest, b) = ...
```

You may use `isElementOf` from this chapter.

2. Write a predicate `hasNoRepeats` which tests if a list has no duplicate elements.
3. Write a function `intersection` which computes the intersection of two sets.
4. Write a function `difference` which computes the difference of one set from another.
5. Write a new `union` function which does not use `makeNoRepeats` but instead uses `@`, `intersection`, and `difference`
6. Write a function `sum` which computes the sum of the elements in a list of integers.
7. Write a function `findMin` like the one in Section 16.4 except that it finds the minimum element in a list of integers. Hint: Your recursive call will result in something akin to the “minimum so far” idea we had when proving the old `findMin` correct, and you likely use that value twice. Do not write a redundant recursive call; instead, save the result to a variable using a `let` expression.
8. Write a function `addToAll` which will take an element and a list of lists and will produce a similar list of lists except with the other element prepended to all of the sub-lists, as in


```
- addToAll(5, [[1,2],[3],[],[4,5,7]]);

val it = [[5,1,2],[5,3],[5],[5,4,5,7]]
: int list list
```
9. In Chapter 15, we noted that the `powerset` function was far from the best way to compute a powerset in ML. Write a better, recursive `powerset`. Hint: use your `addToAll` from Exercise 8.

Part V

Relation

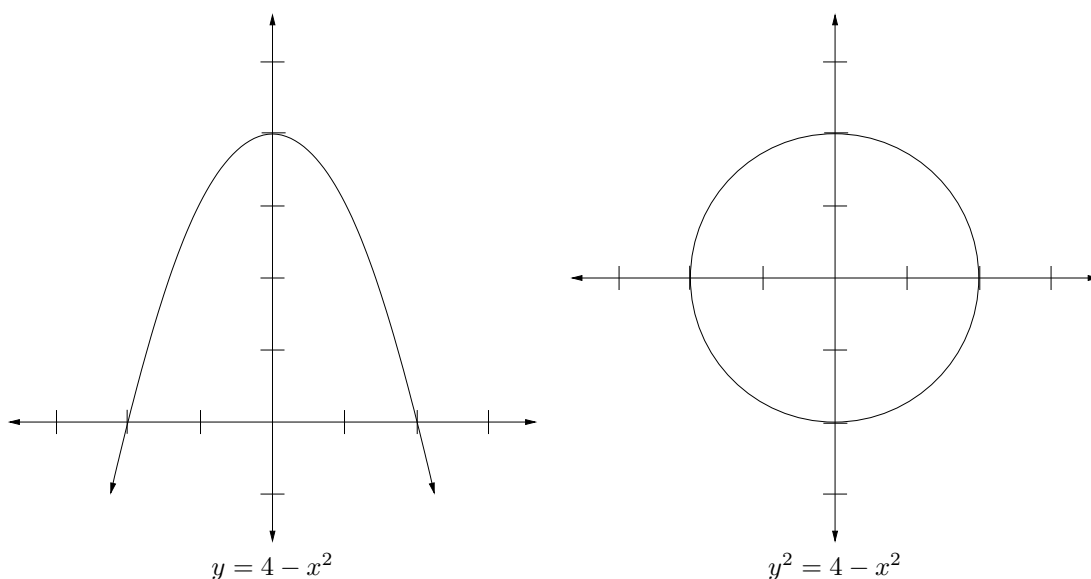
Chapter 19

Relations

19.1 Definition

Relations are not a new concept for you. Most likely, you learned about relations by how they differed from functions. We, too, will use relations as a building block for studying functions in Part VI, but we will also consider them for logical and computational interest in their own right.

If we consider curves in the real plane, the equation $y = 4 - x^2$ represents a function because its graph passes the vertical line test. A circle, like $y^2 = 4 - x^2$, fails the test and thus is not a function, but it is still a relation.



Notice that a “curve” in a graph is really a set of points; so is the equation the curve represents, for that matter. $y^2 = 4 - x^2$ can be thought of as defining the set that includes $(0, 2)$, $(2, 0)$, $(0, -2)$, $(-2, 0)$, $(\sqrt{2}, -\sqrt{2})$, $(-\sqrt{2}, \sqrt{2})$, etc. A point, in turn, is actually an ordered pair of real numbers. This leads to what you may remember as the high school definition of a relation: a set of ordered pairs. We should also recognize them as subsets of $\mathbb{R} \times \mathbb{R}$.

Our notion of relation will be more broad and technical, allowing for subsets of any Cartesian product. Thus if X and Y are sets, then a *relation* R from X to Y is a subset of $X \times Y$. If $X = Y$, we say that R is a relation on X . (Sometimes subsets on higher-ordered Cartesian products are also considered; in that context, our definition of a relation more specifically is of a *binary relation*.) If $(x, y) \in R$, we say that x is related to y . This is sometimes written xRy , especially if R is a special symbol like $|$, \in , or $=$, which, you should notice, are all relations. We can also consider R to be a

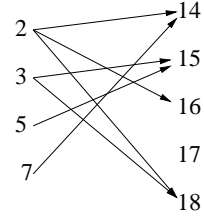
relation

predicate and write $R(x, y)$ if $(x, y) \in R$.

Small relations are visualized using *graphs*, which we will treat formally in Part VIII. Their intent should be intuitively clear for the following examples of relations.

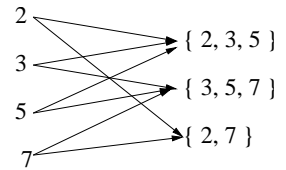
Relation $|$ (divides) from $\{2, 3, 5, 7\}$ to $\{14, 15, 16, 17, 18\}$

$\{(2, 14), (2, 16), (2, 18), (3, 15), (3, 18), (5, 15), (7, 14)\}$



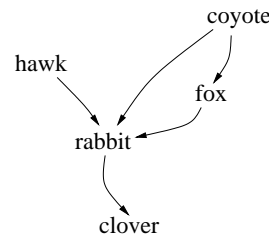
Relation \in from $\{2, 3, 5, 7\}$ to $\{\{2, 3, 5\}, \{3, 5, 7\}, \{2, 7\}\}$.

$\{(2, \{2, 3, 5\}), (2, \{2, 7\}), (3, \{2, 3, 5\}), (3, \{3, 5, 7\}), (5, \{2, 3, 5\}), (5, \{3, 5, 7\}), (7, \{3, 5, 7\}), (7, \{2, 7\})\}$



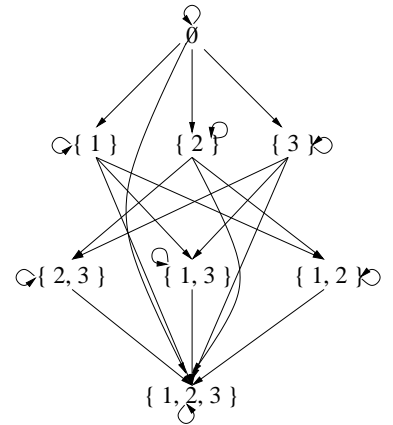
Relation eats on the set $\{\text{hawk, coyote, rabbit, fox, clover}\}$

$\{(\text{coyote, rabbit}), (\text{hawk, rabbit}), (\text{coyote, fox}), (\text{fox, rabbit}), (\text{rabbit, clover})\}$



Relation \subseteq on $\mathcal{P}(\{1, 2, 3\})$.

$\{(\emptyset, \emptyset), (\emptyset, \{1\}), (\emptyset, \{2\}), (\emptyset, \{3\}), (\emptyset, \{1, 2\}), (\emptyset, \{2, 3\}), (\emptyset, \{1, 3\}), (\emptyset, \{1, 2, 3\}), (\{1\}, \{1\}), (\{1\}, \{1, 2\}), (\{1\}, \{1, 3\}), (\{1\}, \{1, 2, 3\}), (\{2\}, \{2\}), (\{2\}, \{1, 2\}), (\{2\}, \{2, 3\}), (\{2\}, \{1, 2, 3\}), (\{3\}, \{3\}), (\{3\}, \{1, 3\}), (\{3\}, \{2, 3\}), (\{3\}, \{1, 2, 3\}), (\{1, 2\}, \{1, 2\}), (\{1, 2\}, \{1, 2, 3\}), (\{2, 3\}, \{2, 3\}), (\{2, 3\}, \{1, 2, 3\}), (\{1, 3\}, \{1, 3\}), (\{1, 3\}, \{1, 2, 3\}), (\{1, 2, 3\}, \{1, 2, 3\})\}$



self-loop

Notice that in some cases, an element may be related to itself. This is represented graphically by a *self-loop*.

19.2 Representation

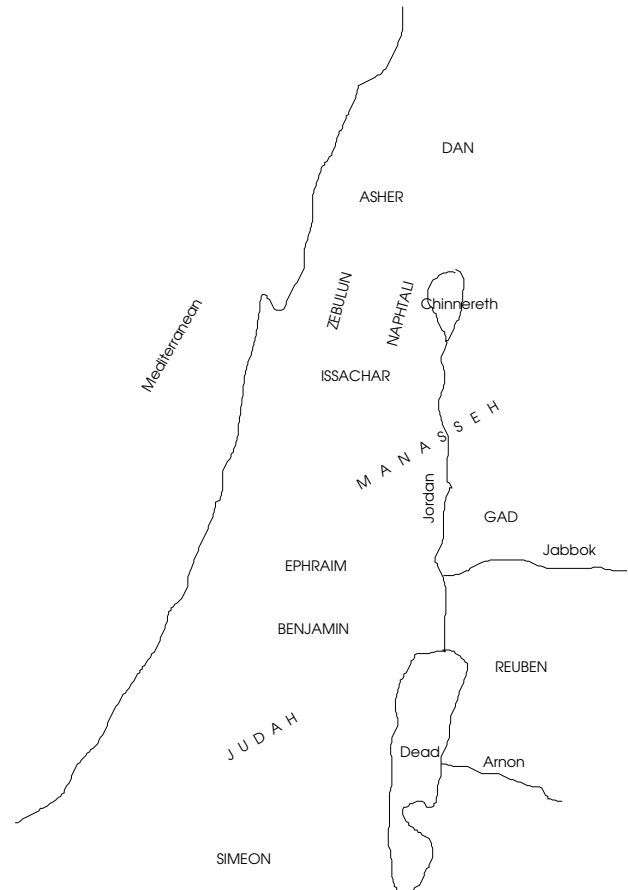
A principal goal of this course is to train you to think generally about mathematical objects. Originally, the mathematics you knew was only about numbers. Gradually you learned that mathematics can be concerned about other things as well, such as points or matrices. Our primary extension of this list has been sets. You should now be expanding your horizon to include relations as full-fledged mathematical objects, since a relation is simply a special kind of set. In ML, our concept of *value*

corresponds to what we mean when we say *mathematical object*. We now consider how to represent relations in ML.

Our running example through this part is the relations among geographic entities of Old Testament Israel. As raw material, we represent (as datatypes) the sets of tribes and bodies of water.

```
- datatype Tribe = Asher | Naphtali | Zebulun
=                | Issachar | Dan | Gad
=                | Manasseh | Reuben | Ephraim
=                | Benjamin | Judah | Simeon;

- datatype WaterBody = Mediterranean | Dead
=                    | Jordan | Chinnereth
=                    | Arnon | Jabbok;
```



Now we consider the relation from *Tribe* to *WaterBody* defined so that tribe t is related to water body w if t is bordered by w . We could represent this using a predicate, and we will later find some circumstances where a predicate is more convenient to use. Using lists, however, it is much easier to define a relation.

```
- val tribeBordersWater =
= [(Asher, Mediterranean), (Manasseh, Mediterranean), (Ephraim, Mediterranean),
=  (Naphtali, Chinnereth), (Naphtali, Jordan), (Issachar, Jordan),
=  (Manasseh, Jordan), (Gad, Jordan), (Benjamin, Jordan), (Judah, Jordan),
=  (Reuben, Jordan), (Judah, Dead), (Reuben, Dead), (Simeon, Dead),
=  (Gad, Jabbok), (Reuben, Jabbok), (Reuben, Arnon)];

val tribeBordersWater =
[(Asher, Mediterranean), (Manasseh, Mediterranean), (Ephraim, Mediterranean),
(Naphtali, Chinnereth), (Naphtali, Jordan), (Issachar, Jordan),
(Manasseh, Jordan), (Gad, Jordan), (Benjamin, Jordan), (Judah, Jordan),
(Reuben, Jordan), (Judah, Dead), ...] : (Tribe * WaterBody) list
```

The important thing is the type reported by ML: $(\text{Tribe} * \text{WaterBody}) \text{ list}$. Since $(\text{Tribe} * \text{WaterBody})$ is the Cartesian product and list is how we represent sets, this fits perfectly with our

formal definition of a relation. Now we can use our predicate `isElementOf` from Chapter 18 as a model for a predicate determining if two items are related.

```
- fun isRelatedTo(a, b, []) = false
=   | isRelatedTo(a, b, (h1, h2)::rest) =
=       (a = h1 andalso b = h2) orelse isRelatedTo(a, b, rest);

val isRelatedTo = fn : ''a * ''b * (''a * ''b) list -> bool

- isRelatedTo(Judah, Jordan, tribeBordersWater);

val it = true : bool

- isRelatedTo(Simeon, Chinnereth, tribeBordersWater);

val it = false : bool
```

Alternatively, we could use `isElementOf` directly.

```
- fun isRelatedTo(a, b, relat) = isElementOf((a,b), relat);

val isRelatedTo = fn : ''a * ''b * (''a * ''b) list -> bool
```

19.3 Manipulation

We conclude this introduction to relations by defining a few objects that can be computed from relations. First, the *image* of an element $a \in X$ under a relation R from X to Y is the set

$$\mathcal{I}_R(a) = \{b \in Y \mid (a, b) \in R\}$$

The image of 3 under `|` (assuming the subsets of \mathbb{Z} from the earlier example) is {15, 18}. The image of fox under `eats` is { rabbit }. The image of Reuben under `tribeBordersWater` is [Jordan, Dead, Jabbok, Arnon].

The *inverse* of a relation R from X to Y is the relation

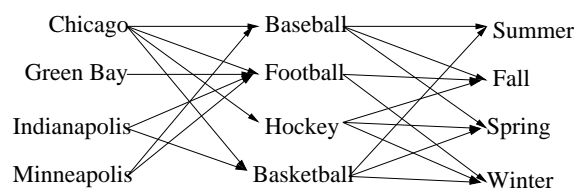
$$R^{-1} = \{(b, a) \in Y \times Y \mid (a, b) \in R\}$$

The inverse is easy to perceive graphically; all one does is reverse the direction of the arrows.

The *composition* of a relation R from X to Y and a relation S from Y to Z is the relation

$$R \circ S = \{(a, c) \in X \times Z \mid \exists b \in Y \text{ such that } (a, b) \in R \text{ and } (b, c) \in S\}$$

Suppose we had the set of cities $C = \{\text{Chicago, Green Bay, Indianapolis, Minneapolis}\}$, the set of professional sports $P = \{\text{baseball, football, hockey, basketball}\}$, and the set of seasons $S = \{\text{summer, fall, spring, winter}\}$. Let `hasMajorProTeam` be the relation from C to T representing whether a city has a major professional team in a given sport, and let `playsSeason` be the relation from T to S representing whether or not a professional sport plays in a given season. The composition `hasMajorProTeam` \circ `playsSeason` represents whether or not a city has a major professional team playing in a given season. In the illustration below, pairs in the composition are found by following two arrows.



Exercises

1. Complete the on-line relation diagram drills found at www.ship.edu/~deensl/DiscreteMath/flash/ch4/sec4_1/arrowdiagramsrelations.html and www.ship.edu/~deensl/DiscreteMath/flash/ch4/sec4_1/onesetarrows.html
2. The *identity relation* on a set X is defined as $I_X = \{(x, x) \in X \times X\}$. Prove that for a relation R from A to B , $R \circ I_B = R$.
3. Prove that if R is a relation from A to B , then $(R^{-1})^{-1} = R$.
4. If R is a relation from A to B , is $R \circ R^{-1} = I_A$? Prove or give a counterexample.
5. Write a function `image` which takes an element and a relation and returns the image of the element over the relation.
6. Write a function `addImage` which takes an element and a set (represented as a list) and returns a relation under which the given element's image is the given set.

We define the following relation to represent whether or not a body of water is immediately west of another.

```
- val waterImmedWestOf =
= [(Mediterranean, Chinnereth),
= (Mediterranean, Jordan), (Mediterranean, Dead),
= (Chinnereth, Jabbok), (Chinnereth, Arnon),
= (Jordan, Jabbok), (Jordan, Arnon),
= (Dead, Jabbok), (Dead, Arnon)];
```

Notice that a body of water a is west of another b if a is immediately west of b , or if it is immediately west of another body c which is in turn immediately west of b . The latter condition is found by the composition of `waterImmedWestOf` with itself.

7. Write a function `compose` which takes two relations and returns the composition of those two relations. (Hint: use your `image` and `addImage` functions.)

Chapter 20

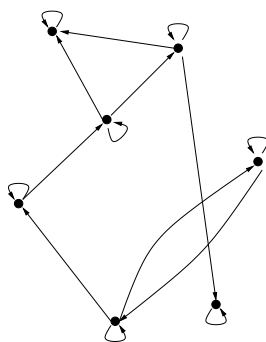
Properties of relations

20.1 Definitions

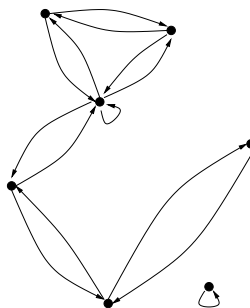
Certain relations that are on a single set X (as opposed to being from X to a distinct Y) have some of several interesting properties.

- A relation R on a set X is *reflexive* if $\forall x \in X, (x, x) \in R$. You can tell a reflexive relation by its graph because every element will have a self-loop. Examples of reflexive relations are $=$ on anything, \equiv on logical propositions, \leq and \geq on \mathbb{R} and its subsets, \subseteq on sets, and “is acquainted with” on people. *reflexive*
- A relation R on a set X is *symmetric* if $\forall x, y \in X$, if $(x, y) \in R$ then $(y, x) \in R$. In the graph of a symmetric relation, every arrow has a corresponding reverse arrow (except for self-loops, which are their own reverse arrow). Examples of symmetric relations are $=$ on anything, \equiv on logical propositions, and “is acquainted with” on people. *symmetric*
- For completeness, we mention that a relation R on a set X is *antisymmetric* if $\forall x, y \in X$, if $(x, y) \in R$ and $(y, x) \in R$, then $x = y$. We will consider antisymmetric relations more carefully in Chapter 22. *antisymmetric*
- A relation R on a set X is *transitive* if $\forall x, y, z \in X$, if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$. If you imagine “traveling” around a graph by hopping from element to element as the arrows permit, then you recognize a transitive relation because any trip that can be made in several hops can also be made in one hop. Examples of transitive relations are $=$ on anything, \equiv on logical propositions, \leq , \geq , $<$, and $>$ on \mathbb{R} and its subsets, and \subseteq on sets. *transitive*

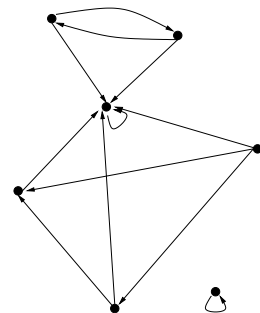
Furthermore, a relation is an *equivalence relation* if it is reflexive, symmetric, and transitive. *equivalence relation*



Reflexivity



Symmetry



Transitivity

20.2 Proofs

These properties give an exciting opportunity to revisit proving techniques because they require careful consideration of what burden of proof their definition demands. For example, suppose we have the theorem

Theorem 20.1 *The relation $|$ on \mathbb{Z}^+ is reflexive.*

If we unpack this using the definition of “reflexive,” we see that this is a “for all” proposition, which itself contains a set-membership proposition which requires the application of the definition of this particular relation.

Proof. Suppose $a \in \mathbb{Z}$.

Let the doubter pick any one element.

By arithmetic, $a \cdot 1 = a$, and so by the definition of divides, $a|a$.

Apply the definition of the relation.

Hence, by the definition of reflexive, $|$ is reflexive. \square

Satisfy the demands of reflexivity.

Symmetry and transitivity require similar reasoning, except that their “for all” propositions require two and three picks, respectively, and they contain General Form 2 propositions.

Theorem 20.2 *The relation $|$ on \mathbb{Z} is transitive.*

Proof. Suppose $a, b, c \in \mathbb{Z}$, and suppose $a|b$ and $b|c$. By the definition of divides, there exist $d, e \in \mathbb{Z}$ such that $a \cdot d = b$ and $b \cdot e = c$. By substitution and association, $a(d \cdot e) = c$. By the definition of divides, $a|c$. Hence $|$ is transitive. \square

Notice that this proof involved two applications of the definition of the relation, the first to analyze the fact that $a|b$ and $b|c$, the second to synthesize the fact that $a|c$. Why did we restrict ourselves to \mathbb{Z}^+ for reflexivity, but consider all of \mathbb{Z} for transitivity?

20.3 Equivalence relations

Equivalence relations are important because they group elements together into useful subsets and, as the name suggests, express some notion of equivalence among certain elements. The graph of an equivalence relation has all the properties discussed above for the graphs of the three properties, and also has the feature that elements are grouped into cliques that all have arrows to each other and no arrows to any elements in other cliques.

Examples of equivalence relations:

R on \mathbb{Z} where $(a, b) \in R$ if a and b are both even or both odd.

R on the set of logical formulas of variables p , q , and r where $(a, b) \in R$ if a and b have identical truth table columns.

For some $n \in \mathbb{Z}^+$, R on \mathbb{Z} where $(a, b) \in R$ if a and b are congruent mod n .

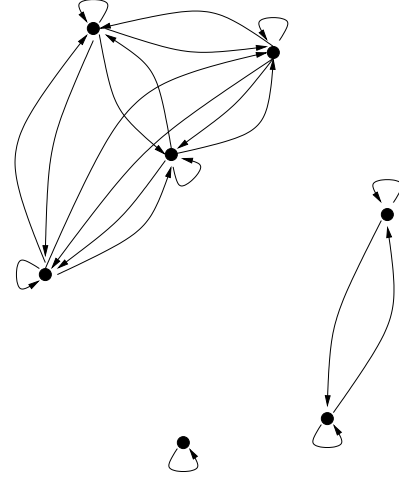
R on the set of points in the real plane where $((x_1, y_1), (x_2, y_2)) \in R$ if (x_1, y_1) and (x_2, y_2) are colinear.

R on the set of students where $(a, b) \in R$ if a and b have the same majors.

R on the set of variable names where $(a, b) \in R$ if a and b have the same first eight letters (which some old programming languages indeed consider equivalent).

R on $\mathbb{Z} \times \mathbb{Z}$ where $((a, b), (c, d)) \in R$ if $\frac{a}{b} = \frac{c}{d}$.

R on the set of real-valued functions where $(f(x), g(x)) \in R$ if $f'(x) = g'(x)$.



Proving that a specific relation is an equivalence relation follows a fairly predictable pattern. The parts of the proof are the proving of the three individual properties.

Theorem 20.3 Let R be a relation on \mathbb{Z} defined that $(a, b) \in R$ if $a + b$ is even. R is an equivalence relation.

Proof. Suppose $a \in \mathbb{Z}$. Then by arithmetic, $a + a = 2a$, which is even by definition. Hence $(a, a) \in R$ and R is reflexive.

Now suppose $a, b \in \mathbb{Z}$ and $(a, b) \in R$. Then, by the definition of even, $a + b = 2c$ for some $c \in \mathbb{Z}$. By the commutativity of addition, $b + a = 2c$, which is still even, and so $(b, a) \in R$. Hence R is symmetric.

Finally suppose $a, b, c \in \mathbb{Z}$, $(a, b) \in R$ and $(b, c) \in R$. By the definition of even, there exist $d, e \in \mathbb{Z}$ such that $a + b = 2d$ and $b + c = 2e$. By algebra, $a = 2d - b$. By substitution and algebra

$$\begin{aligned} a + c &= 2d - b + c &= 2d - 2b + b + c \\ &= 2d - 2b + 2e &= 2(d - b + e) \end{aligned}$$

which is even by definition (since $d - b + e \in \mathbb{Z}$). Hence $(a, c) \in R$, and so R is transitive.

Therefore, R is an equivalence relation by definition. \square

Once one knows that a relation is an equivalence relation, there are many other facts one can conclude about it.

Theorem 20.4 If R is an equivalence relation, then $R = R^{-1}$.

This result might be spiced up with a sophisticated subject, but the way to attack it is to remember that a relation is a set, and so $R = R^{-1}$ is in Set Proposition Form 2 and wrapped in a General Form 2 proposition.

Proof. Suppose R is an equivalence relation.

First suppose $(a, b) \in R$. Since R is an equivalence relation, it is symmetric, so $(b, a) \in R$ by definition of symmetry. Then by the definition of inverse, $(a, b) \in R^{-1}$, and so $R \subseteq R^{-1}$ by definition of subset.

Next suppose $(a, b) \in R^{-1}$. By definition of inverse, $(b, a) \in R$. Again by symmetry, $(a, b) \in R$, and so $R^{-1} \subseteq R$.

Therefore, by definition of set equality, $R = R^{-1}$. \square

relation induced

Equivalence relations have a natural connection to *partitions*, introduced back in Chapter 3. Let X be a set, and let $P = \{X_1, X_2, \dots, X_n\}$ be a partition of X . Let R be the relation on X defined so that $(x, y) \in R$ if there exists $X_i \in P$ such that $x, y \in X_i$. We call R the *relation induced* by the partition.

equivalence class

Similarly, let R be an equivalence relation on X . Let $[x]$ be the image of a given $x \in X$ under R . We call $[x]$ the *equivalence class* of x (under R). It turns out that any relation induced by a partition is an equivalence relation, and the collection of all equivalence classes under an equivalence relation is a partition.

Theorem 20.5 *Let A be a set, $P = \{A_1, A_2, \dots, A_n\}$ be a partition of A , and R be the relation induced by P . R is an equivalence relation.*

Proof. Suppose $a \in A$. Since $A = A_1 \cup A_2 \cup \dots \cup A_n$ by the definition of partition, $a \in A_1 \cup A_2 \cup \dots \cup A_n$. By the definition of union, there exists $A_i \in P$ such that $a \in A_i$. By definition of relation induced, $(a, a) \in R$. Hence R is reflexive.

Now suppose $a, b \in A$ and $(a, b) \in R$. By the definition of relation induced, there exists $A_i \in P$ such that $a, b \in A_i$. Again by the definition of relation induced, $(b, a) \in R$. Hence R is symmetric.

Finally suppose $a, b, c \in A$, $(a, b) \in R$, and $(b, c) \in R$. By the definition of relation induced, there exist $A_i, A_j \in P$ such that $a, b \in A_i$ and $b, c \in A_j$. Suppose that $A_i \neq A_j$. Then, by definition of intersection, $b \in A_i \cap A_j$. However, by definition of partition, A_i and A_j are disjoint, and so $A_i \cap A_j = \emptyset$ by the definition of disjoint. This is a contradiction, and so $A_i = A_j$. By substitution, $c \in A_i$, and so $(a, c) \in R$ by definition of relation induced. Hence R is transitive.

Therefore, R is an equivalence relation. \square

Theorem 20.6 *Let A be a set and R be an equivalence relation on A . Then $\{[a] \mid a \in A\}$ is a partition of A .*

Proof. Exercise 12.

20.4 Computing transitivity

We have already thought about representing relations in ML. Now we consider testing such relations for some of the properties we have talked about. A test for transitivity is delicate because the definition begins with a double “for all.” We can tame this by writing a modified (but equivalent) definition, that a relation R is transitive if

$$\forall (x, y) \in R, \forall (y, z) \in R, (x, z) \in R$$

That is, we choose (x, y) and (y, z) one at a time. This allows us to break down the problem. Assume that (x, y) is already chosen, and we want to know if transitivity holds specifically with respect to (x, y) . In other words, is it the case that for any element to which y is related, x is also related to that element? There are three cases for which we need to test, based on the makeup of the list:

1. The list is empty. Then *true*, the list is (vacuously) transitive with respect to (x, y) .
2. The list begins with (y, z) for some z . Then the list is transitive with respect to (x, y) if (x, z) exists in the relation, and if the rest of the list is transitive with respect to (x, y) .
3. The list begins with (w, z) for some $w \neq y$. Then the list is transitive with respect to (x, y) if the rest of the list is.

Expressing this in ML:

```

- fun testOnePair((a, b), []) = true
=   | testOnePair((a, b), (c, d)::rest) =
=       ((not (b = c)) orelse isRelatedTo(a, d, relation))
=       andalso testOnePair((a,b), rest);

```

Note two things. First, our symbolic notation is more expressive than ML since we can write $\forall(x, y), (y, z) \in R$, using y twice, but we cannot use a variable more than once in a pattern. That is, to write

```
...testOnePair((a, b), (b, c)::rest) = ...
```

is not allowed in ML. Hence we coalesce cases 2 and 3 as far as pattern matching goes and test b and c separately for equality. Second, we used the undefined variable `relation`, something we will have to fix by putting it in a proper context. The reason for this is that it is crucial that in case 2 we test if (a, c) exists in *the original relation*, not just the rest of the list currently being examined. In other words, it would be a serious error (why?) to write

```
...((not (b=c)) orelse relatedTo(a, d, rest))
```

The difficult part is complete. Now we must cover the first “for all,” which can be done by iterating over the list, testing each pair.

```

- fun test([]) = true
=   | test((a,b)::rest) =
=       testOnePair((a,b), relation) andalso test(rest);

```

Note again that we must be able to refer to the original relation, not just the rest of the list. As we saw in Chapter 14, it is good style to bundle into one package pieces like `test` and `testOnePair` that will not be used independently of our test for transitivity. This will also solve our problem of making `relation` a valid variable.

```

- fun isTransitive(relation) =
=   let fun testOnePair((a, b), []) = true
=       | testOnePair((a, b), (c, d)::rest) =
=           ((not (b = c)) orelse isRelatedTo(a, d, relation))
=           andalso testOnePair((a,b), rest);
=       fun test([]) = true
=       | test((a,b)::rest) =
=           testOnePair((a,b), relation) andalso test(rest);
=   in
=       test(relation)
=   end;

val isTransitive = fn : (''a * ''a) list -> bool

- val waterWestOf =
=   [(Mediterranean, Chinnereth), (Mediterranean, Jordan), (Mediterranean, Dead),
=   (Mediterranean, Jabbok), (Mediterranean, Arnon), (Chinnereth, Jabbok),
=   (Chinnereth, Arnon), (Jordan, Jabbok), (Jordan, Arnon), (Dead, Jabbok),
=   (Dead, Arnon)];

val waterWestOf =
  [(Mediterranean, Chinnereth), ...] : (WaterBody * WaterBody) list

- isTransitive(waterWestOf);

val it = true : bool

```

Exercises

1. Give a counterexample proving that $|$ is not symmetric over \mathbb{Z} .

For $x, y \in \mathbb{R}$, let $x \leq y$ if there exists $z \in \mathbb{R}^{\text{nonneg}}$ such that $x+z = y$.

2. Prove that \leq is reflexive over \mathbb{R} .
3. Give a counterexample proving that \leq is not symmetric over \mathbb{R} .
4. Prove that \leq is transitive over \mathbb{R} .

Let A be the set of intervals on the real number line. Let R be the relation on A such that two intervals are related if they overlap, that is, there exists a real number that is in both intervals.

5. Prove that R is reflexive.
6. Prove that R is symmetric.
7. Give a counterexample proving that R is not transitive.

Let R and S be relations on A .

8. Suppose that R is reflexive and that for all $a, b, c \in A$, if $(a, b) \in R$ and $(b, c) \in R$, then $(c, a) \in R$. Prove that R is an equivalence relation.
9. Prove that if R is an equivalence relation, then $R \circ R \subseteq R$.

10. Prove that if R is an equivalence relation and $(a, b) \in R$, then $\mathcal{I}_R(a) = \mathcal{I}_R(b)$. (Hint: Don't overlook the Set Proposition Form 2 of the conclusion.)
11. Prove that if R and S are both equivalence relations, then $R \circ S$ is an equivalence relation.
12. Prove Theorem 20.6.
13. It is tempting to think that if a relation R is symmetric and transitive, then it is also reflexive. This, however, is false. Give a counterexample. (Hint: Try to prove that R is reflexive. Then think about what unfounded assumption you are making in your proof.)
14. Based on what you discovered in Exercise 13, fill in the blank to make this proposition true and prove it. "If a relation R on a set A is symmetric and transitive and if _____, then R is reflexive."
15. Write a predicate `isSymmetric` which tests if a relation is symmetric, similar to `isTransitive`.
16. Write a predicate `isAntisymmetric` which tests if a relation is antisymmetric.
17. Why do we not ask you to write a predicate `isReflexive`? What would such a predicate require that you do not know how to do?

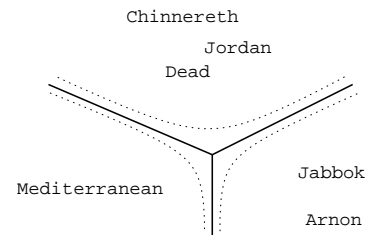
Chapter 21

Closures

21.1 Transitive failure

The water bodies of Israel neatly fit into vertical bands. The Mediterranean is to the west; the Sea of Chinnereth, the Jordan River, and the Dead Sea form a north-to-south line; and the Arnon and the Jabbok run east to west into the Jordan, parallel to each other. The relation “is vertically aligned with” is an equivalence relation, which we (naïvely) represent in with the following list; we also illustrate the partition inducing the relation.

```
- val waterVerticalAlign =  
= [(Chinnereth, Jordan), (Chinnereth, Dead),  
= (Jordan, Chinnereth), (Jordan, Dead),  
= (Dead, Chinnereth), (Dead, Jordan),  
= (Jabbok, Arnon), (Arnon, Jabbok)];  
  
val waterVerticalAlign = [...] : (WaterBody * WaterBody) list  
  
- isTransitive(waterVerticalAlign);  
  
  
val it = false : bool
```



To our surprise, it reports that `waterVerticalAlign`, a supposed equivalence relation, is not transitive. We must then turn our attention to debugging—why is our relation not transitive? Since the test for transitivity fails when pairs (a, b) and (c, d) are found in the list and $b = c$ but (a, d) is not found in the list, the most useful information we could get would be to ask the predicate `isTransitive`, “What pairs did you find that contradict transitivity?”

What if we modified `isTransitive` so that it returns a list of counterexamples instead of a simple true or false? This would tell us not only *that* a relation is intransitive, but *why* it is intransitive. This would require the following changes to `testOnePair`:

- Instead of returning `true` on an empty list or when `a` is related to `d`, return `[]`, that is, an empty list, indicating no contradicting pairs are found.
- Instead of returning `false` when `a` is not related to `d`, return `[(a, d)]`, that is, a list containing the pair that was expected but not found.
- Instead of anding the (boolean) value of `testOnePair` on the rest of the list with what we found for the current pair, concatenate the result for the current pair to the (list) value of `testOnePair`.

Observe the new function.

```

- fun counterTransitive(relation) =
= let fun testOnePair((a, b), []) = []
=     | testOnePair((a, b), (c,d)::rest) =
=         (if ((not (b=c)) orelse isRelatedTo(a, d, relation))
=             then [] else [(a, d)])
=         @ testOnePair((a,b), rest);
=     fun test([]) = []
=     | test((a,b)::rest) = testOnePair((a,b), relation) @ test(rest)
= in
=     test(relation)
= end;

```

```

val counterTransitive = fn : ('a * 'a) list -> ('a * 'a) list

```

Note the type. It is no trouble to write the same things again, and it is a safeguard for you.

```

- counterTransitive(waterVerticalAlign);

val it =
  [(Chinnereth,Chinnereth),(Chinnereth,Chinnereth),(Jordan,Jordan),
   (Jordan,Jordan),(Dead,Dead),(Dead,Dead),(Jabbok,Jabbok),(Arnon,Arnon)]
  : (WaterBody * WaterBody) list

```

This reveals the problem: we forgot to add self-loops. Adding them (but eliminating repeats) makes the predicate transitive.

```

- val correctedWaterVerticalAlign =
=   waterVerticalAlign @ makeNoRepeats(counterTransitive(waterVerticalAlign));

val correctedWaterVerticalAlign =
  [(Chinnereth,Jordan),(Chinnereth,Dead),(Jordan,Chinnereth),(Jordan,Dead),
   (Dead,Chinnereth),(Dead,Jordan),(Jabbok,Arnon),(Arnon,Jabbok),
   (Chinnereth,Chinnereth),(Jordan,Jordan),(Dead,Dead),(Jabbok,Jabbok),...]
  : (WaterBody * WaterBody) list

- isTransitive(correctedWaterVerticalAlign);

val it = true : bool

```

Similarly, we could use this to derive the transitive relation `waterWestOf` from `waterImmedWestOf`:

```

- isTransitive(waterImmedWestOf);

val it = false : bool

- val waterWestOf =
=   waterImmedWestOf @ makeNoRepeats(counterTransitive(waterImmedWestOf));

```

```

val waterWestOf =
  [...] : (WaterBody * WaterBody) list

- isTransitive(waterWestOf);

val it = true : bool

```

21.2 Transitive and other closures

`waterVerticalAlign` and `waterWestOf` are both examples of cases where adding certain pairs to a relation produces a new relation which is now transitive. The *smallest* transitive relation that is a superset of a relation R is called the *transitive closure*, R^T , of R . The requirement that this be the smallest such relation reflects that fact that there may be many transitive relations that are supersets of R (for example, the universal relation where everything is related to everything else). We are interested in adding the *fewest* possible pairs to make R transitive. Formally, R^T is the transitive closure of R if

transitive closure

1. R^T is transitive.
2. $R \subseteq R^T$.
3. If S is a relation such that $R \subseteq S$ and S is transitive, then $R^T \subseteq S$.

Theorem 21.1 *The transitive closure of a relation R is unique.*

Proof. Suppose S and T are relations fulfilling the requirements for being transitive closures of R . By items 1 and 2, S is transitive and $R \subseteq S$, so by item 3, $T \subseteq S$. By items 1 and 2, T is transitive and $R \subseteq T$, so by item 3, $S \subseteq T$. Therefore $S = T$ by the definition of set equality. \square

Some examples of transitive closures:

- The transitive closure of our relation “eats” on { hawk, coyote, rabbit, fox, clover } is “gets nutrients from.” A coyote ultimately gets nutrients from clover.
- Let R be the relation on \mathbb{Z} defined that $(a, b) \in R$ if $a+1 = b$. Thus $(-15, -14), (1, 2), (23, 24) \in R$. The transitive closure of R is $<$.

The *reflexive closure* and the *symmetric closure* are defined similarly, though these are of less importance. The reflexive closure of $<$ is \leq . “Is in love with” in an ideal world is the symmetric closure of “is in love with” in the real world.

reflexive closure

symmetric closure

21.3 Computing the transitive closure

Our success using `@` and `makeNoRepeats` might make us optimistically write a function

```

- fun transitiveClosure(relation) =
  = relation @ makeNoRepeats(counterTransitive(relation));

```

However, this is wrong. Suppose we test it on a relation that relates the tribes descended from Leah according to who immediately precedes whom in birth (since we are considering tribes as geographic entities, we have ignored Levi, who received no land).

```

- val immediatelyPrecede =
=   [(Reuben, Simeon), (Simeon, Judah), (Judah, Issachar),
=   (Issachar, Zebulun)];

val immediatelyPrecede = [...] : (Tribe * Tribe) list

- val birthOrder = transitiveClosure(immediatelyPrecede);

val birthOrder =
  [(Reuben, Simeon), (Simeon, Judah), (Judah, Issachar), (Issachar, Zebulun),
   (Reuben, Judah), (Simeon, Issachar), (Judah, Zebulun)] : (Tribe * Tribe) list

- isTransitive(birthOrder);

val it = false : bool

- counterTransitive(birthOrder);

val it =
  [(Reuben, Issachar), (Simeon, Zebulun), (Reuben, Issachar), (Reuben, Zebulun),
   (Simeon, Zebulun)] : (Tribe * Tribe) list

```

The transitive closure should completely express who is older than whom, yet the answer is missing, for example, (Reuben, Issachar). By adding the pair (Reuben, Judah), we have also created a new “missing pair.” To compute the transitive closure correctly, we must add missing pairs repeatedly until the relation is transitive. In other words, we must add not only the pairs of $R^2 = R \circ R$, but also those of R^3 , R^4 , etc. The following theorem informs how to calculate the transitive closure.

Theorem 21.2 *If R is a relation on a set A , then*

$$R^\infty = \bigcup_{i=1}^{\infty} R^i = \{(x, y) \mid \exists i \in \mathbb{N} \text{ such that } (x, y) \in R^i\}$$

is the transitive closure of R .

Proof. Suppose R is a relation on a set A .

Suppose $a, b, c \in A$, $(a, b) \in R^\infty$, and $(b, c) \in R^\infty$. By the definition of R^∞ , there exist $i, j \in \mathbb{N}$ such that $(a, b) \in R^i$ and $(b, c) \in R^j$. By the definition of relation composition, $(a, c) \in R^i \circ R^j = R^{i+j} \subseteq R^\infty$. By the definition of subset, $(a, c) \in R^\infty$, and so R^∞ is transitive.

Suppose $a, b \in A$ and $(a, b) \in R$. By the definition of R^∞ (taking $i = 1$), $(a, b) \in R^\infty$, and so $R \subseteq R^\infty$.

Suppose S is a transitive relation on A and $R \subseteq S$. Further suppose $(a, b) \in R^\infty$. Then, by definition of R^∞ , there exists $i \in \mathbb{N}$ such that $(a, b) \in R^i$. We will prove that $(a, b) \in S$ by induction on i .

Suppose $i = 1$. Then $(a, b) \in R \subseteq S$, so $(a, b) \in S$. Hence there exists some $I \geq 1$ such that for all $(e, f) \in R^I$, $(e, f) \in S$.

Next suppose that $i = I + 1$. Then by the definition of relation composition there exist $j, k \in \mathbb{N}$, $j + k = i$ and $c \in A$ such that $(a, c) \in R^j$ and $(c, b) \in R^k$. Since $j, k < i$, $j, k \leq I$, both by arithmetic. By our induction hypothesis, $(a, c), (c, b) \in S$. Since S is transitive, $(a, b) \in S$.

Hence, by math induction, $(a, b) \in S$ for all $i \in \mathbb{N}$.

Hence $R^\infty \subseteq S$ by definition of subset.

Therefore, R^∞ is the transitive closure of R . \square

The potential need for making an infinity of compositions is depressing. However, on a finite set, the number of possible pairs is also finite, so eventually these compositions will not have anything more to add; we can stop when the relation we are constructing is finally transitive. Interpreting this iteratively,

```
- fun transitiveClosure(relation) =
=   let val closure = ref relation;
=   in
=     (while not (isTransitive(!closure)) do
=       closure := !closure @ makeNoRepeats(counterTransitive(!closure));
=       !closure)
=   end;
```

Iu Manin said, “A good proof is one which makes us wiser” [10]. The same can be said about good programs. In this case, we can restate our definition of the transitive closure of a relation to be

- The relation itself, if it is already transitive, or
- The transitive closure of the union of the relation to its immediately missing pairs, otherwise.

Hence in the applicative style, we have

```
- fun transitiveClosure(relation) =
=   if isTransitive(relation)
=     then relation
=     else transitiveClosure(makeNoRepeats(counterTransitive(relation))
=                               @ relation);
```

21.4 Relations as predicates

Recall that our initial problem with the relation `waterVerticalAlign` was that it was missing self-loops. Hence the reflexive closure would have solved our problem just as well as the transitive closure. Unfortunately, there is no way to compute the reflexive closure given our way of representing relations (think back to Exercise 17 of Chapter 20).

This is particularly frustrating because the reflexive closure is so simple. For example, we could write a predicate like `isRelatedTo` except that it tests if two elements are related by the relation’s reflexive closure.

```
- fun isRelatedToByRefClos(a, b, relation) =
=   a = b orelse relatedTo(a, b, relation);
```

This suggests that our situation could be remedied if we followed the intuition of the other way to represent relations, as predicates. For example, `eats` back in Chapter 9 is a relation.

What we do not want to lose is the ability to treat relations as what we have been calling “mathematical entities.” What we mean is that we should be able to pass a relation to a function and have a function return a relation, as `transitiveClosure` does. A value in a programming environment that can be passed to and returned from a function is a *first-class value*. A pillar of functional programming is that functions are first-class values.

first-class value

A relation represented by a predicate will have a type like $(\text{'a} * \text{'a}) \rightarrow \text{bool}$. This means “function that maps from an $\text{'a} \times \text{'a}$ pair to a `bool`.” Our first task is to write a function that will convert from list representation to predicate representation. To return a predicate from a function, simply define the predicate (locally) within the function and return it by naming it without giving any parameters.

```

- fun listToPredicate(oldRelation) =
=   let fun newRelation(a, b) = isRelatedTo(a, b, oldRelation);
=   in
=     newRelation
=   end;

```

```

val listToPredicate = fn : (''a * ''b) list -> ''a * ''b -> bool

```

In a similar way, a function can receive a predicate. Observe this function to compute the reflexive closure:

```

- fun reflexiveClosure(relation) =
=   let fun closure(a, b) = a = b orelse relation(a, b);
=   in
=     closure
=   end;

```

```

val reflexiveClosure = fn : (''a * ''a -> bool) -> ''a * ''a -> bool

```

Computing the symmetric closure is an exercise. We cannot compute the transitive closure directly, but if the relation is originally represented as a list, we could compute the transitive closure before converting to predicate form.

As in Chapter 4, we are faced with the dilemma of choosing among two representations, each of which has favorable features and drawbacks. The list representation is the more intuitive, at least in terms of the formal definition of a relation; with the predicate representation, however, we can test a pair for membership directly, as opposed to relying on a predicate like `isRelatedTo`. Because we can iterate through all pairs, the list representation allows testing and computing of transitivity, but with the predicate representation we can compute reflexive and symmetric closures. Conversion is one-way, from lists to predicates. Neither representation allows us to test reflexivity. These aspects are summarized below. Every “no” would become a “yes” if only we had the means to iterate through all elements of a datatype.

	List	Predicate
first class value	yes	yes
membership test	indirectly	directly
<code>isReflexive</code>	no	no
<code>isSymmetric</code>	yes	no
<code>isTransitive</code>	yes	no
<code>isAntiSymmetric</code>	yes	no
<code>reflexiveClosure</code>	no	yes
<code>symmetricClosure</code>	yes	yes
<code>transitiveClosure</code>	yes	no
convert to other	yes	no

Finally, a word of interest only to those who have programmed in an object-oriented language such as Java. When you read examples like these, you should be thinking about the best way to represent a concept in other programming languages. In an object-oriented language, objects are first-class values; hence we would want to write a class to represent relations. The primary difference between the list and predicate representations presented here is that the former represents the relation as *data*, the latter as *functionality*. The primary characteristic of objects is that they encapsulate data and functionality together in one package. Since the predicate representation can be built from a list representation, one would expect that it would be strictly more powerful; however, in the conversion we lost the ability to test for transitivity, and this is because we have lost access to the list. If instead we made the list to be an instance variable of a class, the methods could do the functional work

of `reflexiveClosure` as well as the iterative work of `isTransitive`. Moreover, Java 5 has enum types unavailable in earlier versions of Java, which provide the functionality of ML's datatypes, as well as a means of iterating over all elements of a set, something that hinders representing relations as lists or predicates in ML (ML's datatype is more powerful than Java's enum types in other ways, however). The following shows an implementation of relations using Java 5.

```

/**
 * Class Relation to model mathematical relations.
 * The relation is assumed to be over a set modeled
 * by a Java enum.
 *
 * @author ThomasVanDrunen
 * Wheaton College
 * June 30, 2005
 */
public class Relation<E extends Enum<E>> {

    private class List {
        public E first;
        public E second;
        public List tail;
        public List(E first, E second, List tail) {
            this.first = first;
            this.second = second;
            this.tail = tail;
        }
    }

    /**
     * Concatenate a give list to the end of this list.
     * @param other The list to add.
     * POSTCONDITION: The other list is added to
     * the end of this list; the other list is not affected.
     */
    public void concatenate(List other) {
        if (tail == null) tail = other;
        else tail.concatenate(other);
    }

    /**
     * The set ordered pairs of the relation.
     */
    private List pairs;

    /**
     * Constructor to create a new relation of n pairs from an
     * n by 2 array.
     * @param input The array of pairs; essentially an array of
     * length-2 arrays of the base enum.
     */
    public Relation(E[][] input) {
        for (int i = 0; i < input.length; i++) {
            assert input[i].length == 2;
            pairs = new List(input[i][0], input[i][1], pairs);
        }
    }

    public boolean relatedTo(E a, E b) {
        for (List current = pairs; current != null;
             current = current.tail)
            if (current.first == a && current.second == b)
                return true;
        return false;
    }

    public boolean isReflexive() {
        // if there are no pairs, we can assume this is
        // not reflexive.
        if (pairs == null) return false;
        try {
            for (E t : (E[]) pairs.first.getClass()
                       .getMethod("values").invoke(null))
                if (! relatedTo(t, t)) return false;
        } catch (Exception e) { } // won't happen
        return true;
    }

    public boolean isSymmetric() {
        for (List current = pairs; current != null;
             current = current.tail)
            if (! relatedTo(current.second, current.first))
                return false;
        return true;
    }

    private boolean isTransitiveWRTPair(E a, E b) {
        for (List current = pairs; current != null;
             current = current.tail)
            if (b == current.first
                && ! relatedTo(a, current.second))
                return false;
        return true;
    }

    public boolean isTransitive() {
        for (List current = pairs; current != null;
             current = current.tail)
            if (! isTransitiveWRTPair(current.first,
                                      current.second))
                return false;
        return true;
    }

    public boolean isAntisymmetric() {
        if (isSymmetric()) return false;
        for (List current = pairs; current != null;
             current = current.tail)
            if (current.first != current.second &&
                relatedTo(current.second, current.first))
                return false;
        return true;
    }

    public Relation reflexiveClosure() {
        if (isReflexive()) return this;
        else return new Relation<E>() {
            public boolean isReflexive() { return true; }
            public boolean relatedTo(E a, E b) {
                return a == b
                    || Relation.this.relatedTo(a, b);
            }
        };
    }

    public Relation symmetricClosure() {
        if (isSymmetric()) return this;
        else return new Relation<E>() {
            public boolean isSymmetric() { return true; }
            public boolean relatedTo(E a, E b) {
                return Relation.this.relatedTo(a, b) ||
                    Relation.this.relatedTo(b, a);
            }
        };
    }

    private List counterTransitiveWRTPair(E a, E b) {
        List toReturn = null;
        for (List current = pairs; current != null;
             current = current.tail)
            if (b == current.first
                && ! relatedTo(a, current.second))
                toReturn = new List(a, current.second, toReturn);
        return toReturn;
    }

    private List counterTransitive() {
        List toReturn = null;
        for (List current = pairs; current != null;
             current = current.tail) {
            List currentCounter =
                counterTransitiveWRTPair(current.first,
                                         current.second);
            if (currentCounter != null) {
                currentCounter.concatenate(toReturn);
                toReturn = currentCounter;
            }
        }
        return toReturn;
    }

    /**
     * Default constructor used by transitiveClosure().
     */
    private Relation() {}

    public Relation transitiveClosure() {
        if (isTransitive()) return this;
        Relation toReturn = new Relation();
        toReturn.pairs = counterTransitive();
        toReturn.pairs.concatenate(pairs);
        return toReturn.transitiveClosure();
    }

    public boolean isEquivalenceRelation() {
        return isReflexive() && isSymmetric() && isTransitive();
    }

    public boolean isPartialOrder() {
        return isReflexive() && isAntisymmetric()
            && isTransitive();
    }
}

```

Exercises

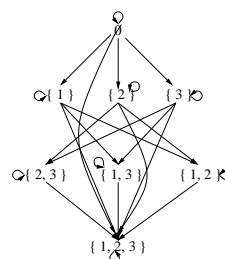
1. Prove that if R is a relation on A , then $R \cup I_A$ is uniquely the reflexive closure R .
2. Prove that if R is a relation on A , then $R \cup R^{-1}$ is uniquely the symmetric closure of R .
3. Write a function `counterSymmetric` which operates on the list representation of relations.
4. Using `counterSymmetric`, write a function `symmetricClosure`.
5. Write a function `counterAntisymmetric` which operates on the list representation of relations and constructs a list of counter examples to antisymmetry. This is different from `counterSymmetric` and `counterTransitive` because it will create a list of extraneous pairs, not missing pairs.
6. Write a function that computes the symmetric closure for the predicate representation.

Chapter 22

Partial orders

22.1 Definition

No one would mistake the \subseteq relation over a powerset for an equivalence relation by looking at the graph. So far from parcelling the set out into autonomous islands, it connects everything in an intricate, flowing, and (if drawn well) beautiful network. However, it does happen to have two of the three attributes of an equivalence relation: it is reflexive and it is transitive. Symmetry makes all the difference here. \subseteq in fact is the opposite of symmetric—it is antisymmetric, the property we mentioned only briefly in Chapter 19, but which you also have seen in a few exercises. Being antisymmetric, informally, means that no two distinct elements in the set are mutually related—though any single element may be (mutually) related to itself. Note carefully, though, that the definition of antisymmetric says “if two elements are mutually related, they must be equal,” not “if two elements are equal, they must be mutually related.” Relations like this are important because they give a sense of order to the set, in this case a hierarchy from the least inclusive subset to the most inclusive, with certain sets at more or less the same level.



A *partial order relation* is a relation R on a set X that is reflexive, transitive, and antisymmetric. A set X on which a partial order is defined is called a *partially ordered set* or a *poset*. The idea of the ordering being only partial is because not every pair of elements in the set is organized by it. In this case, for example, $\{1, 2\}$ and $\{1, 3\}$ are not *comparable*, which we will define formally in the next section.

partial order relation

poset

Theorem 22.1 *Let A be any set of sets over a universal set U . Then A is a poset with the relation \subseteq .*

Proof. Suppose A is a set of sets over a universal set U .

Suppose $a \in A$. By the definition of subset, $a \subseteq a$. Hence \subseteq is reflexive.

Suppose $a, b, c \in A$, $a \subseteq b$, and $b \subseteq c$. Suppose further that $x \in a$. By definition of subset, $x \in b$, and similarly $x \in c$. Again by the definition of subset, $a \subseteq c$. Hence \subseteq is transitive.

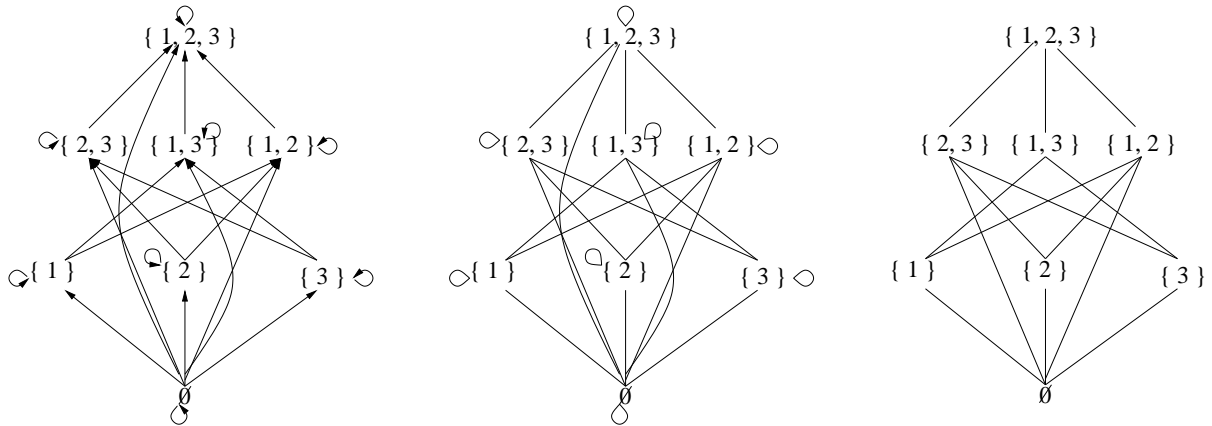
Finally suppose $a, b \in A$, $a \subseteq b$, and $b \subseteq a$. By the definition of set equality, $a = b$. Hence, by the definition of antisymmetry, \subseteq is antisymmetric.

Therefore, A is a poset with \subseteq . \square

The graphs of equivalence relations and of partial orders become very cluttered. For equivalence relations, it is more useful visually to illustrate regions representing the equivalence classes, like on page 141. For partial orders, we use a pared down version of a graph called a *Hasse diagram*, after German mathematician Helmut Hasse. It strips out redundant information. To transform the

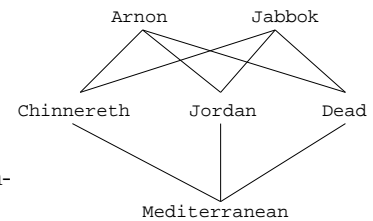
Hasse diagram

graph of a partial order relation into a Hasse diagram, first draw it so that all the arrows (except for self-loops) are pointing up. Antisymmetry makes this possible. Then, since the arrangement on the page informs us what direction the arrows are going, the arrowheads themselves are redundant and can be erased. Finally, since we know that the relation is transitive and reflexive, we can remove self-loops and short-cuts. In the end we have something more readable, with the (visual) symmetry apparent.



Some examples of partial orders:

- $|$ (divides) on \mathbb{Z}^+ .
- \leq on \mathbb{R} .
- **waterWestOf** on **WaterBody**.
- Alphabetical ordering over the set of lexemes in a language.



Generic partial orders are often denoted by the symbol \preceq , with obvious reference to \leq and \subseteq .

22.2 Comparability

comparable

We have noted that the partial order relation \subseteq does not put in order, for example, $\{1,3\}$ and $\{2,3\}$. We say that for a partial order relation \preceq on a set A , $a, b \in A$ are *comparable* if $a \preceq b$ or $b \preceq a$. 12 and 24 are comparable for $|$ (12|24), but 12 and 15 are noncomparable. Mediterranean and Arnon are comparable for **waterWestOf** (the Mediterranean is west of the Arnon), but Arnon and Jabbok are noncomparable. For \subseteq , \emptyset is comparable to everything; in fact, it is a subset of everything. Arnon is not comparable to everything, but everything it is comparable with is west of it. These last observations lead us to say that if \preceq is a partial order relation on A , then $a \in A$ is

maximal

- *maximal* if $\forall b \in A, b \preceq a$ or b and a are not comparable.

minimal

- *minimal* if $\forall b \in A, a \preceq b$ or b and a are not comparable.

greatest

- *greatest* if $\forall b \in A, b \preceq a$.

least

- *least* if $\forall b \in A, a \preceq b$

As you can see, a poset may have many maximal or minimal elements, but at most one greatest or least. An infinite poset, like \mathbb{R} with \leq may have none, but we can prove that a finite poset has at least one maximal element. First, a trivial result that will be useful: For any poset, we can remove one element and it will still be a poset.

Lemma 22.1 *If A is a poset with partial order relation \preceq , and $a \in A$, then $A - \{a\}$ is a poset with partial order relation $\preceq - \{(b, c) \in \preceq \mid b = a \text{ or } c = a\}$.*

Proof. Suppose A is a poset with partial order relation \preceq , and suppose $a \in A$. Let $A' = A - \{a\}$ and $\preceq' = \preceq - \{(b, c) \in \preceq \mid b = a \text{ or } c = a\}$

Now suppose $b \in A'$. By definition of difference, $b \in A$, and since \preceq is reflexive, $b \preceq b$. Also by definition of difference, $b \neq a$, so $b \preceq' b$. Hence \preceq' is reflexive.

Suppose $b, c \in A'$, $b \preceq' c$, and $c \preceq' b$. By definition of difference, $b, c \in A$, $b \preceq c$, and $c \preceq b$. Since \preceq is antisymmetric, $b = c$. Hence \preceq' is reflexive.

Finally, suppose $b, c, d \in A'$, $b \preceq' c$, and $c \preceq' d$. By definition of difference, $b, c, d \in A$, $b \preceq c$, and $c \preceq d$. Since \preceq is transitive, $b \preceq d$. Also by the definition of difference, $b \neq a$ and $d \neq a$, so $b \preceq' d$. Hence \preceq' is transitive.

Therefore A' is a partial order relation with \preceq' . \square

Theorem 22.2 *A finite, non-empty poset has at least one maximal element.*

Proof. Suppose A is a poset with partial order relation \preceq . We will prove it has at least one maximal element by induction on $|A|$.

Base case. Suppose $|A| = 1$. Let a be the one element of A . Trivially, suppose $b \in A$. Since $|A| = 1$, $b = a$, and since \preceq is reflexive, $b \preceq a$. Hence a is a maximal element, and so there exists an $N \geq 1$ such that for any poset of size N , it has at least one maximal element.

Inductive case. Suppose $|A| = N + 1$. Suppose $a \in A$. Let $A' = A - \{a\}$ and $\preceq' = \preceq - \{(b, c) \in \preceq \mid b = a \text{ or } c = a\}$. By Lemma 22.1, A' is a partial order with \preceq' . By the inductive hypothesis, A' has at least one maximal element. Let b be a maximal element of A' . Now we have three cases.

Case 1: Suppose $a \preceq b$. Then suppose $c \in A$. If $c \neq a$, then since b is maximal in A' , either $c \preceq' b$ (in which case, by definition of difference, $c \preceq b$) or c and b are noncomparable (in which case c and b are still incomparable in \preceq'). If $c = a$, then $c \preceq b$ by our supposition. Hence b is a maximal element in A .

Case 2: Suppose $b \preceq a$. See Exercise 12.

Case 3: Suppose b and a are incomparable with \preceq . See Exercise 13.

22.3 Topological sort

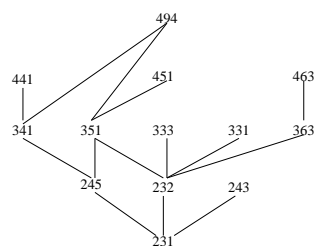
If all pairs in a poset are comparable, then the partial order relation \preceq is a *total order relation*. Total orders we have seen include \leq on \mathbb{R} and alphabetical ordering on lexemes. As the name suggests, such a relation puts the set into a complete ordering; a relation like this could be used to sort a subset of a poset. However, there are many situations where elements of a poset need to be put into sequence even though the relation is not a total order relation. In that case, we merely must disambiguate the several possibilities for sequencing noncomparable elements. If X is a poset with partial order relation \preceq , then the relation \preceq' is a *topological sort* if $\preceq \subseteq \preceq'$ and \preceq' is a total order relation. In other words, a topological sort of a partial order relation is simply that partial order relation with the ambiguities worked out.

total order relation

topological sort

Wheaton College's mathematics program has the following courses with their prerequisites:

231: none	351: 232 and 245
232: 231	363: 232
243: 231	441: 341
245: 231	451: 351
331: 232	463: 363
333: 232	494: 341 and 351
341: 245	



A topological sort would be a sequence of these courses taken in an order that does not conflict with the prerequisite requirement, for example,

231—232—243—333—245—331—351—341—441—363—463—494—451

Exercises

Consider the set of Jedis and Siths, $A = \{ \text{Yoda, Palpatine, Mace, Anakin, Obi-Wan, Maul, Qui-Gon, Dooku, Luke} \}$. Let R be a relation on A defined so that $(a, b) \in R$ if a has defeated b in a light-saber fight (not necessarily fatally). Specifically:

- Ep 1.** Maul defeats Qui-Gon.
- Ep 1.** Obi-Wan defeats Maul.
- Ep 2.** Dooku defeats Obi-Wan.
- Ep 2.** Dooku defeats Anakin.
- Ep 2.** Yoda defeats Dooku.
- Ep 3.** Anakin defeats Dooku.
- Ep 3.** Mace defeats Palpatine.
- Ep 3.** Palpatine defeats Yoda.
- Ep 4.** Anakin defeats Obi-Wan.
- Ep 5.** Anakin defeats Luke.
- Ep 6.** Luke defeats Anakin.

1. Draw a graph representing this relation.
2. Let R^T be the transitive closure of R . Which of the following are true?
 - (a) $(\text{Obi-Wan, Qui-Gon}) \in R^T$.
 - (b) $(\text{Mace, Maul}) \in R^T$.
 - (c) $(\text{Luke, Yoda}) \in R^T$.
 - (d) $(\text{Palpatine, Luke}) \in R^T$.
3. Give a counterexample to show that R is not antisymmetric.
4. If you consider only the *final* meeting of each Jedi/Sith pair (that is, eliminate the pairs (Obi-Wan, Anakin), (Anakin, Luke), and (Dooku, Anakin)) and take the transitive closure, we are left with a partial order. Draw the Hasse diagram.
5. Suppose you wanted to find a relation S that associated people with others on the same side in the conflict (that is, $S = \text{"is on the same side"}$). S should be an equivalence relation and it should be that if $(a, b) \in S$ then $(a, b) \notin R$, that is, a person never fights with someone on the same side. Explain why S does not exist for the given data (you need not know the story; merely find a counterexample in the given data). If you know the movies, explain why this is the case in terms of the story.
6. Give a topological sort of the partial order in Exercise 4.
7. Notice that the Hasse diagram of `waterWestOf` looks very much like a graph of `waterImmedWestOf`, just without the arrowheads. Using this observation, we could implement in ML a minimal representation of a partial order that only contained pairs that would be connected in the Hasse diagram. Write a function `isRelatedPO` which takes two elements and a minimal list representation of a partial order

and determines if the first element is related to the second in the partial order.

8. Alphabetical order is generalized by what is called *lexicographical order*. If A_1, A_2, \dots, A_n are posets with total order relations $\preceq_1, \preceq_2, \dots, \preceq_n$, respectively, then the lexicographical order of $A_1 \times A_2 \times \dots \times A_n$ is \preceq_ℓ defined as $(a_1, a_2, \dots, a_n) \preceq_\ell (b_1, b_2, \dots, b_n)$ if $a_1 \preceq_1 b_1, a_2 \preceq_2 b_2, \dots, a_n \preceq_n b_n$. Prove that a lexicographical order is a total order relation.
9. The previous example was defined for tuples. We can imagine a similar concept for lists, where all the elements must come from the same set, using only one total order relation. To deal with the varying size of lists, we will say that if a longer list has a shorter list as a prefix, then the shorter list comes before the longer. Write a function `isRelatedLex` which takes two lists and a total order relation on the elements in those lists and determines if the first list should come after the second.
10. If R and S are antisymmetric relations on a set A , is $R \cup S$ antisymmetric? Prove or give a counterexample.
11. If R and S are partial order relations on a set A , is $R \circ S$ a partial order relation? Prove or give a counterexample.
12. Prove case 2 of Theorem 22.2.
13. Prove case 3 of Theorem 22.2.
14. Suppose A is a poset with a total order \preceq , and suppose $a \in A$ is a maximal element. Prove that a is the greatest element.
15. What relation is reflexive, symmetric, transitive, and antisymmetric?
16. Prove that $|$ (divides) on \mathbb{Z}^+ is antisymmetric.
17. \mathbb{C} stands out from the other standard number sets in that \leq (and similar comparison operators) is no longer defined. Find a partial order for \mathbb{C} based on \leq .
18. Find a total order for \mathbb{C} . (Hint: Find a topological sort of your answer to Exercise 17.)
19. The following exercise comes from Cormen et al [4]. Professor Bumstead is getting dressed. He needs to put on a belt, a jacket, pants, a shirt, shoes, socks, a tie, an undershirt, undershorts, and a watch. Draw a Hasse diagram showing the order in which certain articles must be put on in relation to one another. Then give a topological sort that gives a reasonable order in which Professor Bumstead can put them on.

Part VI

Function

Chapter 23

Functions

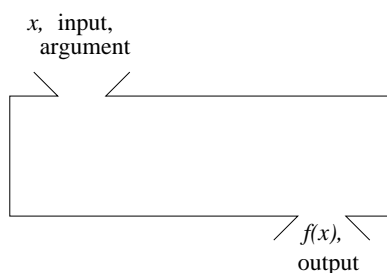
23.1 Intuition

Of all the major topics in this course, the function is probably the one with which your prior familiarity is the greatest. In the modern treatment of mathematics, functions pervade algebra, analysis, the calculus, and even analytic geometry and trigonometry. It is hard for today's student even to comprehend the mathematics of earlier ages which did not have the modern understanding and notation of functions. Recall the distinction we made at the beginning of this course, that though you had previously studied the *contents* of various number sets, you would now study sets themselves. Similarly, up till now you have used functions to talk about other mathematical objects. Now we shall study functions as mathematical objects themselves.

Your acquaintance with functions has given you several models or metaphors by which you conceive functions. One of the first you encountered was that of *dependence*—two phenomena or quantities are related to each other in such a way that one was dependent on the other. In high school chemistry, you may have performed an experiment where, given a certain volume of water, the change of temperature of the water was affected by how much heat was applied to the water. The number of joules applied is considered the *independent variable*, a quantity you can control. The temperature change, in Kelvins, is a *dependent variable*, which changes predictably based on the independent variable. Let f be the temperature change in Kelvins and x be the heat in joules. With a kilogram of water, you would discover that

$$f(x) = 4.183x$$

Next, one might think of a function as a kind of machine, one that has a slot into which you can feed raw materials and a slot where the finished product comes out, like a Salad Shooter.



A function is sometimes defined as a *mapping* or association between two collections of things. Think of how a telephone book associates names with phone numbers. This concept of mapping crosses disciplines in surprising ways. It is interesting to note that mapping is not only a synonym for *function* in mathematics but also for *metaphor* in cognitive linguistics.

Finally, think back to our introduction to relations. Considered graphically, a (real-valued) function is a curve that passes the vertical line test—that is, there is no x value for which there are more than one y value. This last model best informs our formal definition of a function—a function is a restricted kind of relation. This works not only for functions in the specific world of the real-number plane, but for functions between any sets.

23.2 Definition

function
domain
codomain

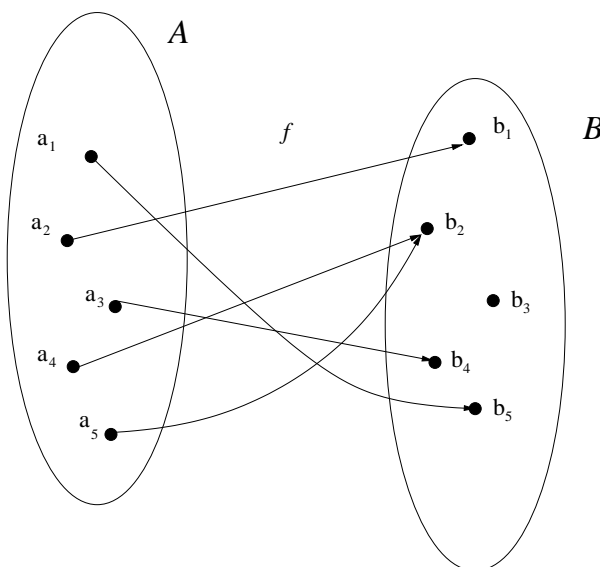
A *function* f from a set X to a set Y is a relation from X to Y such that each $x \in X$ is related to exactly one $y \in Y$, which we denote $f(x)$. We call X the *domain* of f and Y the *codomain*. We write $f : X \rightarrow Y$ to mean “ f is a function from X to Y .”

well defined

Let $A = \{1, 2, 3\}$ and $B = \{5, 6, 7\}$. Let $f = \{(1, 5), (2, 5), (1, 7)\}$. f is not a function because 1 is related to two different items, 5 and 7, and also because 3 is not related to any item. (It is not a problem that more than one item is related to 5, or that nothing is related to 6.) When a supposed function meets the requirements of the definition, we sometimes will say that it is *well defined*. Make sure you remember, however, that being well defined is the same thing as being a function. Do not say “well-defined function” unless the redundancy is truly warranted for emphasis.

range

The term *codomain* might sound like what you remember calling the *range*. However, we give a specific and slightly different definition for that: for a function $f : X \rightarrow Y$, the *range* of f is the set $\{y \in Y \mid \exists x \in X \text{ such that } f(x) = y\}$. That is, a function may be broadly defined to a set but may actually contain pairs for only some of the elements of the codomain. An arrow diagram will illustrate.



Since f is a function, each element of A is at the tail of exactly one arrow. Each element of B may be at the head of any number (including zero) of arrows. Although the codomain of f is $B = \{b_1, b_2, b_3, b_4, b_5\}$, since nothing maps to b_3 , the range of f is $\{b_1, b_2, b_4, b_5\}$.

function equality

Two functions are *equal* if they map all domain elements to the same things. That is, for $f : X \rightarrow Y$ and $g : X \rightarrow Y$, $f = g$ if for all $x \in X$, $f(x) = g(x)$. The following result is not surprising, but it demonstrates the structure of a proof of function equality.

Theorem 23.1 Let $f : \mathbb{R} \rightarrow \mathbb{R}$ as $f(x) = x^2 - 4$ and $g : \mathbb{R} \rightarrow \mathbb{R}$ as $g(x) = \frac{(2x-4)(x+1)}{2}$. $f = g$.

Proof. Suppose $x' \in \mathbb{R}$. Then, by algebra

$$\begin{aligned}
 f(x') &= x'^2 - 4 \\
 &= x'^2 - 2x' + 2x' - 4 \\
 &= (x' - 2)(x' + 2) \\
 &= \frac{2(x' - 2)(x' + 2)}{2} \\
 &= \frac{(2x' - 4)(x' + 2)}{2} = g(x')
 \end{aligned}$$

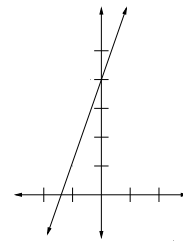
Hence, by definition of function equality, $f = g$. \square

Notice that we chose x' as the variable to work with instead of x . This is to avoid equivocation by the reuse of variables. We used x in the rules given to define f and g . x' is the symbol we used to stand for an arbitrary element of \mathbb{R} which we were plugging into the rule.

23.3 Examples

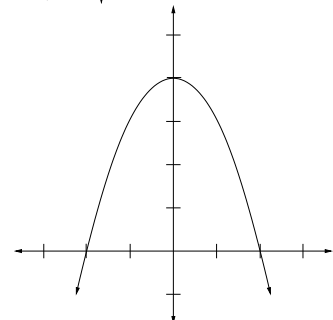
$$f(x) = 3x + 4$$

Domain: \mathbb{R}
 Codomain: \mathbb{R}
 Range: \mathbb{R}



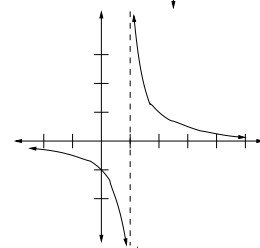
$$f(x) = 4 - x^2$$

Domain: \mathbb{R}
 Codomain: \mathbb{R}
 Range: $(-\infty, 4]$



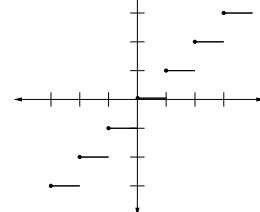
$$f(x) = \frac{1}{x-1}$$

Domain: $(-\infty, 1) \cup (1, \infty)$
 Codomain: \mathbb{R}
 Range: $(-\infty, 0) \cup (0, \infty)$



$$f(x) = \lfloor x \rfloor$$

Domain: \mathbb{R}
 Codomain: \mathbb{R} or \mathbb{Z}
 Range: \mathbb{Z}



$$f(x) = 5$$

Domain: \mathbb{R} or \mathbb{Z}
 Codomain: \mathbb{R} or \mathbb{Z}
 Range: $\{5\}$

$$P(x) = x > -5 \wedge x < 3$$

Domain: \mathbb{R} or \mathbb{Z}
 Codomain: $\{\text{true}, \text{false}\}$
 Range: $\{\text{true}, \text{false}\}$

constant function

Notice that in some cases, the domain codomain are open to interpretation. Accordingly, in exercises like the previous examples you will be asked to give a “reasonable” domain and codomain. A function like $f(x) = 5$ whose range is a set with a single element is called a *constant function*. We can now redefine the term *predicate* to mean a function whose codomain is $\{\text{true}, \text{false}\}$. Though not displayed above, notice that the identity relation on a set is a function. Finally, you may recall seeing some functions with more than one argument. Our simple definition of function still works in this case if we consider the domain of the function to be a Cartesian product. That is, $f(4, 12)$ is simply f applied to the tuple $(4, 12)$. This, in fact, is exactly how ML treats functions apparently with more than one argument.

23.4 Representation

It is almost awkward to introduce functions in ML, since by now they are like a well-known friend to you. However, you still have much to learn about each other. You have been told to think of an ML function as a *parameterized expression*. Nevertheless, we are primarily interested in using ML to *represent* the mathematical objects we discuss, and the best way to represent a function is with, well, a function. Let us take the parabola example and a new, more subtle curve:

$$g(x) = \begin{cases} 0 & \text{if } x = 1 \\ \frac{x^2-1}{x-1} & \text{otherwise} \end{cases}$$

```
- fun f(x) = 4.0 - x*x;

val f = fn : real -> real

- fun g(x) = if x <= 1.0 andalso x >= 1.0
=           then 0.0
=           else (x * x - 1.0) / (x - 1.0);

val g = fn : real -> real
```

The type `real -> real` corresponds to what we write $\mathbb{R} \rightarrow \mathbb{R}$. More importantly, the fact that a function has a type emphasizes that it is a *value* (as we have seen, a *first class* value no less). An identifier like f is simply a variable that happens to store a value that is a function. As we saw with the function `reflexiveClosure`, it can be used in an expression without *applying* it to any arguments.

apply

```
- f;

val it = fn : real -> real

- it(5.0);

val it = ~21.0 : real
```

Once the value of `f` has been saved to `it`, `it` can be used as a function. In fact, we need not store a function in a variable; to write an anonymous function, use the form

`fn (<identifier>) => <expression>`

Thus we have

```
- fn (x) => (x + 3) mod 5;
```

```
val it = fn : int -> int
```

```
- it(15);
```

```
val it = 3 : int
```

or even

```
- (fn (x) => (x + 3) mod 5) (15);
```

```
val it = 3 : int
```

Note that

$$\text{fun } \langle \text{identifier} \rangle_1 (\langle \text{identifier} \rangle_2) = \langle \text{expression} \rangle;$$

is (almost—see Chapter 28) equivalent to

$$\text{val } \langle \text{identifier} \rangle_1 = \text{fn } (\langle \text{identifier} \rangle_2) => \langle \text{expression} \rangle;$$

Exercises

1. Complete the on-line function arrow diagrams drills found at www.ship.edu/~deensl/DiscreteMath/flash/ch4/sec4_1/arrowdiagrams.html
2. To convert from degrees Fahrenheit to degrees Celsius, we need to consider that a degree Fahrenheit is $\frac{5}{9}$ of a degree Celsius and that degrees Fahrenheit are “off by 32,” that is, 32° F corresponds to 0° C. Write a function `fahrToCel` to convert from Fahrenheit to Celsius.
3. Redo your solution to Exercise 2 in `val/fn` form.
4. A mathematical *sequence*, for example 1, 2, 4, 8, ..., can

be thought of as a function with \mathbb{W} , \mathbb{N} , or a finite subset of one of these as the domain. Conventional notation would say $a_0 = 1, a_1 = 2, a_k = 2^k$. Since a list can also be interpreted as a (finite) sequence, a list therefore can also be interpreted as a function. Write a function that takes a list and an integer n and returns the n th element of the list. It is unspecified how your function should behave if given an n outside the size of the list.

5. Write a function that takes a list and returns a function that takes an integer n and returns the n th item in the list. This essentially converts the list into a function.

Chapter 24

Images

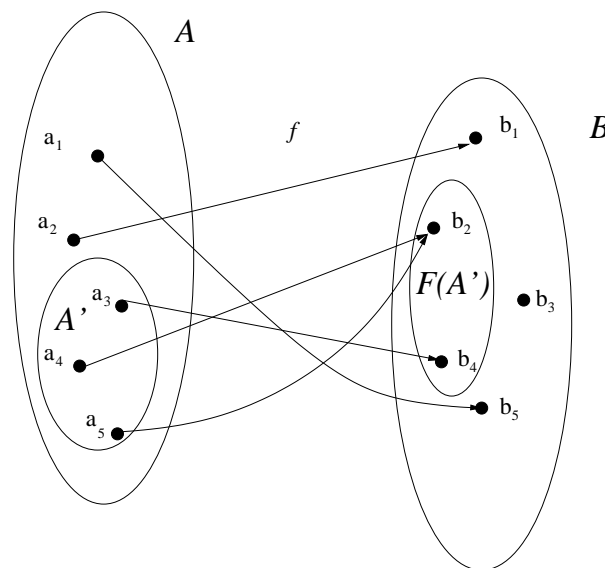
24.1 Definition

We noticed in the previous chapter that a function's range may be a proper subset of its codomain; that is, a function may take its domain to a smaller set than the set it is theoretically defined to. This leads us to consider the idea of a function mapping a set (rather than just a single element). A function will map a subset of the domain to a subset of the codomain. Suppose $f : X \rightarrow Y$ and $A \subseteq X$. The *image* of A under f is

image

$$F(A) = \{y \in Y \mid \exists x \in A \text{ such that } f(x) = y\}$$

The image of a subset of the domain is the set of elements “hit” by elements in the subset. Note that we capitalize the name of the function when we are using it to map an image. This is not standard notation (typically you would see the same name of the function itself being used, merely applied to a set instead of to an element), but it will help you identify more readily when the image is being spoken of. In the following diagram, let $A' = \{a_3, a_4, a_5\}$. Then $F(A') = \{b_2, b_4\}$.



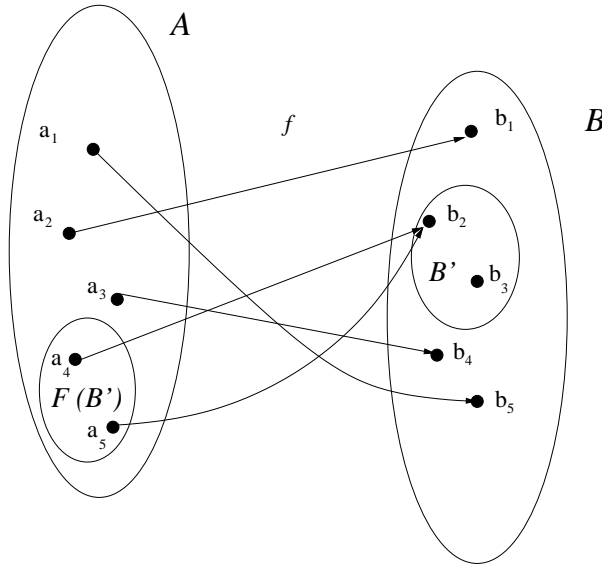
The *inverse image* of a set $B \subseteq Y$ under a function $f : X \rightarrow Y$ is

inverse image

$$F^{-1}(Y) = \{x \in X \mid f(x) \in Y\}$$

The image of a subset of the codomain is the set of elements that hit the subset. It is vital to

remember that although the image is defined by a set in the domain, it *is* a set in the *codomain*, and although the inverse image is defined by a set in the codomain, it *is* a set in the *domain*. It is also important to be able to distinguish an inverse image from an inverse (of a) function, which we will meet in Chapter 25. Let $B' = \{b_2, b_3\}$. Then $F^{-1}(B') = \{a_4, a_5\}$. Notice that $F^{-1}(\{b_3\}) = \emptyset$.



24.2 Examples

Proofs of propositions involving images and inverse images are a straightforward matter of applying definitions. However, students frequently have trouble with them, possibly because they are used to thinking about functions operating on elements rather than on entire sets of elements. The important thing to remember is that *images and inverse images are sets*. Therefore, you must put the techniques you learned for set proofs to work.

Theorem 24.1 Let $f : X \rightarrow Y$ and $A, B \subseteq X$. Then $F(A \cup B) = F(A) \cup F(B)$.

Do not be distracted by the new definitions. This is a proof of set equality, Set Proof Form 2. Moreover, $F(A \cup B) \subseteq Y$. Choose your variable names in a way that shows you understand this.

Proof. Suppose $y \in F(A \cup B)$. By the definition of image, there exists an $x \in A \cup B$ such that $f(x) = y$. By the definition of union, $x \in A$ or $x \in B$. Suppose $x \in A$. Then, again by the definition of image, $y \in F(A)$. Similarly, if $x \in B$, then $y \in F(B)$. Either way, by definition of union, $y \in F(A) \cup F(B)$. Hence $F(A \cup B) \subseteq F(A) \cup F(B)$ by definition of subset.

Next suppose $y \in F(A) \cup F(B)$. By definition of union, either $y \in F(A)$ or $F(B)$. Suppose $y \in F(A)$. By definition of image, there exists an $x \in A$ such that $f(x) = y$. By definition of union, $x \in A \cup B$. Again by definition of image, $y \in F(A \cup B)$. The argument is similar if $y \in F(B)$. Hence by definition of subset, $F(A) \cup F(B) \subseteq F(A \cup B)$.

Therefore, by definition of set equality, $F(A \cup B) = F(A) \cup F(B)$. \square

The inverse image also may look intimidating, but reasoning about them is still a matter of set manipulation. Just remember that an inverse image is a subset of the domain.

Theorem 24.2 Let $f : X \rightarrow Y$ and $A, B \subseteq Y$. Then $F^{-1}(A - B) = F^{-1}(A) - F^{-1}(B)$.

Proof. Suppose $x \in F^{-1}(A - B)$. By definition of inverse image, $f(x) \in A - B$. By definition of set difference, $f(x) \in A$ and $f(x) \notin B$. Again by definition of inverse image, $x \in F^{-1}(A)$ and $x \notin F^{-1}(B)$. Again by definition of set difference, $x \in F^{-1}(A) - F^{-1}(B)$. Hence, by definition of subset, $F^{-1}(A - B) \subseteq F^{-1}(A) - F^{-1}(B)$.

Next suppose $x \in F^{-1}(A) - F^{-1}(B)$. By definition of set difference, $x \in F^{-1}(A)$ and $x \notin F^{-1}(B)$. By definition of inverse image, $f(x) \in A$ and $f(x) \notin B$. Again by definition of set difference $f(x) \in A - B$. Again by definition of inverse image, $x \in F^{-1}(A - B)$. Hence, by definition of subset, $F^{-1}(A) - F^{-1}(B) \subseteq F^{-1}(A - B)$.

Therefore, by definition of set equality, $F^{-1}(A - B) = F^{-1}(A) - F^{-1}(B)$. \square

These are classic examples of the analysis/synthesis process. We take apart expressions by definition, and by definition we reconstruct other expressions.

24.3 Map

Frequently it is useful to apply an operation over an entire collection of data, and therefore to get a collection of results. For example, suppose we wanted to square every element in a list of integers.

```
- fun square([]) = nil
=   | square(x::rest) = x*x :: square(rest);

val square = fn : int list -> int list

- square([1,2,3,4,5]);

val it = [1,4,9,16,25] : int list
```

We can generalize this pattern using the fact that functions are first class values. A program traditionally called `map` takes a function and a list and applies that function to the entire list.

```
- fun map(func, []) = nil
=   | map(func, x::rest) = func(x) :: map(func, rest);

val map = fn : ('a -> 'b) * 'a list -> 'b list

- map(fn (x) => x*x, [1,2,3,4,5]);

val it = [1,4,9,16,25] : int list
```

Keep in mind that we are considering lists in general, not lists as representing sets. However, `map` very naturally adapts to our notion of image, using the list representation of sets.

```
- fun image(f, set) = makeNoRepeats(map(f, set));
```

Exercises

In Exercises 1–11, assume $f : X \rightarrow Y$. Prove, unless you are asked to give a counterexample.

1. If $A, B \subseteq X$, $F(A \cap B) \subseteq F(A) \cap F(B)$.
2. If $A, B \subseteq X$, $F(A \cap B) = F(A) \cap F(B)$. This is false; give a counterexample.
3. If $A, B \subseteq X$, $F(A) - F(B) \subseteq F(A - B)$.
4. If $A, B \subseteq X$, $F(A - B) \subseteq F(A) - F(B)$. This is false; give a counterexample.
5. If $A \subseteq B \subseteq Y$, then $F^{-1}(A) \subseteq F^{-1}(B)$.
6. If $A, B \subseteq Y$, then $F^{-1}(A \cup B) = F^{-1}(A) \cup F^{-1}(B)$.
7. If $A, B \subseteq Y$, then $F^{-1}(A \cap B) = F^{-1}(A) \cap F^{-1}(B)$.
8. If $A \subseteq X$, $A \subseteq F^{-1}(F(A))$.
9. If $A \subseteq X$, $A = F^{-1}(F(A))$. This is false; give a counterexample.
10. If $A \subseteq Y$, $F(F^{-1}(A)) \subseteq A$.
11. If $A \subseteq Y$, $F(F^{-1}(A)) \subseteq A$. This is false; give a counterexample.
12. Sometimes it is useful to write a program that operates on a list but also takes some extra arguments. For example a program **scale** might take a list of integers and another integer and return a list of the old integers multiplied by the other integer. Write a program **mapPlus** that takes a function, a list, and an extra argument and applies the function to every element in the list and the extra argument. For example, `mapPlus(fn (x, y) => x * y, [1, 2, 3, 4], 2)` would return `[2, 4, 6, 8]`.
13. Write a program **mapPlusMaker** that takes a function and returns a function that will take a list and an extra argument and apply the given function to all the elements in the list. For example, `mapPlusMaker(fn (x, y) => x * y)` would return the **scale** program described above.

Chapter 25

Function properties

25.1 Definitions

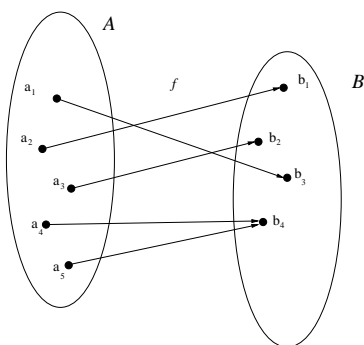
Some functions have certain properties that imply that they behave in predictable ways. These properties will come in particularly handy in the next chapter when we consider the composition of functions.

We have seen examples of functions where some elements in the codomain are hit more than once, and functions where some are hit not at all. Two properties give names for (the opposite) of these situations. A function $f : X \rightarrow Y$ is *onto* if for all $y \in Y$, there exists an $x \in X$ such that $f(x) = y$. In other words, an onto function hits every element in the codomain (possibly more than once). If $B \subseteq Y$ and for all $y \in B$ there exists an $x \in X$ such that $f(x) = y$, then we say that f is *onto B*. A function is *one-to-one* if for all $x_1, x_2 \in X$, if $f(x_1) = f(x_2)$, then $x_1 = x_2$. If any two domain elements hit the same codomain element, they must be equal (compare the structure of this definition with the definition of antisymmetry); this means that no element of the codomain is hit more than once. A function that is both one-to-one and onto is called a *one-to-one correspondence*; in that case, every element of the codomain is hit exactly once.

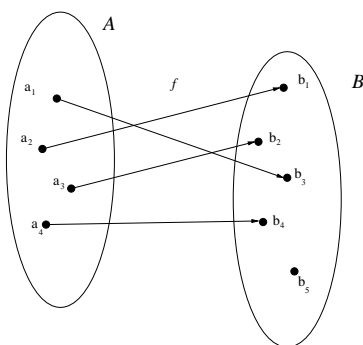
onto

one-to-one

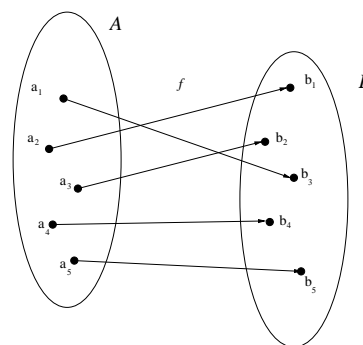
*one-to-one
correspondence*



Onto but not one-to-one



One-to-one but not onto



One-to-one correspondence

Sometimes onto functions, one-to-one functions, and one-to-one correspondences are called *surjections*, *injections*, and *bijections*, respectively.

Last time you proved that $F(A \cap B) \subseteq F(A) \cap F(B)$ but gave a counterexample against $F(A) \cap F(B) \subseteq F(A \cap B)$. If you look back at that counterexample, you will see that the problem is that your f is not one-to-one. Thus

Theorem 25.1 *If $f : X \rightarrow Y$, $A, B \subseteq X$, and f is one-to-one, then $F(A) \cap F(B) \subseteq F(A \cap B)$.*

Proof. Suppose $f : X \rightarrow Y$, $A, B \subseteq X$, and f is one-to-one.

Now suppose $y \in F(A) \cap F(B)$. Then $y \in F(A)$ and $y \in F(B)$ by definition of intersection. By the definition of image, there exist $x_1 \in A$ such that $f(x_1) = y$ and $x_2 \in B$

such that $f(x_2) = y$. By definition of one-to-one, $x_1 = x_2$. By substitution, $x_1 \in B$. By definition of intersection, $x_1 \in A \cap B$. Therefore, by definition of image, $y \in F(A \cap B)$. \square

Note that with $y \in F(A) \cap F(B)$, we had to assume *two* x 's, one in A and one in B . Only then could we apply the fact that f is one-to-one and prove the two x 's equal. *This is extremely important.* Notice also that by putting this result and Exercise 1 of Chapter 24 together, we conclude that if f is one-to-one, $F(A \cap B) = F(A) \cap F(B)$.

25.2 Cardinality

If a function $f : X \rightarrow Y$ is onto, then every element in Y has at least one domain element seeking it, but two elements in X could be rivals for the same codomain element. If it is one-to-one, then every domain element has a codomain element all to itself, but some codomain elements may be left out. If it is a one-to-one correspondence, then everyone has a date to the dance. When we are considering finite sets, we can use this as intuition for comparing the cardinalities of the domain and codomain. For example, f could be onto only if $|X| \geq |Y|$ and one-to-one only if $|X| \leq |Y|$. If f is a one-to-one correspondence, then it must be that $|X| = |Y|$.

How could we prove this, though? In fact, we have never given a formal definition of cardinality. The one careful proof we did involving cardinality was Theorem 15.1 which relies in part on this chapter. Rather than proving that the existence of a one-to-one correspondence implies sets of equal cardinality, we simply define cardinality so. Two finite sets X and Y have the same *cardinality* if there exists a one-to-one correspondence from X to Y . We write $|X| = n$ for some $n \in \mathbb{N}$ if there exists a one-to-one correspondence from $\{1, 2, \dots, n\}$ to X , and define $|\emptyset| = 0$.

(There is actually one wrinkle in this system: it lacks a formal definition of the term *finite*. Technically, we should define a set X to be finite if there exists an $n \in \mathbb{N}$ and a one-to-one correspondence from $\{1, 2, \dots, n\}$ to X . Then, however, we would need to use this to define *cardinality*. We would prefer to keep the definition of cardinality separate from the notion of finite subsets of \mathbb{N} to make it easier to extend the concepts to infinite sets later. We are also assuming that a set's cardinality is unique, or that the cardinality operator $|\cdot|$ is well-defined as a function. This can be proven, but it is difficult.)

Now we can use cardinality formally.

Theorem 25.2 *If A and B are finite, disjoint sets, then $|A \cup B| = |A| + |B|$.*

Proof. Suppose A and B are finite, disjoint sets. By the definition of finite, there exist $i, j \in \mathbb{N}$ and one-to-one correspondences $f : \{1, 2, \dots, i\} \rightarrow A$ and $g : \{1, 2, \dots, j\} \rightarrow B$. Note that $|A| = i$ and $|B| = j$. Define a function $h : \{1, 2, \dots, i+j\} \rightarrow A \cup B$ as

$$h(x) = \begin{cases} f(x) & \text{if } x \leq i \\ g(x-i) & \text{otherwise} \end{cases}$$

Now suppose $y \in A \cup B$. Then either $y \in A$ or $y \in B$ by definition of union, and it is not true that $y \in A$ and $y \in B$ by definition of disjoint. Hence we have two cases:

Case 1: Suppose $y \in A$ and $y \notin B$. Then, since f is onto, there exists a $k \in \{1, 2, \dots, i\}$ such that $f(k) = y$. By our definition of h , $h(k) = y$. Further, suppose $\ell \in \{1, 2, \dots, i+j\}$ and $h(\ell) = y$. Suppose $\ell > i$; then $y = h(\ell) = g(\ell - i) \in B$, contradiction; hence $\ell \leq i$. This implies $h(\ell) = f(\ell)$, and since f is one-to-one, $\ell = k$.

Case 2: Suppose $y \in B$ and $y \notin A$. Then, since g is onto, there exist a $k \in \{1, 2, \dots, j\}$ such that $g(k) = y$. By our definition of h , $h(k+i) = g(k) = y$. Further, suppose $\ell \in \{1, 2, \dots, i+j\}$ and $h(\ell) = y$. Suppose $\ell \leq i$; then $y = h(\ell) = f(\ell) \in A$, contradiction; hence $\ell > i$. This implies $h(\ell) = g(\ell - i)$, and since g is one-to-one, $\ell - i = k$ or $\ell = k + i$.

In either case, there exists a unique element in $m \in \{1, 2, \dots, i+j\}$ ($m = k$ and $m = k+i$, respectively) such that $h(m) = y$. Hence h is a one-to-one correspondence. Therefore, $|A \cup B| = i + j = |A| + |B|$. \square

cardinality

25.3 Inverse functions

Our work with images and inverse images, particularly problems like proving $A \subseteq F^{-1}(F(A))$, force us to think about functions as taking us on a journey from a place in one set to a place in another. The inverse image addressed the complexities of the return voyage. Suppose $f : X \rightarrow Y$ were not onto, specifically that for some $x_1, x_2 \in X$ and $y \in Y$, both $f(x_1) = y$ and $f(x_2) = y$. $f(x_1)$ will take us to y (or $F(\{x_1\})$ will take us to $\{y\}$), but we have in a sense lost our way: Going backwards through f might take us to x_2 instead of x_1 ; $F^{-1}(y) = \{x_1, x_2\} \neq \{x_1\}$. If f is one-to-one, this is not a problem because only one element will take us to y ; that way, if we start in X and go to Y , we know we can always retrace our steps back to where we were in X . However, this does not help if we start in Y and want to go to X ; unless f is also onto, there may not be a way from X to that place in Y .

Accordingly, if $f : X \rightarrow Y$ is a one-to-one correspondence, we define the *inverse* of f , to be the relation

inverse

$$f^{-1} = \{(y, x) \in Y \times X \mid f(x) = y\}$$

It is more convenient to call this the *inverse function* of f , but the title “function” does not come for free.

inverse function

Theorem 25.3 *If $f : X \rightarrow Y$ is a one-to-one correspondence, then $f^{-1} : Y \rightarrow X$ is well-defined.*

Proof. Suppose $y \in Y$. Since f is onto, there exists $x \in X$ such that $f(x) = y$. Hence $(y, x) \in f^{-1}$ or $f^{-1}(y) = x$.

Next suppose $(y, x_1), (y, x_2) \in f^{-1}$ or $f^{-1}(y) = x_1$ and $f^{-1}(y) = x_2$. Then $f(x_1) = y$ and $f(x_2) = y$. Since f is one-to-one, $x_1 = x_2$.

Therefore, by definition of function, f^{-1} is well-defined. \square

Do not confuse the inverse of a function and the inverse image of a function. Remember that the inverse image is a set, a subset of the domain, applied to a subset of the codomain; the inverse image always exists. The inverse function exists only if the function itself is a one-to-one correspondence; it takes an element of the codomain and produces an element of the domain.

As an application of functions and the properties discussed here, we consider an important concept in information security. A *hash function* is a function that takes a string (that is, a variable-length sequence of characters) and returns fixed-length string. Since the output is smaller than the input, a hash function could not be one-to-one. However, a good hash function (often called a one-way hash function) should have the following properties:

hash function

- It should be very improbable for two arbitrary input strings to produce the same output. Obviously some strings will map to the same output, but such pairs should be very difficult to find. In this way, the function should be “as one-to-one as possible” and any collisions should happen without any predictable pattern.
- It should be impossible to approximate an inverse for it. Since it is not one-to-one, a true inverse is impossible, but in view here is that given an output string, it should be very difficult to produce *any* input string that could be mapped to it.

The idea is that a hash function can be used to produce a fingerprint of a document or file which proves the document’s existence without revealing its contents. Suppose you wanted to prove to someone that you have a document that they also have, but you do not want to make that document public. Instead, you could compute the hash of that document and make that public. All those who have the document already can verify the hash, but no one who did not have the document could invert the hash to reproduce it. Another use is time stamping. Suppose you have a document that contains intellectual property (say, a novel or a blueprint) for which you want to ensure that you get credit. You could compute the hash of the document and make the hash public (for example, printing it in the classified ads of a newspaper); some time later, when you make the document itself public, you will have proof that you knew its contents on or before the date the hash was published.

Exercises

1. Which of the following are one-to-one and/or onto? (Assume a domain and range of \mathbb{R} .)

- (a) $f(x) = x$.
- (b) $f(x) = x^2$.
- (c) $f(x) = \ln x$.
- (d) $f(x) = x^3$.
- (e) $f(x) = \frac{1}{3}x^3 - 4x$.

In Exercises 2–9, assume $f : X \rightarrow Y$. Prove.

2. If $A, B \in X$ and f is one-to-one, then $F(A - B) \subseteq F(A) - F(B)$.

- 3. If $A \in X$ and f is one-to-one, then $F^{-1}(F(A)) \subseteq A$.
- 4. If $A \in Y$ and f is onto, then $A \subseteq F(F^{-1}(A))$.
- 5. If f is onto, then the range of f is Y .
- 6. If for all non-empty $A, B \subseteq X$, $F(A \cap B) = F(A) \cap F(B)$, then f is a one-to-one correspondence.
- 7. If f is one-to-one, then $|X| \leq |Y|$.
- 8. Let A be a set. i_A is a one-to-one correspondence.
- 9. If f is onto, then $|X| \geq |Y|$.
- 10. If $A \subseteq B$, then $|B - A| = |B| - |A|$.
- 11. If A is a finite set and $a \in A$, then $|\{ \{a\} \cup A' \mid A' \in \mathcal{P}(A - \{a\}) \}| = |\mathcal{P}(A - \{a\})|$.

Chapter 26

Function composition

26.1 Definition

We have seen that two relations can be composed to form a new relation, say, given relations R from X to Y and S from Y to Z :

$$R \circ S = \{(a, c) \in X \times Z \mid \exists b \in Y \text{ such that } (a, b) \in R \text{ and } (b, c) \in S\}$$

This easily can be specialized for the composition of functions. Suppose $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ are functions. Then the *composition* of f and g is

function composition

$$g \circ f = \{(x, z) \in X \times Z \mid z = g(f(x))\}$$

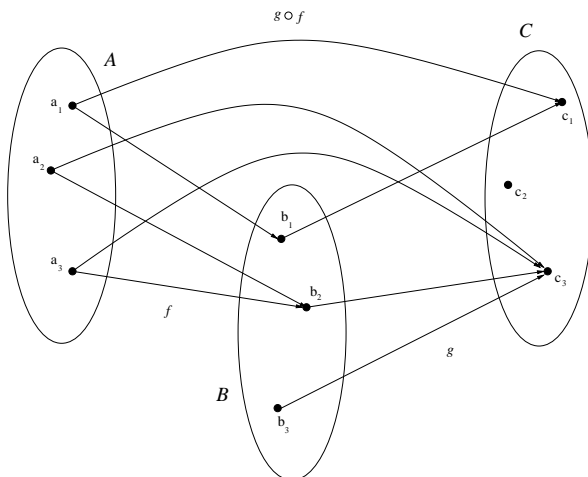
This is merely a rewriting of the definition of composition of relations. The interesting part is that $g \circ f$ is a function.

Theorem 26.1 *If $f : A \rightarrow B$ and $g : B \rightarrow C$ are functions, then $g \circ f : A \rightarrow C$ is well-defined.*

Proof. Suppose $a \in A$. Since f is a function, there exists a $b \in B$ such that $f(a) = b$. Since g is a function, there exists a $c \in C$ such that $g(b) = c$. By definition of composition, $(a, c) \in g \circ f$, or $g \circ f(a) = c$.

Next suppose $(a, c_1), (a, c_2) \in g \circ f$, or $g \circ f(a) = c_1$ and $g \circ f(a) = c_2$. By definition of composition, there exist b_1, b_2 such that $f(a) = b_1, f(a) = b_2, g(b_1) = c_1$, and $g(b_2) = c_2$. Since f is a function, $b_1 = b_2$. Since g is a function, $c_1 = c_2$.

Therefore, by definition of function, $g \circ f$ is well-defined. \square



The most intuitive way to think of composition is to use the machine model of functions. We simply attach two machines together, feeding the output slot of the one into the input slot of the other. To make this work, the one output slot must fit into the other input slot, and the one machine's output material must be appropriate input material for the other machine. Mathematically, we describe this by considering the domains and codomains. Given $f : A \rightarrow B$ and $g : C \rightarrow D$, $g \circ f$ is defined only for $B = C$, though it could easily be extended for the case where $B \subseteq C$.

Function composition happens quite frequently without our noticing it. For example, the real-valued function $f(x) = \sqrt{x-12}$ can be considered the composition of the functions $g(x) = x-12$ and $h(x) = \sqrt{x}$.

26.2 Functions as components

In ML, a function application is like any other expression. It can be the argument to another function application, and in this way we do function composition. The sets used for domain and codomain are represented by types, and so when doing function composition, the types must match. The important matter is to consider functions as components. Our approach to building algorithms from this point on is the defining and applying of functions, like so many interlocking machines.

For our example, suppose we want to write a program that will predict the height of a tree after a given number of years. The prediction is calculated based on three pieces of information: the variety of tree (assume that this determines the tree's growth rate), the number of years, and the initial height of the tree. Obviously this assumes tree growth is linear, as well as making other assumptions, with nothing based on any real botany.

Suppose we associate the following growth rates in terms of units per month for the following varieties of tree.

```
- datatype treeGenus = Oak | Elm | Maple | Spruce | Fir | Pine | Willow;

datatype treeGenus = Elm | Fir | Maple | Oak | Pine | Spruce | Willow

- fun growthRate(Elm) = 1.3
=   | growthRate(Maple) = 2.7
=   | growthRate(Oak) = 2.4
=   | growthRate(Pine) = 0.4
=   | growthRate(Spruce) = 2.9
=   | growthRate(Fir) = 1.1
=   | growthRate(Willow) = 5.3;

val growthRate = fn : treeGenus -> real
```

`growthRate` is a function mapping tree genera to their rates of growth. We have overlooked something, though. There is a larger categorization of these trees that will affect how this growth rate is applied: Coniferous trees grow all year round, but deciduous trees are inactive during the winter. The number of months used for growing, therefore, is a function of the taxonomic division.

```
- datatype treeDivision = Deciduous | Coniferous;

datatype treeDivision = Coniferous | Deciduous

- fun growingMonths(Deciduous) = 6.0
=   | growingMonths(Coniferous) = 12.0;

val growingMonths = fn : treeDivision -> real
```


Because of the hierarchy of taxonomy, we can determine a tree's division based on its genus. To avoid making the mapping longer than necessary, we pattern-match on coniferous trees and make deciduous the default case.

```
- fun division(Pine) = Coniferous
=   | division(Spruce) = Coniferous
=   | division(Fir) = Coniferous
=   | division(x) = Deciduous;

val division = fn : treeGenus -> treeDivision
```

Since `division` maps `treeGenus` to `treeDivision` and `growingMonths` maps `treeDivision` to `real`, their composition maps `treeGenus` to `real`.

The predicted height of the tree is of course the initial height plus the growth rate times growth time. The growth rate is calculated from the genus, and the growth time is calculated by multiplying years times the growing months. Thus we have

```
- fun predictHeight(genus, initial, years) =
=   initial + (years * growingMonths(division(genus))
=   * growthRate(genus));

val predictHeight = fn : treeGenus * real * real -> real
```

We can generalize this composition process by writing a function that takes two functions and returns a composed function. Just for the sake of being fancy, we can use it to rewrite the expression `growingMonths(division(genus))`.

```
- fun compose(f, g) = fn (x) => g(f(x));

val compose = fn : ('a -> 'b) * ('b -> 'c) -> 'a -> 'c

- fun predictHeight(genus, initial, years) =
=   initial + (years * compose(division, growingMonths)(genus)
=   * growthRate(genus));

val predictHeight = fn : treeGenus * real * real -> real
```

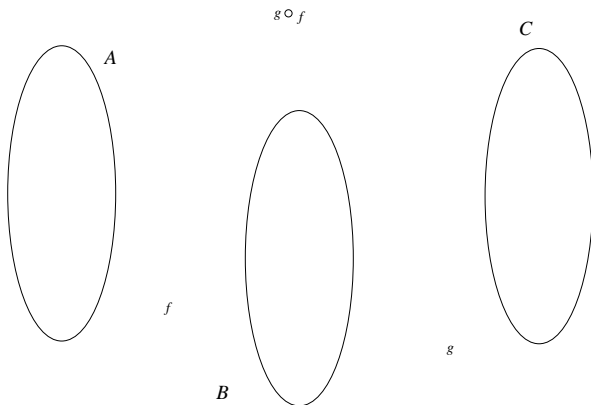
26.3 Proofs

Composition finally gives us enough raw material to prove some fun results on functions. Remember in all of these to apply the definitions carefully and also to follow the outlines for proving propositions in now-standard forms. For example, it is easy to verify visually that the composition of two one-to-one functions is also a one-to-one function. If no two f or g arrows collide, then no $g \circ f$ arrows have a chance of colliding. Proving this result relies on the definitions of composition and one-to-one.

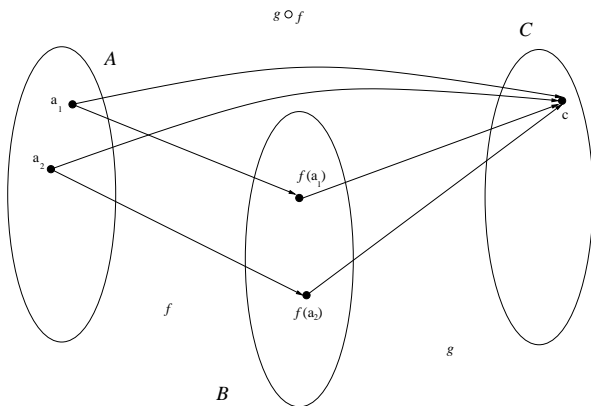
Theorem 26.2 *If $f : A \rightarrow B$ and $g : B \rightarrow C$ are one-to-one, then $g \circ f : A \rightarrow C$ is one-to-one.*

Proof. Suppose $f : A \rightarrow B$ and $g : B \rightarrow C$ are one-to-one. Now suppose $a_1, a_2 \in A$ and $c \in C$ such that $g \circ f(a_1) = c$ and $g \circ f(a_2) = c$. By definition of composition, $g(f(a_1)) = c$ and $g(f(a_2)) = c$. Since g is one-to-one, $f(a_1) = f(a_2)$. Since f is one-to-one, $a_1 = a_2$. Therefore, by definition of one-to-one, $g \circ f$ is one-to-one. \square

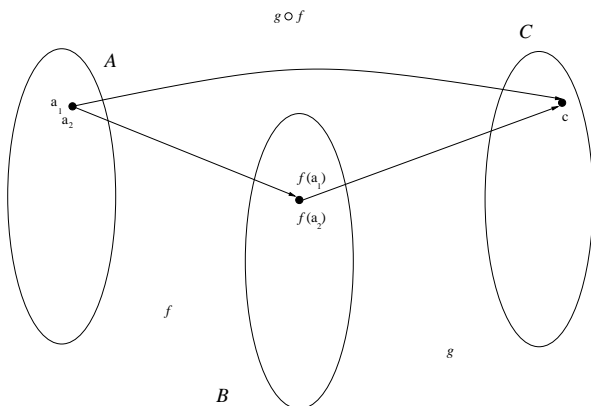
In the long run, we want to prove that something is one-to-one, so we need to gather the materials to synthesize the definition. It means we can pick any two elements from the domain of $g \circ f$, and if they map to the same element, they themselves must be the same. Here is how the proof connects with a visual verification.



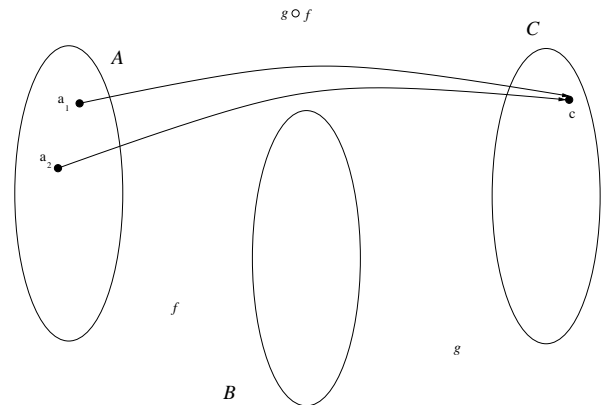
Suppose $f : A \rightarrow B$ and $g : B \rightarrow C$ are one-to-one.



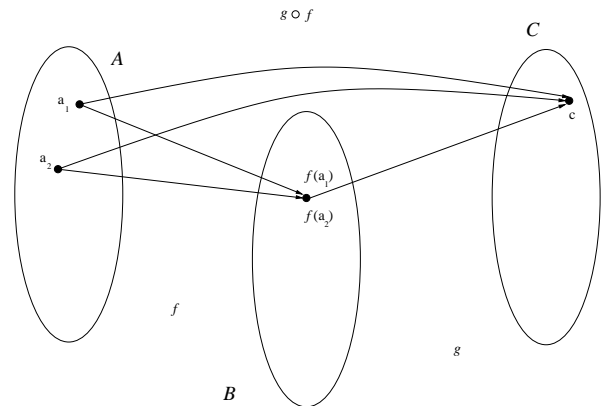
By definition of composition, $g(f(a_1)) = c$ and $g(f(a_2)) = c$.



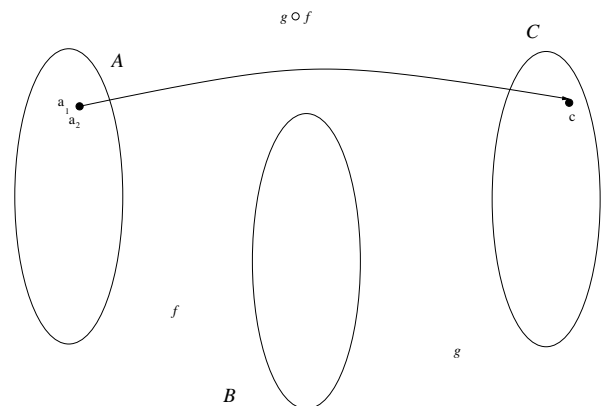
Since f is one-to-one, $a_1 = a_2$.



Now suppose $a_1, a_2 \in A$ and $c \in C$ such that $g \circ f(a_1) = c$ and $g \circ f(a_2) = c$.



Since g is one-to-one, $f(a_1) = f(a_2)$.



Therefore, by definition of one-to-one, $g \circ f$ is one-to-one. \square

Our intuition about inverses from the previous chapter conceived of functions as taking us from a spot in one set to a spot in another. Only if the function were a one-to-one correspondence could we assume a “round trip” (using, an inverse function) existed. Since the net affect of a round trip is getting you nowhere, you may conjecture that if you compose a function with its inverse, you will get an identify function. The converse is true, too.

Theorem 26.3 *Suppose $f : A \rightarrow B$ is a one-to-one correspondence and $g : B \rightarrow A$. Then $g = f^{-1}$ if and only if $g \circ f = i_A$.*

Proof. Suppose that $g = f^{-1}$. (Since f is a one-to-one correspondence, f^{-1} is well-defined.) Suppose $a \in A$. Then

$$\begin{aligned} g \circ f(a) &= g(f(a)) && \text{by definition of composition} \\ &= f^{-1}(f(a)) && \text{by substitution} \\ &= a && \text{by definition of inverse function} \end{aligned}$$

Therefore, by function equality, $g \circ f = i_A$.

For the converse (proving that if you compose a function and get the identity function, it must be the inverse), see Exercise 9. \square

Exercises

1. Write functions to compute the circumference and area of a circle based on radius, the surface area and volume of a sphere based on radius, and the surface area and volume of a cylinder based on radius of base and height. Use **real** values and reuse functions as much as possible.
2. Make a data type that represents a set of six of your friends. Then write a function `cpo` that maps these friends to their CPO (campus post office) address. At the milbox section of the CPO, boxes 1–2039 are on the west wall, 2040–2579 are on the north wall, and 2560–3299 are on the east wall. Make a data type for the set of these three walls, and write a function that maps CPO address to their appropriate wall. Then write a function (using the two functions you just wrote) that maps your friends to the wall where one can find their box.

Prove.

3. If $f : A \rightarrow B$, then $f \circ i_A = f$.
4. If $f : A \rightarrow B$, then $i_B \circ f = f$.
5. If $f : A \rightarrow B$, $g : B \rightarrow C$, and $h : C \rightarrow D$, then $h \circ (g \circ f) = (h \circ g) \circ f$.
6. If $f : A \rightarrow B$ and $g : B \rightarrow C$ are both onto, then $g \circ f$ is also onto.
7. If $f : A \rightarrow B$ and $g : B \rightarrow C$, and $g \circ f$ is one-to-one, then f is one-to-one.
8. If $f : A \rightarrow B$ and $g : B \rightarrow C$, and $g \circ f$ is onto, then g is onto.
9. If $f : A \rightarrow B$ is a one-to-one correspondence, $g : B \rightarrow A$, and $g \circ f = i_A$, then $g = f^{-1}$.
10. If $f : A \rightarrow B$, $g : A \rightarrow B$, $h : B \rightarrow C$, h is one-to-one, and $h \circ f = h \circ g$, then $f = g$.
11. If $f : A \rightarrow B$ and $g : B \rightarrow C$ are both one-to-one correspondences, then the inverse function $(g \circ f)^{-1}$ exists and $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$. (Hint: Notice that this requires you to prove two things. However, the first thing can be proven quickly from previous exercises in this chapter.)
12. Let $G \circ F(X)$ be the image of a set X under the function $g \circ f$, for functions f and g ; similarly define inverse images for composed functions. If $f : A \rightarrow B$, $g : B \rightarrow C$ and $X \subseteq C$, then $(G \circ F)^{-1}(X) = F^{-1}(G^{-1}(X))$.
13. Let B be a set of subsets of a set A (for example, B could be $\mathcal{P}(A)$). Cardinality (that is, the relation R on B such that $(X, Y) \in R$ if $|X| = |Y|$) is an equivalence relation.

Chapter 27

Special Topic: Countability

Both our informal definition of cardinality in Chapter 3 and the more careful one in Chapter 25 were restricted to finite sets. This was in deference to an unspoken assumption that the cardinality of a set ought to be *something*, that is, a whole number. As has been mentioned already, we cannot merely say that a set like \mathbb{Z} has cardinality infinity. Infinity is not a whole number—or even a number at all, if one has in mind the number sets \mathbb{N} , \mathbb{W} , \mathbb{Z} , \mathbb{Q} , \mathbb{R} , and \mathbb{C} . The definition of cardinality taken at face value, however, does not guarantee that the cardinality is something; it merely inspired us to define what the operator $||$ means by comparing a set to a subset of \mathbb{N} . Indeed, the definition of cardinality merely gives us a way to say that two sets *have the same cardinality as each other*. What would happen if we extended this bare notion to infinite sets—that is, drop the term *finite*, which we have not defined anyway?

Two sets X and Y have the same *cardinality* if there exists a one-to-one correspondence from X to Y . We know from Exercise 13 of Chapter 26 that this relation partitions sets into equivalence classes. (Your proof did not depend on the sets being finite, did it?) We say that a set X is *finite* if it is the empty set or if there exists an $n \in \mathbb{W}$ such that X has the same cardinality as $\{1, 2, \dots, n\}$. Otherwise, we say the set is *infinite*.

cardinality

finite

infinite

It is worth remembering that the term *infinite* is defined negatively; this also makes sense etymologically, since the word is simply *finite* with a negative prefix. Did you ever wonder what was so infinite about the grammatical term *infinitive*? The term makes more sense in Latin, where the pronoun subject of a verb is given by a suffix: *ambulo*, *ambulas*, *ambulat* is the conjugation “I walk,” “you walk,” “he/she/it walks,” *o*, *s*, and *t* being the singulars for first, second, and third person, respectively. An infinitive (in this case, *ambulare*) has no pronominal suffix. This does not mean that it goes on forever, just that it, literally, has no proper ending. Since the caboose has fallen out of use in American railways, can we now say that freight trains are therefore infinite? Perhaps not, since even though there is no longer a formal ending car, they are at least terminated by a box called a FRED (which stands for *Flashing Rear-End Device*; no kidding).

On a more serious note, this raises the question, are all infinities equal? More than merely raising the question, it gives a rigorous way to phrase it: Are all infinite sets in the same equivalence class? Our intuition could go either way. On one hand, one might assume that infinity is infinity without qualification. Thus

$$|\mathbb{N}| = |\mathbb{Z}| = |\mathbb{Q}| = |\mathbb{R}|$$

On the other hand, every integer is a rational number, and the real numbers are all the rationals plus a whole many more. Perhaps

$$|\mathbb{N}| < |\mathbb{Z}| < |\mathbb{Q}| < |\mathbb{R}|$$

Which could it be?

Let us ask a more general question first. Is it possible for a proper subset to have the same cardinality as the whole set? Take \mathbb{Z} and \mathbb{N} for example. Both are infinite, but obviously $\mathbb{N} \subseteq \mathbb{Z}$ and $\mathbb{N} \neq \mathbb{Z}$.

Theorem 27.1 \mathbb{N} and \mathbb{Z} have the same cardinality.

This calls for a proof of existence. The definition of cardinality requires that a one-to-one correspondence exists between the two sets. We must either prove that it is impossible for such a function not to exist or propose a candidate function and demonstrate that it meets the requirements. We use the latter strategy.

Proof. Define a function $f : \mathbb{N} \rightarrow \mathbb{Z}$ thus:

$$f(n) = \begin{cases} n \operatorname{div} 2 & \text{if } n \text{ is even} \\ -(n \operatorname{div} 2) & \text{otherwise} \end{cases}$$

Now suppose $n' \in \mathbb{Z}$. We have three cases: $n' = 0$, $n' > 0$, and $n' < 0$. Suppose $n' = 0$; then $f(1) = 1 \operatorname{div} 2 = 0 = n'$. Suppose $n' > 0$; then $f(2n') = 2n' \operatorname{div} 2 = n'$. Suppose $n' < 0$; then $f((-2n') + 1) = -((-2n') + 1) \operatorname{div} 2 = -(-n') = n'$. In each case, there exists an element of \mathbb{N} that maps to n' and hence f is onto.

Next suppose $n_1, n_2 \in \mathbb{Z}$ and $f(n_1) = f(n_2)$. We have three cases: $f(n_1) = 0$, $f(n_1) > 0$, and $f(n_1) < 0$.

Case 1: Suppose $f(n_1) = 0$. Since $-0 = 0$, then by our formula, $n_1 \operatorname{div} 2 = 0$ whether n_1 is even or odd. Then $n_1 = 0$ or $n_1 = 1$, but since $0 \notin \mathbb{N}$, $n_1 = 1$. By substitution and similar reasoning, $n_2 = 1$.

Case 2: Suppose $f(n_1) > 0$. Then n_1 is even. Moreover, by our formula, either $n_1 = 2 \cdot f(n_1)$ or $n_1 = 2 \cdot f(n_1) + 1$. Since n_1 is even, $n_1 = 2 \cdot f(n_1)$. By substitution and similar reasoning, $n_2 = 2 \cdot f(n_1)$.

Case 3: Suppose $f(n_1) < 0$. Then n_1 is odd. Moreover, by our formula, either $n_1 = 2 \cdot (-f(n_1))$ or $n_1 = 2 \cdot (-f(n_1)) + 1$. Since n_1 is odd, $n_1 = 2 \cdot (-f(n_1)) + 1$. By substitution and similar reasoning, $n_2 = 2 \cdot (-f(n_1)) + 1$.

In each case, $n_1 = n_2$. Hence f is onto.

Hence f is a one-to-one correspondence, and therefore \mathbb{Z} has the same cardinality as \mathbb{N} .

□

So it is possible for a proper subset to have as many (infinite) elements as the set that contains it. Moreover, $|\mathbb{N}| = |\mathbb{Z}|$, so the rest of the equality chain seems plausible. But is it true? Before asking again a slightly different question, we say that a set X is *countably infinite* if X has the same cardinality as \mathbb{N} . A set X is *countable* if it is finite or countably infinite. The idea is that we could count every element in the set by assigning a number 1, 2, 3, ... to each one of them, even if it took us forever. A set X is *uncountable* if it is not countable. This gives us a new question: Are all sets countable?

Let us try \mathbb{Q} next. The jump from \mathbb{N} to \mathbb{Z} was not so shocking since \mathbb{Z} has only about “twice” as many elements as \mathbb{N} . \mathbb{Q} , however, has an infinite number of elements between 0 and 1 alone, and an infinite number again between 0 and $\frac{1}{10}$. Nevertheless,

Theorem 27.2 \mathbb{Q} has the same cardinality as \mathbb{N} .

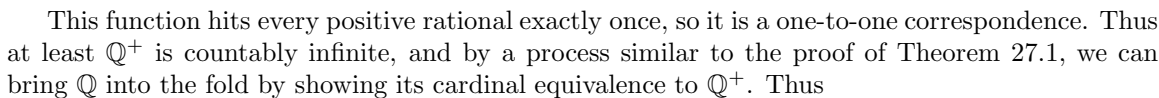
A formal proof is delicate, but this ML program demonstrates the main idea:

```
- fun cantorDiag(n) =
=   let
=     fun gcd(a, 0) = a
=       | gcd(a, b) = gcd(b, a mod b);
=     fun reduce(a, b) =
=       let val comDenom = gcd (a, b);
=       in (a div comDenom, b div comDenom)
```

countably infinite
countable

uncountable

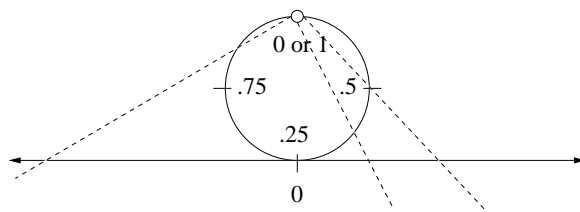
What this function computes is a diagonal walk over the set of positive rationals, invented by Cantor, illustrated here. We lay out all ratios of integers as an infinite 2×2 grid and weave our way around them, assigning a natural number to each in the order that we come to them, but skipping over ones that are equivalent to something we have seen before.



This strongly suggests that all infinities are equal, but the tally is not finished. We still need to consider \mathbb{R} . To simplify things, we will restrict ourselves just to the set $(0, 1)$, that is, only the real numbers greater than 0 and less than 1. This might sound like cheating, but it is not.

179

For this we will use a geometric argument. Imagine taking the line segment $(0, 1)$ and rolling it up into a ball with one point missing where 0 and 1 would meet. Then place the ball on the real number line, so .5 on the ball is tangent with 0 on the line. Now define a function $f : (0, 1) \rightarrow \mathbb{R}$ so that to find $f(x)$ we draw a line from the 0/1 point on the ball through x on the ball; the value $f(x)$ is the point where the drawn line hits the real number line. Proving that this function is a one-to-one correspondence is a matter of using analytic geometry to find a formula for f and then showing that every \mathbb{R} is hit from exactly one value on the ball.



This appears to cut our task down immensely. To prove all infinities (that we know of) equal, all we need to show is that any of the sets already proven countable can be mapped one-to-one and onto the simple line segment $(0, 1)$. However,

Theorem 27.4 $(0, 1)$ is uncountable.

Since Countability calls for an existence proof, uncountability requires a non-existence proof, for which we will need a proof by contradiction.

Proof. Suppose $(0, 1)$ is countable. Then there exists a one-to-one correspondence $f : \mathbb{N} \rightarrow (0, 1)$. We will use f to give names to the all the digits of all the numbers in $(0, 1)$, considering each number in its decimal expansion, where each $a_{i,j}$ stands for a digit.:

$$\begin{aligned} f(1) &= 0.a_{1,1}a_{1,2}a_{1,3} \dots a_{1,n} \dots \\ f(2) &= 0.a_{2,1}a_{2,2}a_{2,3} \dots a_{2,n} \dots \\ &\vdots \\ f(n) &= 0.a_{n,1}a_{n,2}a_{n,3} \dots a_{n,n} \dots \\ &\vdots \end{aligned}$$

Now construct a number $d = 0.d_1d_2d_3 \dots d_n \dots$ as follows

$$d_n = \begin{cases} 1 & \text{if } a_{n,n} \neq 1 \\ 2 & \text{if } a_{n,n} = 1 \end{cases}$$

Since $d \in (0, 1)$ and f is onto, there exists an $n' \in \mathbb{N}$ such that $f(n') = d$. Moreover, $f(n') = 0.a_{n',1}a_{n',2}a_{n',3} \dots a_{n',n'} \dots$, so $d = 0.a_{n',1}a_{n',2}a_{n',3} \dots a_{n',n'} \dots$ by substitution. However, by how we have defined d , $d_n \neq a_{n',n'}$ and so $d \neq 0.a_{n',1}a_{n',2}a_{n',3} \dots a_{n',n'} \dots$, a contradiction.

Therefore $(0, 1)$ is not countable. \square

Corollary 27.1 \mathbb{R} is uncountable.

Proof. Theorems 27.3 and 27.4. \square

Anticlimactic? Perhaps. But also profound. There a just as many naturals as integers as rationals. But there are many, many more reals.

This chapter draws heavily from Epp[5].

Part VII

Program

Chapter 28

Recursion Revisited

First, a word of introduction for this entire part on functional programming. We have already seen many of the building blocks of programming in the functional paradigm, and in the next few chapters we will put them together in various applications and also learn certain advanced techniques. The chapters do not particularly depend on each other and could be resequenced. The order presented here has the following rationale: This chapter will ease you into the sequence with a fair amount of review, and will also look at recursion from its mathematical foundations. Chapter 29 will present the datatype construct as a much more powerful tool than we have seen before, particularly in how the idea of recursion can be extended to types. Chapter 30 is the climax of the part, illustrating the use of functional programming in the fixed-point iteration algorithm strategy; it is also the only chapter of the book that requires a basic familiarity with differential calculus, so if you have not taken the first semester of calculus, ask a friend who has to give you the five-minute explanation of what a derivative is. Chapter 31 is intended as a breather following Chapter 30, applying our skills to computing combinations and permutations.

28.1 Scope

In Chapter 23, we implied that

$$\text{fun } \langle \text{identifier} \rangle_1 (\langle \text{identifier} \rangle_2) = \langle \text{expression} \rangle;$$

is merely shorthand for

$$\text{val } \langle \text{identifier} \rangle_1 = \text{fn } (\langle \text{identifier} \rangle_2) => \langle \text{expression} \rangle;$$

However, this is not true; there is a subtle but important difference, and it boils down to scope. Recall that a variable's scope is the duration of its validity. A variable (including one that holds a function value) is valid from the point of its declaration on. It is not valid, however, in its declaration itself; in the **val**/**fn** form, $\langle \text{identifier} \rangle_1$ cannot appear in $\langle \text{expression} \rangle$. The name of a function defined using the **fun** form, however, has scope including its own definition. Recall that recursion is self-reference; a function defined using **fun** can call itself—or return itself, for that matter.

Functional programming is a style where no variables are modified. We will demonstrate this distinction and how recursive calls make this possible by transforming our iterative factorial function from Chapter 14 into the functional style. We modify this slightly by counting from 1 to n instead of from 0 to $n - 1$, and accordingly we update **fact** before **i**.

```

- fun factorial(n) =
=   let val i = ref 1;
=     val fact = ref 1;
=   in
=     (while !i <= n do
=       (fact := !fact * !i;
=        i := !i + 1);
=     !fact)
=   end;

```

Our first change is to encapsulate the body of the while loop into a function, which we will call **factBody**. The body of the while loop does two things: It updates both **fact** and **i**. Our function, then, must consume the old **fact** and **i** and produce new values for them. We can handle the need to return two values by returning a tuple. Thus we also consolidate **fact** and **i** into one value, the tuple **current**. This essentially represents the current state of the computation.

```

- fun factorial(n) =
=   let fun factBody(fact, i) = (i * fact, i + 1);
=     val current = ref (1, 1);
=   in
=     (while #2(!current) <= n do
=       current := factBody(!current);
=     #1(!current))
=   end;

```

Next, we can be more ambitious about how much of the work we subsume into the function **factBody**. At this point we also take advantage of the recursive use of function names. That the while loop is doing one thing: calling **factBody** repeatedly. Since **factBody** can call itself, it may as well eat up the rest of the while loop. Notice that the while is effectively replaced with an if. Both the old while and the new if are making the same decision—either stop (and do not change the state of **current** or (**fact**, **i**)) or make the change and repeat.

```

- fun factorial(n) =
=   let fun factBody(fact, i) =
=     if i <= n then factBody(i * fact, i + 1)
=     else (fact, i);
=     val current = ref (1, 1);
=   in
=     (current := factBody(!current);
=     #1(!current))
=   end;

```

Now we notice that the second item in **current** is no longer used; **current** can be a single **int**. We have come full circle, in a way—**current** is now equivalent to the old variable (now parameter) **fact**. Accordingly, **factBody** should only return one thing, the new current **fact** value. The main call to **factBody** needs to be given an initial value for the second item (the old variable, now parameter **i**).

```

- fun factorial(n) =
=   let fun factBody(fact, i) =
=     if i <= n then factBody(i * fact, i + 1)
=     else fact;
=     val current = ref 1;
=   in
=     (current := factBody(!current, 1);
=     !current)
=   end;

```

Next, consider the statement list inside the `let` expression. It is rather silly to store the result of the main call to `factBody` and immediately retrieve it. Why not replace the statement list with just the call?

```
- fun factorial(n) =
=   let fun factBody(fact, i) =
=       if i <= n then factBody(i * fact, i + 1)
=       else fact;
=       val current = ref 1;
=   in
=       factBody(!current, 1)
=   end;
```

Now that `current` is never updated, there is no need for it to be a reference variable—or a variable at all, for that matter. We replace its one remaining use with its initial value, 1.

```
- fun factorial(n) =
=   let fun factBody(fact, i) =
=       if i <= n then factBody(i * fact, i + 1)
=       else fact;
=   in
=       factBody(1, 1)
=   end;
```

Next we make use of the associativity of multiplication. Our current version of `factBody` performs its multiplication first and then passes the result to the call, essentially performing $(\dots(((1 \times 1) \times 2) \times 3) \dots \times n)$. This means that `fact` gets bigger on the way down, and the result is unchanged as it comes back up from the series of recursive calls. Instead, we can do the multiplication after the call, so that `fact` stays the same on the way down, but the result gets bigger as it comes up, essentially $(1 \times (1 \times (2 \times (3 \times \dots (n) \dots))))$.

```
- fun factorial(n) =
=   let fun factBody(fact, i) =
=       if i = 0 then fact
=       else factBody(i * fact, i - 1);
=   in
=       factBody(1, n)
=   end;
```

We can also make use of associativity. Instead of performing multiplication first and then passing the result to the call (so `fact` gets bigger on the way down, and the result is unchanged as it comes back up from the series of recursive calls) we can do the multiplication after the call, so that `fact` stays the same on the way down, but the result gets bigger as it comes up, essentially $(n \times ((n - 1) \times ((n - 2) \times (\dots(1) \dots))))$.

```
- fun factorial(n) =
=   let fun factBody(fact, i) =
=       if i = 0 then fact
=       else i * factBody(fact, i - 1);
=   in
=       factBody(1, n)
=   end;
```

An amazing thing has happened: The variable `fact` no longer varies with each call. This means we can eliminate it from the parameter list and replace its use with its only value, 1.

```

- fun factorial(n) =
=   let fun factBody(i) =
=       if i = 0 then 1
=       else i * factBody(i - 1);
=   in
=       factBody(n)
=   end;

```

Now all that `factorial` does is make a local function and apply it to its input without modification. We may as well replace its body with the body of `factBody`—but be careful to substitute `factorial` for `factBody` and `n` for `i`.

```

- fun factorial(n) =
=   if n = 0 then 1
=   else n * factorial(n - 1);

```

Finally, we summon the beautifying magic of pattern-matching.

```

- fun factorial(0) = 1
=   | factorial(n) = n * factorial(n - 1);

```

The mathematical definition of factorial is

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

28.2 Recurrence relations

The medieval mathematician Leonardo of Pisa proposed the following problem. A newborn male/female pair of an immortal species of rabbit is introduced to an island. Individuals of the species reach sexual maturity after one month and have a one-month gestation period. Each litter contains exactly one male and one female. Individuals mate once a month for life with their siblings with a perfect fertility rate and no miscarriages. How many pairs of rabbits will there be after n months?

Ultimately we would like a function $f(n)$ which compute the number of pairs given a number of months. Before defining such a function directly, we consider how one function value may be related to others. The number of pairs of rabbits alive after n months will be

- the number of pairs born in the n th month, plus
- the number of pairs that were alive before the n th month.

The second of these is the same as the number of pairs alive after $n-1$ months, that is, $f(n-1)$. What about the first point? How many new pairs will be born? Since the fertility rate is 100%, every pair will give birth to a new pair—except for juvenile pairs, which are not fertile yet. Since we assume all the pairs older than one month are fertile, this means that every pair alive after $n-2$ months is ready, and so $f(n-2)$ new pairs are born.

$$f(n) = f(n-1) + f(n-2)$$

Yet this does not fully define the function, since $f(n-1)$ and $f(n-2)$ are unknown, unless we define it for a set of base cases.

$$\begin{aligned} f(0) &= 1 && \text{just the original pair} \\ f(1) &= 1 && \text{since the original pair was not sexually mature the previous month} \end{aligned}$$

If you have not guessed, Leonardo of Pisa was better known by his nickname, *Fibonacci*.

```

- fun fibonacci(0) = 1
=   | fibonacci(1) = 1
=   | fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);

```

Recall that a mathematical sequence is a function with domain \mathbb{W} or \mathbb{N} . For a sequence a , we often write a_k for $a(k)$ and refer to this as the k th term in the sequence. A *recurrence relation* for a sequence a is a formula which, for some N relate each term a_k (for all $k \geq N$) to a finite number of predecessor terms $a_{k-N}, a_{k-N+1}, \dots, a_{k-1}$; moreover, terms $a_0, a_1, a_2, \dots, a_{N-1}$ are explicitly defined, which we call the *initial conditions* of the recurrence relation. (The term “relation” is misleading here, since it is not directly related to *relation* as a set of ordered pairs.) This demonstrates for us the basic structure of recursion: A formula has base case by which it is grounded explicitly, and in other cases it is defined in terms of other values of the same formula.

*recurrence relation**initial conditions*

For another example, consider the well-known Tower of Hanoi puzzle. Imagine you have three pegs and k disks. Each disk has a different size and a hole in the middle big enough for the pegs to fit through. Initially all disks are on the first peg from the largest disk on the bottom to the smallest disk on the top. Your task is to move all the disks to the third peg under the following rules:

- You may move only one disk at a time.
- You may never put a larger disk on top of a smaller disk on the same peg.

The strategy we use is itself recursive. Assume we call the pegs 0, 1, and 2, and generalize the problem to moving k disks from peg i to peg j . Then

1. Move the top $k - 1$ disks from peg i to the peg other than i or j .
2. Move the k th disk from peg i to peg j .
3. Move the top $k - 1$ disks from the other peg to peg j .

How many moves will this take? If m_k is the number of moves it takes to solve the puzzle for k disks, then

$$\begin{aligned}
 m_k &= m_{k-1} && \text{to move the top } k-1 \text{ disks} \\
 &+ 1 && \text{to move the } k\text{th disk} \\
 &+ m_{k-1} && \text{to move the top } k-1 \text{ disks again}
 \end{aligned}$$

And

$$m_1 = 1$$

In ML,

```

- fun hanoi(1) = 1
=   | hanoi(n) = 2 * hanoi(n-1) + 1

```

Exercises

1. The definition given for recurrence relations assumes the sequence has domain \mathbb{W} . Rewrite it for sequences that have domain \mathbb{N} .
2. The ML function `fibonacci` is awful. The call `fibonacci(n-1)` itself will call `fibonacci(n-2)`, redoing the work of the other `fibonacci(n-1)`. When you consider how this situation spreads through all the recursive calls, the inefficiency is obscene. Write an iterative version (using reference variables and a while loop) of `fibonacci` that does no duplicate work.
3. Using a process similar to that used to transform `factorial`, transform your `fibonacci` function from Exercise 2 into the functional style (no reference variables) but still without duplicate work.
4. Prove, using math induction, that the sequence m we defined to represent the number of moves in our Towers of Hanoi algorithm is the same as

$$m_k = 2^k - 1$$

Chapter 29

Recursive Types

29.1 Datatype constructors

Suppose we are writing an application whose data includes fractions. The obvious way to represent fractions is a tuple of two integers. Thus, to add two fractions

```
- fun add((a, b), (c, d)) =  
=   let  
=     val numerator = a * d + c * b;  
=     val denominator = b * d;  
=     val divisor = gcd(numerator, denominator);  
=   in  
=     (numerator div divisor, denominator div divisor)  
=   end;  
  
val add = fn : (int * int) * (int * int) -> int * int  
  
- add((1,3), (1,2));  
  
val it = (5,6) : int * int
```

However, this introduces a software engineering danger: What if there are other uses for tuples of two integers in the system, say, points, pairs in a relation, or simply the result of a function that needed to return two values? It would be very easy for a programmer to become careless and introduce bugs that improperly treats an `int` tuple as a fraction or a fraction as some other `int` tuple. In this course we have mostly considered an idealized mathematical world apart from practical software engineering concerns. However, one purpose of types is to make mistakes like this harder to make and easier to detect.

What is missing in the situation above is proper *abstraction*, in this case a way to encapsulate a fraction value and designate it specifically as a fraction so, for example, the point $(5, 3)$ is not mistaken for $\frac{5}{3}$. A better solution is to extend ML's type system to include a rational number type. We know one way to create new types, and that is using the datatype construct. So far we have seen it used to represent only finite sets, not (nearly)¹ infinite sets like the rational type we have in mind. We can expand on the old way of writing datatypes by tacking a little extra data onto the options we give for a datatype.

```
- datatype rational = Fraction of int * int;  
  
datatype rational = Fraction of int * int
```

¹The `int` type, after all, is a finite set, bound by the memory limitations of the computer.

```

- Fraction(3,5);

val it = Fraction (3,5) : rational

- fun add(Fraction(a, b), Fraction(c, d)) =
=   let
=       val numerator = a * d + c * b;
=       val denominator = b * d;
=       val divisor = gcd(numerator, denominator);
=   in
=       Fraction(numerator div divisor, denominator div divisor)
=   end;

val add = fn : rational * rational -> rational

- add(Fraction(1,2), Fraction(1,3));

val it = Fraction (5,6) : rational

```

constructor expression

The construct `Fraction of int * int` is called a *constructor expression*. The idea is that it explains one way to construct a value of the type `rational`. Another way to think of it is that `Fraction` is a husk that contains an `int` tuple as its core. A datatype declaration, then, is a sequence of constructor expressions each following the form

<identifier> of <type expression>

with the “of ...” part optional.

Suppose we were writing an application to manage payroll and other human resources operations for a company. The company has both hourly and salaried employees. We wish to represent both kinds by a single type (so that, for example, values of both can be stored in the same list or array), but different data is associated with them: for hourly employees, their hourly wage and the number of hours clocked since the last pay period; for salaried employees, their yearly salary. We use this datatype:

```

- datatype employee = Hourly of real * int ref | Salaried of real;

```

The reason the second value of `Hourly` is a reference type is so that values can be updated as the employee clocks more hours. Thus

```

- fun clockHours(Hourly(rate, hours), newHours) = hours := !hours + newHours
=   | clockHours(Salaried(salary), newHours) = ();

```

Notice how pattern matching naturally extends to more complicated datatypes. This function does nothing when clocking hours for salaried employees. (If this were a realistic example, it might store those hours for the sake of assessing the employee’s work; we also likely would store information such as the employee’s name and office location.) Similarly, we use pattern matching to determine how to compute an employee’s wage (assume a two-week pay period):

```

- fun computePay(Hourly(rate, hours)) =
=   let val hoursThisPeriod = !hours;
=   in
=       (hours := 0;
=       rate * real(hoursThisPeriod))
=   end
=   | computePay(Salaried(salary)) = salary / 26.0;

```

Notice how the option for hourly employees both resets the hours to 0 and returns the computed wage.

29.2 Peano numbers

Italian mathematician Giuseppe Peano introduced a way of reasoning about whole numbers that includes the following axioms:

Axiom 8 *There exists a whole number 0.*

Axiom 9 *Every whole number n has a successor, `succ n` .*

successor

Axiom 10 *No whole number has 0 as its successor.*

Axiom 11 *If $a, b \in \mathbb{W}$, then $a = b$ iff `succ a` = `succ b` .*

If we interpret *successor* to mean “one more than,” these axioms allow us to define whole numbers recursively (called *Peano numbers*); a whole number is

Peano numbers

- zero, or
- one more than another whole number.

Now we see just how flexible the datatype construct is: The scope of the name of the type being defined includes the definition itself. This means types can be defined recursively.

```
- datatype wholeNumber = Zero | OnePlus of wholeNumber;
```

When we wrote self-referential functions or algorithms, we essentially were thinking of recursive verbs. What is new here is that we are now speaking of recursive nouns. A noun appearing in its own definition should not in itself seem strange; take, for example, this definition of *coral*, adapted from *Merriam-Webster*:

coral 1 : the calcareous or horny skeletal deposit produced by anthozoan polyps.
 2 : a piece of coral

The second definition is merely shorthand for “a piece of the calcareous or horny skeletal deposit produced by anthozoan polyps”—that is, the use of the word *coral* internal to the second definition refers only to the first definition, not back to the second definition itself. No reasonable person would interpret this as a rule that would produce, as a replacement for the occurrence of *coral* in a text, “a piece of a piece of a piece of a piece of the calcareous or horny skeletal deposit produced by anthozoan polyps.”

That is, however, how we interpret our definition of whole numbers. The recursive part establishes a pattern for generating every possible whole number. For example,

5 is a whole number because it is the successor of
 4, which is a whole number because it is the successor of
 3, which is a whole number because it is the successor of
 2, which is a whole number because it is the successor of
 1, which is a whole number because it is the successor of
 0, which is a whole number by axiom.

In ML,

```
- val five = OnePlus(OnePlus(OnePlus(OnePlus(OnePlus(Zero))))) ;

val five = OnePlus (OnePlus (OnePlus (OnePlus (OnePlus Zero))) ) : wholeNumber

- val six = OnePlus(five);

val six = OnePlus (OnePlus (OnePlus (OnePlus (OnePlus (OnePlus Zero)))) ) : wholeNumber
```

Finding the successor of a number is just a matter of tacking “`OnePlus`” to the front, a process easily automated.

```
- fun succ(num) = OnePlus(num);
```

Conversion from an `int` to a `wholeNumber` is a recursive process—the base case, 0, can be returned immediately; for any other case, we add one to the `wholeNumber` representation of the `int` that comes before the one we are converting. (Negative ints will get us into trouble.)

```
- fun asWholeNumber(0) = Zero
  | asWholeNumber(n) = OnePlus(asWholeNumber(n-1));
```

```
val asWholeNumber = fn : int -> wholeNumber
```

predecessor

Notice how subtracting one from the `int` and adding one to the resulting `wholeNumber` balance each other off. Opposite the successor, we define the *predecessor* of a natural number n , `pred n`, to be the number of which n is the successor. From Axiom 11 we can prove that the predecessor of a number, if it exists, is unique; Axiom 10 says that 0 has no predecessor. Pattern matching makes stripping off a “`OnePlus`” easy:

```
- fun pred(OnePlus(num)) = num;
```

```
Warning: match nonexhaustive
       OnePlus num => ...
```

```
val pred = fn : wholeNumber -> wholeNumber
```

You may remember this warning from the first time we saw pattern-matching in Chapter 9—or from more recent mistakes you have made. In this case it is not a mistake; we truly want to leave the operation undefined for `Zero`. Using the function on `Zero`, rather than failing to define it for `Zero`, would be the mistake.

```
- val three = asWholeNumber(3);
```

```
val three = OnePlus (OnePlus (OnePlus Zero)) : wholeNumber
```

```
- pred(three);
```

```
val it = OnePlus (OnePlus Zero) : wholeNumber
```

```
- pred(Zero);
```

```
uncaught exception nonexhaustive match failure
```

Now we can start defining arithmetic recursively. `Zero` will always be our base case; anything we add to `Zero` is just itself. For other numbers, picture an abacus. We have two wires, each with a certain number of beads pushed up. At the end of the computation, we want one of the wires to contain our answer. Thus we push down one bead from the other wire, bring up one bead on the answer wire, and repeat until the other wire has no beads left. In other words, we define addition similarly to our recursive gcd lemmas from Chapter 17.

$$\begin{aligned} 0 + b &= b \\ a + 0 &= a \\ a + b &= (a + 1) + (b - 1) \quad \text{if } b \neq 0 \end{aligned}$$

In ML,

```
- fun plus(Zero, num) = num
=   | plus(num, Zero) = num
=   | plus(num1, OnePlus(num2)) = plus(OnePlus(num1), num2);
```

Examine for yourself the similarity of structure for `isLessThanOrEqualTo`. Keep in mind that recursively-define predicates have two base cases, one true and one false. Here the first and second parameter are in a survival contest; they repeatedly shed a `OnePlus`, and the first one reduced to `Zero` loses.

```
- fun isLessThanOrEqualTo(Zero, num) = true
=   | isLessThanOrEqualTo(num, Zero) = false
=   | isLessThanOrEqualTo(OnePlus(num1), OnePlus(num2)) = lessThanOrEq(num1, num2);
```

For subtraction, we observe

$$\begin{aligned} a - 0 &= a \\ a - b &= (a - 1) - (b - 1) \quad \text{if } a \neq 0 \text{ and } b \neq 0 \end{aligned}$$

In ML,

```
- fun minus(num, Zero) = num
=   | minus(OnePlus(num1), OnePlus(num2)) = minus(num1, num2);
```

```
Warning: match nonexhaustive
      (num, Zero) => ...
      (OnePlus num1, OnePlus num2) => ...
```

```
val minus = fn : wholeNumber * wholeNumber -> wholeNumber
```

This rightly leaves the pattern `minus(Zero, OnePlus(num))` undefined. Finally, conversion back to `int` is just a literal interpretation of the identifiers we gave to the constructor expressions.

```
- fun asInt(Zero) = 0
=   | asInt(OnePlus(num)) = 1 + asInt(num);
```

29.3 Parameterized datatype constructors

A *stack* is any structuring or collection of data for which the following operations are defined:

stack

push Add a new piece of data to the collection.

top Return the most recently-added piece of data that has not yet been removed.

pop Remove the most recently-added piece of data that has not yet been removed.

depth Compute the number of pieces of data in the collection.

The data structure can grow and shrink as you add to and remove from it. You retrieve elements in the reverse order of which you added them. A favorite real-world example is a PEZ dispenser, which will help explain the intuition of the operation name **depth**.

One thing we have left unspecified is what sort of thing (that is, what type) these pieces of data are. This is intentional; we would like to be able to use stacks of any type, just as we have for lists and arrays. This, too, can be implemented using a datatype, because datatypes can be parameterized by type. For example

```
- datatype ('a) someType = Container of 'a;
```

```

datatype 'a someType = Container of 'a

- Container(5);

val it = Container 5 : int someType

- Container(true);

val it = Container true : bool someType

- Container([(Container(3.4), 2),(Container(2.7), 8)]);

val it = Container [(Container 3.4,2),(Container 2.7,8)]
  : (real someType * int) list someType

```

type parameter

Here 'a is a *type parameter*. Indeed, variables may be used to store types, in which case the identifier should begin with an apostrophe. (This should explain some unusual typing judgments ML has been giving you.) The form for declaring a parameterized datatype is

datatype <type parameter> <identifier>

and the form for referring to a parameterized type is

<type expression> <identifier>

type expression

where the identifier is the name of the parameterized type. Notice that this second form is itself a *type expression*, any construct that expresses a type.

With this in hand, we can define a stack recursively as being either

- empty, or
- a single item on top of another stack

and define a generic stack as

```
- datatype ('a) Stack = Empty | NonEmpty of 'a * ('a) Stack;
```

Implementing the operations for the stack come easily by applying the principles from the Peano numbers example to the definitions of the operations.

```

- fun top(NonEmpty(x, rest)) = x;

- fun pop(NonEmpty(x, rest)) = rest;

- fun push(stk, x) = NonEmpty(x, stk);

- fun depth(Empty) = 0
  = | depth(NonEmpty(x, rest)) = 1 + depth(rest);

```

This chapter received much inspiration from Felleisen and Friedman[6].

Exercises

1. Write a function `subtract` for the `rational` type.
2. Write a function `multiply` for the `rational` type.
3. Write a function `divide` for the `rational` type.
4. Write a function `multiply` for the `wholeNumber` type.
5. Write a function `divide`, performing integer division, for the `wholeNumber` type. (Hint: The Division Algorithm from Chapter 17 is a good place to start.)
6. Write a function `modulo`, performing the `mod` operation, for the `wholeNumber` type.
7. Write a function `gcd`, computing the greatest common divisor, for the `wholeNumber` type.

Natural numbers that are powers of 2 can be defined as

- 1, or
- 2 times a power of 2.

We can represent this in ML as

```
- datatype powerOfTwo = One | TwoTimes of powerOfTwo;
```

8. Write a function `multiply`, multiplying two `powerOfTwo`s to get another `powerOfTwo`.
9. Write a function `asPowerOfTwo`, converting an `int` to the nearest `powerOfTwo` less than or equal to it.
10. Write a function `asInteger`, converting a `powerOfTwo` to an equivalent `int`.
11. Write a function `logBase2`, computing an `int` base 2 logarithm from a `powerOfTwo` (that is, the type of this function should be `powerOfTwo -> int`).

If ML did not come with a list construct, we could define our own using the datatype

```
- datatype ('a) homemadeList =  
  =      Null | Cons of 'a * 'a homemadeList;
```

12. Write a function `head` for `'a homemadeList`, equivalent to the ML primitive `hd`.
13. Write a function `tail` for `'a homemadeList`, equivalent to the ML primitive `tl`.
14. Write a function `cat` to concatenate two `'a homemadeList`s. equivalent to the ML primitive `@`.
15. Write a function `isElementOf` for `'a homemadeList`.
16. Write a function `makeNoRepeats` for `'a homemadeList`.
17. Write a function `listify` for `'a homemadeList`.
18. Write a function `sum` for `int homemadeList`.
19. Write a function `map` for `'a homemadeList`, like the `map` function of Chapter 24.

Chapter 30

Fixed-point iteration

30.1 Currying

Before we begin, we introduce a common functional programming technique for reducing the number of arguments or *arity* of a function by partially evaluating it. Take for a simple example a function that takes two arguments and multiplies them.

arity

```
- fun mul(x, y) = x * y;
```

We could specialize this by making a function that specifically doubles any given input by calling `mul` with one argument hardwired to be 2.

```
- fun double(y) = mul(2, y);
```

This can be generalized and automated by a function that takes any first argument and returns a function that requires only the second argument.

```
- fun makeMultiplier(x) = fn y => mul(x, y);
```

```
val makeMultiplier = fn : int -> int -> int
```

```
- makeMultiplier(2);
```

```
val it = fn : int -> int
```

```
- it(3);
```

```
val it = 6 : int
```

This process is called *currying*, after mathematician Haskell Curry, who studied this process (though he did not invent it). We can generalize this with a function that transforms any two-argument function into curried form.

currying

```
- fun curry(func) = fn x => fn y => func(x, y);
```

```
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

```
- curry(mul)(2)(3);
```

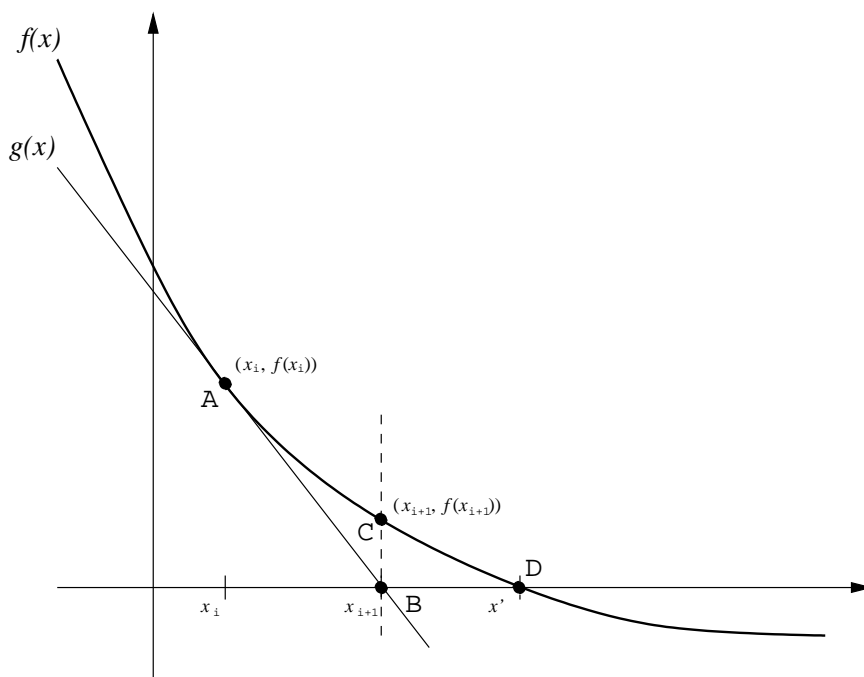
```
val it = 6 : int
```

30.2 Problem

In this chapter we apply our functional programming skills to a non-trivial problem: calculating square roots. One approach is to adapt Newton's method for finding roots in general, which you may recall from calculus. Here is how Newton's method works.

root

Suppose a curve of a function f crosses the x -axis at x' . Functionally, this means $f(x') = 0$, and we say that x' is a *root* of f . Finding x' may be difficult or even impossible; obviously x' may not be rational, and if f is not a polynomial, then x' may not even be an algebraic number (do you remember the difference between \mathbb{A} and \mathbb{T} ?). Instead, we use Newton's method to approximate the root.



tolerance

The approximation is done by making an initial guess and then improving the guess until it is “close enough” (in technical terms, it is within a desired *tolerance* of the correct answer). Suppose x_i is a guess in this process. To improve the guess, we draw a tangent to the curve at the point A, $(x_i, f(x_i))$, and then calculate the point at which the tangent strikes the x -axis. The slope of the tangent can be calculated by evaluating the derivative at that point, $f'(x_i)$. Recall from first-year algebra that if you have a slope m and a point (x', y') , a line through that point with that slope satisfies the equation

$$\begin{aligned} y - y' &= m \cdot (x - x') \\ y &= m \cdot (x - x') + y' \end{aligned}$$

Using point-slope form, we define a function corresponding to the tangent,

$$g(x) = f'(x_i) \cdot (x - x_i) + f(x_i)$$

Let x_{i+1} be the x value where g strikes the x -axis. Thus we want $g(x_{i+1}) = 0$, and solving this equation for x_{i+1} lets us find point B, the intersection of the tangent and the axis.

$$\begin{aligned}
0 &= f'(x_i)(x_{i+1} - x_i) + f(x_i) \\
x_{i+1} &= \frac{x_i f'(x_i) - f(x_i)}{f'(x_i)} \\
&= x_i - \frac{f(x_i)}{f'(x_i)}
\end{aligned} \tag{30.1}$$

Drawing a vertical line through B leads us to C , the next point on the curve where we will draw a tangent. Observe how this process brings us closer to x' , the actual root, at point D . x_{i+1} is thus our next guess. Equation 30.1 tells us how to generate each successive approximation from the previous one. When the absolute value of $f(x_i)$ is small enough (the function value of our guess is within the tolerance of zero), then we declare x_i to be our answer.

We can use this method to find \sqrt{c} by noting that the square root is simply the positive root of the function $f(x) = x^2 - c$. In this case we find the derivative $f'(x) = 2x$ and plug this into Equation 30.1 to produce a function for improving a given guess x :

$$I(x) = x - \frac{x^2 - c}{2x}$$

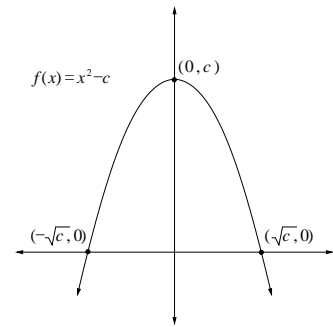
In ML,

```

- fun improve(x) =
=       x - (x * x - c) / (2.0 * x);

val improve = fn : real -> real

```



Obviously this will work only if c has been given a valid definition already. Now that we have our guess-improver in place, our concern is the repetition necessary to achieve a result. Stated algorithmically, while our current approximation is not within the tolerance, simply improve the approximation. By now we should have left iterative solutions behind, so we omit the ML code for this approach. Instead, we set up a recursive solution based on the current guess, which is the data that is being tested and updated. There are two cases, depending on whether the current guess is within the tolerance or not.

- *Base case:* If the current guess is within the tolerance, return it as the answer.
- *Recursive case:* Otherwise, improve the guess and reapply the this test, returning the result as the answer.

In ML,

```

- fun sqrtBody(x) =
=       if inTolerance(x)
=       then x
=       else sqrtBody(improve(x))

val sqrtBody = fn : real -> real

```

We called this function `sqrtBody` instead of `sqrt` because it is a function of the previous guess, x , not a function of the radicand, c . Two things remain: a predicate to determine if a guess is within the tolerance (say, .001; then we are in the tolerance when $|x^2 - c| < .001$), and an initial guess (say, 1). If we package this together, we have

```

- fun sqrt (c) =
=   let fun isInTolerance(x) =
=         abs((x * x) - c) < 0.001;
=       fun improve(x) =
=         x - (x * x - c) / (2.0 * x);
=       fun sqrtBody(x) =
=         if isInTolerance(x)
=           then x
=           else sqrtBody(improve(x))
=   in
=     sqrtBody(1.0)
=   end;

val sqrt = fn : real -> real

- sqrt(2.0);

val it = 1.41421568627 : real

- sqrt(16.0);

val it = 4.00000063669 : real

- sqrt(121.0);

val it = 11.0000000016 : real

- sqrt(121.75);

val it = 11.0340382471 : real

```

30.3 Analysis

Whenever you solve a problem in mathematics or computer science, the next question to ask is whether the solution can be generalized so that it applies to a wider range of problems and thus can be reused more readily. To generalize an idea means to reduce the number of assumptions and to acknowledge more unknowns. In other words, we are replacing constants with variables.

Our square root algorithm was a specialization of Newton's method. The natural next question is how to program Newton's method in general. What assumptions or restrictions did we make on Newton's method when we specialized it? Principally, we assumed that the function for which we were finding a root was in the form $x^2 - c$ where c is a variable to the system. Let us examine how this assumption affects the segment of the solution that tests for tolerance.

```

- fun isInTolerance(x) =
=   abs((x * x) - c) < 0.001;

```

The assumed function shows itself in the expression $(x * x) - c$. By taking the absolute value of that function for a supplied x and comparing with $.001$, we are checking if the function is within an epsilon of zero. We know that functions can be passed as parameters to functions; here, as happens frequently, *generalization* manifests itself as *parameterization*.

```

- fun isInTolerance(function, x) =
=   abs(function(x)) < 0.001;

```

```
val isInTolerance = fn : ('a -> real) * 'a -> bool
```

Take stock of the type. The function `isInTolerance` takes a function (in turn mapping from a type `'a` to `real`) and a value of type `'a`. The given information does not allow ML to infer what type `function` would accept; hence the type variable `'a`. How `function`'s return type is inferred to be `real` is more subtle: `abs` is a special kind of function that is defined so that it can accept either `reals` or `ints`, but it must return the same type that it receives. Since we compare its result against `0.001`, its result must be `real`; thus its parameter must also be `real`, and finally we conclude that `function` must return a `real`.

`isInTolerance` is now less easy to use because we must pass in the function whenever we want to use it, unless `function` is in scope already and we can eliminate it as a parameter. However, we know that functions can also return functions. To make this more general, instead of writing a function to test the tolerance, we write a function that produces a tolerance tester, based on a given function.

```
- fun toleranceTester(function) =
=   fn x => abs(function(x)) < 0.001;

val toleranceTester = fn : ('a -> real) -> 'a -> bool
```

Notice that the `->` operator is right associative, which means it groups items on the right side unless parentheses force it to do otherwise. `toleranceTester` accepts something of type `'a -> real` and returns something of type `'a -> real`. Now we need call `toleranceTester` only once and call the function it returns whenever we want to test for tolerance. To further generalize, let us no longer assume $\epsilon = .001$, but instead parameterize it.

```
- fun toleranceTester(function, tolerance) =
=   fn x => abs(function(x)) < tolerance;

val toleranceTester = fn : ('a -> int) * int -> 'a -> bool
```

What happened to the type? Since `0.001` no longer appears, there is nothing to indicate that we are dealing with `reals`. Yet ML cannot simply introduce a new type variable (for, say, `('a -> 'b) * 'b -> 'a -> bool`) because `abs` is not defined for all types, just `int` and `real`. Instead, ML has to guess, and when it comes between `real` and `int`, it goes with `int`. We will force it to choose `real`.

```
- fun toleranceTester(function, tolerance) =
=   fn x => abs(function(x):real) < tolerance;

val toleranceTester = fn : ('a -> real) * real -> 'a -> bool
```

Next, consider the function `improve`. We can generalize this by stepping back and considering how we formulated it in the first place. It comes from applying Equation 30.1 to a specific function f . Thus we can generalize it by making the function of the curve a parameter. Since we do not have a means of differentiating f , we will need f' to be supplied as well.

```
- fun nextGuess(guess, function, derivative) =
=   guess - (function(guess) / derivative(guess));
```

However, just as with tolerance testing, we would prefer to think of our next-guesser as a function only of the previous guess, not of the curve function and derivative. We can modify `nextGuess` easily so that it produces a function like `improve`:

```
- fun nextGuesser(function, derivative) =
=   fn guess => guess - (function(guess) / derivative(guess));
```

Notice that this process amounts to the partial application of a function, and example of currying. `nextGuess` has three parameters; `nextGuesser` allows us to supply values for some of the parameters, and the result is another function. `sqrtBody` also demonstrates a widely applicable technique. If we generalize our function $I(x)$ based on Equation 30.1 we have

$$G(x) = x - \frac{f(x)}{f'(x)}$$

If x is an actual root, then $f(x) = 0$, and so $G(x) = x$. In other words, a root of $f(x)$ is a solution to the equation

$$x = G(x)$$

fixed point problems

Problems in this form are called *fixed point problems* because they seek a value which does not change when $G(x)$ is applied to it (and so it is fixed). If the fixed point is a local minimum or maximum and one starts with a good initial guess, one approach to solving (or approximating) a fixed point problem is *fixed point iteration*, the repeated application of the function $G(x)$, that is

fixed point iteration

$$G(G(G(\dots G(x') \dots)))$$

where x' is the initial guess, until the result is “good enough.” In parameterizing this process by function, initial guess, and tester, we have this generalized version of `sqrtBody`:

```
- fun fixedPoint(function, guess, tester) =
=   if tester(guess) then guess
=   else fixedPoint(function, function(guess), tester);

val fixedPoint = fn : ('a -> 'a) * 'a * ('a -> bool) -> 'a
```

30.4 Synthesis

We have decomposed our implementation of the square root function to uncover the elements in Newton’s method (and more generally, a fixed point iteration). Now we assemble these to make useful, applied functions. In the analysis, parameters proliferated; as we synthesize the components into something more useful, we will reduce the parameters, or “fill in the blanks.” Simply hooking up `fixedPoint`, `nextGuesser`, and `toleranceTester`, we have

```
- fun newtonsMethod(function, derivative, guess) =
=   fixedPoint(nextGuesser(function, derivative), guess,
=   toleranceTester(function, 0.001));

val newtonsMethod = fn : (real -> real) * (real -> real) * real -> real
```

Given a function, its derivative, and a guess, we can approximate a root. However, one parameter in particular impedes our use of `newtonsMethod`. We are required to supply the `derivative` of `function`; in fact, many curves on which we wish to use Newton’s method may not be readily differentiable. In those cases, we would be better off finding a numerical approximation to the derivative. The easiest such approximation is the *secant method*, where we take a point on the curve near the point at which we want to evaluate the derivative and calculate the slope of the line between those points (which is a secant to the curve). Thus for small ϵ , $f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$. In ML, taking $\epsilon = .001$,

```
- fun numDerivative(function) =
=   fn x => (function(x + 0.001) - function(x)) / 0.001;

val numDerivative = fn : (real -> real) -> real -> real
```

Now we make an improved version of our earlier function. Since any user of this new function is concerned only about the results and not about how the results are obtained, our name for it shall reflect *what the function does* rather than *how it does it*.

```
- fun findRoot(function, guess) =
=   newtonsMethod(function, numDerivative(function), guess);

val findRoot = fn : (real -> real) * real -> real
```

Coming full circle, we can apply these pre-packaged functions to a special case: finding the square root. We can use `newtonsMethod` directly and provide an explicit derivative or we can use `findRoot` and rely on a numerical derivative, with different levels of precision. Since $x^2 - c$ is monotonically increasing, 1 is a safe guess, which we provide.

```
- fun sqrt(c) =
=   newtonsMethod(fn x => x * x - c, fn x => 2.0 * x, 1.0);

val sqrt = fn : real -> real

- sqrt(2.0);

val it = 1.41421568627 : real

- sqrt(16.0);

val it = 4.00000063669 : real

- sqrt(121.0);

val it = 11.0000000016 : real

- sqrt(121.75);

val it = 11.0340382471 : real
```

There are several lessons here. First, this has been a demonstration of the interaction between mathematics and computer science. The example we used comes from an area of study called numerical analysis which straddles the two fields. Numerical analysis is concerned with the numerical approximation of calculus and other topics of mathematical analysis. More importantly, this also demonstrates the interaction between discrete and continuous mathematics. We have throughout assumed that $f(x)$ is a real-valued function like you are accustomed to seeing in calculus or analysis. However, *the functions themselves* are discrete objects. The most important lesson is how functions can be parameterized to become more general, curried to reduce parameterization, and, as discrete objects, passed and returned as values.

The running example in this chapter was developed from Abelson and Sussman [1].

Exercises

1. Write a function similar to the original `sqrt` of Section 30.2 but to calculate cubed roots.
2. Notice that a root of f is either a local minimum or a local maximum of G . Identify under what circumstances it is a minimum and under what circumstances it is a maximum.
3. Write a function similar to `sqrt` of Section 30.4 but to calculate cubed roots.
4. The implementation of `fixedPoint` presented in this chapter differs from traditional fixed point iteration because it requires the user to provide a function which determines when the iteration should stop, based on the current guess. That approach is tied to the application of finding roots, since our criterion for termination how close $f(\text{guess})$ is to zero. Instead, termination should depend on how close *successive guesses are to each other*, that is, if $|x_i - x_{i-1}| < \epsilon$. Rewrite `fixedPoint` so that it uses this criterion and receives a tolerance instead of a tester function.

Chapter 31

Combinatorics

31.1 Counting

In Chapter 25, we proved that if A and B are finite, disjoint sets, then $|A \cup B| = |A| + |B|$. We will generalize this idea, but first

Lemma 31.1 *If A and B are finite sets and $B \subseteq A$, then $|A - B| = |A| - |B|$.*

Proof. By Exercise 4 of Chapter 12, $(A - B) \cup B = A$. By Exercise 22 of Chapter 11, $A - B$ and B are disjoint. Thus, by Theorem 25.2, $|A - B| + |B| = |A|$. By algebra, $|A - B| = |A| - |B|$. \square .

Now,

Theorem 31.1 *If A and B are finite sets, then $|A \cup B| = |A| + |B| - |A \cap B|$.*

Proof. Suppose A and B are finite sets. Note that by Exercise 23 of Chapter 11, A and $B - (A \cap B)$ are disjoint; and that by Exercise 10.1 also of Chapter 11, $A \cap B \subseteq B$. Then,

$$\begin{aligned} |A \cup B| &= |A \cup (B - (A \cap B))| && \text{by Exercise 15 of Chapter 11 and substitution} \\ &= |A| + |B - (A \cap B)| && \text{by Theorem 25.2} \\ &= |A| + |B| - |A \cap B| && \text{by Lemma 31.1. } \square \end{aligned}$$

This result can be interpreted as directions for counting the elements of A and B . It is not simply a matter of adding the number of elements in A to the number of elements in B , because A and B might overlap. For example, if you counted the number of math majors and the number of computer science majors in this course, you may end up with a number larger than the enrollment because you counted the double math-computer science majors twice. To avoid counting the overlapping elements twice, we subtract the cardinality of the intersection.

The area of mathematics that studies counting sets and the ways they combine and are ordered is *combinatorics*. (Elementary combinatorics is often simply called “counting,” but that invites derision from those unacquainted with higher mathematics.) It plays a part in many field of mathematics, probability and statistics especially. Lemma 31.1 is called the *difference rule* and Theorem 31.1 is called the *inclusion/exclusion rule*. We can generalize Theorem 25.2 to define the *addition rule*:

combinatorics

difference rule

inclusion/exclusion rule

Theorem 31.2 *If A is a finite set with partition A_1, A_2, \dots, A_n , then $|A| = \sum_{i=1}^n |A_i|$.*

addition rule

and introduce the *multiplication rule*:

multiplication rule

Theorem 31.3 *If A_1, A_2, \dots, A_n are finite sets, then $|A_1 \times A_2 \times \dots \times A_n| = |A_1| \cdot |A_2| \cdot \dots \cdot |A_n|$.*

The proofs are left as exercises.

Consider some examples of these. ML allows identifiers to be made up of upper and lower case letters, numbers, underscores, and apostrophes, with the first character being a letter or apostrophe. Furthermore, an identifier may not be the same as a reserved word. How many possible four-character identifiers are there in ML? To solve this, consider the various sets of characters. Let A be the set of capital letters, B the set of lowercase letters, C the set $\{ '\}$, D the set $\{ _\}$, and E the set of digits. Thus

$$\begin{array}{ll}
 & \overbrace{(|A| + |B| + |C|)}^{\text{addition rule}} \\
 \text{multiplication rule} & \left\{ \begin{array}{l} \times (|A| + |B| + |C| + |D| + |E|) \\ \times (|A| + |B| + |C| + |D| + |E|) \\ \times (|A| + |B| + |C| + |D| + |E|) \end{array} \right. \\
 \text{difference rule} & - |\{ \text{case, else, open, then, type, with} \}|
 \end{array}$$

The number of identifiers is $(53 \cdot 64 \cdot 64 \cdot 64) - 6 = 13893626$.

Next consider the number of possible phone numbers in an area code. Phone numbers are seven characters, chosen from the ten digits with the restrictions that no numbers begin with 0, 1, 555, 411, or 911. Notice that there are

$$10 \cdot 10 \cdot 10 \cdot 10 = 10000$$

four-digit suffixes, and

$$8 \cdot 10 \cdot 10 - 3 = 797$$

three-digit prefixes that do not start with 0 or 1 and do not include the restricted prefixes. Choosing a phone number means choosing a prefix and a suffix, hence $797 \cdot 10000 = 7970000$ possible phone numbers.

31.2 Permutations and combinations

permutation

A classic set of combinatorial examples comes from a scenario where we have an urn full of balls and we are pulling balls out of the urn. Suppose first that we are picking each ball one at a time; thus we are essentially giving an ordering the set of balls. A *permutation* of a set is a sequence of its elements. A permutation is like a tuple of the set with dimension the same as the cardinality of the entire set, except one important difference—once we choose the first element, that element is not in the set to choose from for the remaining elements, and so on. In ML, our lists are a permutation of a set. Applying the multiplication rule, we see that the number of permutations of a set of size n is

$$\begin{aligned}
 P(n) &= n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1 \\
 &= n!
 \end{aligned}$$

r-permutation

An *r-permutation* of a set is a permutation of a subset of size r —in other words, we do not pull out all the balls, only the first r we come to. The number of r -permutations of a set of size n is

$$\begin{aligned}
 P(n, r) &= n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot (n-r+1) \\
 &= \frac{n!}{(n-r)!}
 \end{aligned}$$

On the other hand, what if we grabbed several balls from the urn at the same time? If you have four balls in your hand at once, it is not clear what order they are in; instead, you have simply

chosen an unordered subset. An *r-combination* of a set of n elements is a subset of size r . How many subsets of size r are there of a set of size n (written $\binom{n}{r}$)? First, we know that the number of *orderings* of subsets of that size is $\frac{n!}{(n-r)!}$. Each of those subsets can be ordered in $r!$ ways. Hence

r-combination

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

31.3 Computing combinations

Our main interest here is computing not just the number of permutations and combinations, but the permutations and combinations themselves. Suppose we want to write a function `combos(x, r)` which takes a set `x` and returns a set of sets, all the subsets of `x` of size r . We need to find a recursive strategy for this. First off, we can consider easy cases. If `x` is an empty set, there are no combinations of any size. Similarly, there are no combinations of size zero of any set. Thus

```
- fun combos([], r) = []
=   | combos(x, 0) = []
```

The case where r is 1 is also straight forward: Every element in the set is, by itself, a combination of size 1. Creating a set of all those little sets is accomplished by our `listify` function from Chapter 18.

```
- fun combos([], r) = []
=   | combos(x, 0) = []
=   | combos(x, 1) = listify(x)
```

The strategy taking shape is odd compared to most of the recursive strategies we have seen before. The variety of base cases seems to anticipate the problem being made smaller in both the `x` argument and the `r` argument. What does this suggest? When you form a combination of size r from a list, you first must decide whether that combination will contain the first element of the list or not. Thus all the combinations are

- all the combinations of size r that do not contain the first element, plus
- all the combinations of size $r - 1$ that do not contain the first element with the first element added to all of them.

Function `addToAll` from Exercise 8 of Chapter 18 will work nicely here.

```
- fun combos([], r) = []
=   | combos(x, 0) = []
=   | combos(x, 1) = listify(x)
=   | combos(head::rest, r) =
=       addToAll(head, combos(rest, r-1)) @ combos(rest, r);
```

Exercises

1. Prove Theorem 31.2.
2. Prove Theorem 31.3.
3. Write a function `P(n, r)` which computes the number of r -permutations of a set of size n .
4. Write a function `C(n, r)` which computes the number of r -combinations of a set of size n .

The following exercises walk you through how to write a function to compute r -permutations.

5. Write a function `addEverywhere(x, y)` which takes an item x and a list of items of that type y and returns a list of

lists, each like y but with x inserted into every position. For example, `addEverywhere(1, [2, 3, 4])` should return `[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1]]`. Given an empty list, it should return the list containing just the list containing x .

6. Write a function `fullPermus(y)` which takes a list and returns the list of all the permutations of that list.
7. Write a function `fullPermusList(y)` which takes a list of lists and returns a united list of all the permutations of all the lists in y .
8. Write a function `permus(y, r)` which computes all the r -permutations of the list y .

Chapter 32

Special Topic: Computability

In Chapter 27, we noted the amazing property that natural numbers, integers, and rationals all have the same cardinality, being countably infinite, but that real numbers are more infinitely numerous. That discussion only considered comparing sizes of number sets. What about infinite sets of other things?

Let us take for example computer programs. The finite nature of any given computer necessitates that the set of computer programs is not even infinite. However, suppose we even allow for a computer with an arbitrary amount of memory, where more always could be added if need be. Since computer programs are stored in memory, and memory is a series of bits, we can interpret the bit-representation of a program as one large natural number in binary. Thus we have a function from computer programs to natural numbers. This function is not necessarily a one-to-one correspondence (it certainly is not, if we exclude from the domain any bit representations of invalid programs), but it is one-to-one, since every natural number has only one binary representation, which means that there are at least as many natural numbers as programs, perhaps more. The set of computer programs is therefore countable.

In ML programming, we think of a program as something that represents and computes a function. How many functions are there? Since every real number can be considered to be a constant function, it is easily argued that there are uncountably many possible functions. Nevertheless, let us look at a narrower scope of functions, ones we have a chance of writing a program to compute (we could not expect, for example, a program to compute an arbitrary real number in a finite amount of time). For convenience, we choose the seemingly arbitrary set T of functions from natural numbers to digits,

$$T = \{f : \mathbb{N} \rightarrow \{0, 1, 2, \dots, 9\}\}$$

Now we will define a function not in the set T but with T as its codomain (a function that returns functions), $h : (0, 1) \rightarrow T$. Suppose we represent a number in $(0, 1)$ as $0.a_1a_2a_3\dots a_n\dots$. Then define

$$h(0.a_1a_2a_3\dots a_n\dots) = \text{the function } f \text{ defined by } f(n) = a_n$$

Every number in $(0, 1)$ will produce a unique function, and every function in T defines a unique real number if one simply feeds the natural numbers into it and interprets the output as a decimal expansion. Hence h is one-to-one and onto, and so T is uncountable.

If T is uncountable and the set of possible programs is countable, this means there are more functions than there are computer programs—and further, that there must be some functions for which there are no possible computer programs. Some functions cannot be computed. This is a very humbling thought about the limitations of the machines that we build.

The set of functions that map natural numbers to digits is still fairly unrealistic. Are any of these uncomputable functions something that we would actually want to compute? Suppose we wanted a program/function that would take as its input another program/function and input for

that program/function and return true if the given program halts (that is, does not loop forever or have infinite recursion) on that input, false otherwise. To clarify, the following program does half of the intended work:

```
- fun halfHalt(f, x) = (f(x); true);

val halfHalt = fn : ('a -> 'b) * 'a -> bool
```

This program will indeed return true if the given program halts on the given input, but rather than return true otherwise, it will go on forever itself. We cannot just let the program run for a while and, if it does not end on its own, break the execution and conclude it loops because we will never know whether or not we have waited long enough. A program that does decide this would be useful indeed. One of the most frequent kinds of programming mistakes (especially for beginners) is untermiated iteration or recursion. Such a program—at least a program that would differentiate all programs precisely and not have any program for which it could not tell either way—is impossible.

To see that this is true, suppose we had such a program, `halt`.

```
- fun d(m) = if halt(m, m)
=           then (while true do ()); false)
=           else true;

val d = fn : ('a -> 'b) -> bool1
```

Given a function `m`, this program feeds `m` into `halt` as both the function to run and the input to run it on. If `m` does not halt on itself, this function returns true. If it does halt, then this program loops forever (the `false` is present in the statement list just so that it will type correctly; it will never be reached because the while loop will not end).

What would be the result of running `d(d)`? If `d` halts when it is applied to itself, then `halt` will return true, so then `d` will loop forever. If `d` loops forever when it is applied to itself, then `halt` will return false, so then `d` halts, returning true. If it will halt, then it will not; if it will not, then it will. This contradiction means that the function `halt` cannot exist.

halting problem

This, the unsolvability of the *halting problem* is a fundamental result in the theory of computation since it defines a boundary of what can and cannot be computed. We can write a program that *accepts* this problem, that is, that will answer true if the program halts but does not answer at all if it does not. What we cannot write is a program that *decides* this problem. Research has shown that all problems that can be accepted but not decided are reducible to the halting problem. If we had a model of computation—something much different from anything we have imagined so far—which could decide the halting problem, then all problems we can currently accept would then also be decidable.

¹This actually would not type in ML because the application `halt(m, m)` requires the equation `'a = ('a -> 'b)` to hold. ML cannot handle recursive types unless datatypes are used.

Chapter 33

Special topic: Comparison with object-oriented programming

This chapter is for students who have taken a programming course using an object-oriented language such as Java. If you plan to take such a course in the future, you are recommended to come back and read this chapter after you have learned the fundamentals of class design, subtyping, and polymorphism.

You have no doubt noticed the striking difference in flavor between functional programming and object-oriented programming. A functional language views a program as a set of interacting functions, whereas an object-oriented program is a set of interacting objects. A quick sampling of their similarities will illuminate these differences. Here are two principles which cut across all styles of programming which are relevant for our purposes:

- A program or system is comprised of data structures and functionality.
- A well-designed language encourages writing code that is modular (it is made up of small, semi-autonomous parts), reusable (those parts can be plugged into other systems), and extensible (the system can be modified by adding new parts).

Notice how functional and object-oriented styles address the first point, at least in the way they are taught to beginners. In an object-oriented language, data structures and the functionality defined on them are packaged together; a class represents the organization of data (the instance variables) and operations (the instance methods) in one unit. In a functional language, the data (defined, for example, by a datatype) is less tightly coupled to the functionality (the functions written for that datatype). This touches on the second point as well: In a functional language, the primary unit of modularity is the function, and in an object-oriented language, the primary unit of modularity is the class.

To see this illustrated, consider this system in ML to model animals and the noises they make.

```
- datatype Animal = Dog | Cat ;

- fun happyNoise(Dog) = "pant pant"
=   | happyNoise(Cat) = "purrrr"

- fun excitedNoise(Dog) = "bark"
=   | excitedNoise(Cat) = "meow"
```

It is easy to extend the functionality (that is, add an operation). We need only write a new function, without any change to the datatype or other functions.

```
- fun angryNoise(Dog) = "grrrrr"
=   | angryNoise(Cat) = "hisssss"
```

It is difficult, on the other hand, to extend the data. Adding a new kind of animal requires changing the datatype and every function that operates on it. From the ML interpreter's perspective, this is rewriting the whole system from scratch.

```
- datatype Animal = Dog | Cat | Chicken;

- fun happyNoise(Dog) = "pant pant"
=      | happyNoise(Cat) = "purrrrr"
=      | happyNoise(Chicken) = "cluck cluck";

- fun excitedNoise(Dog) = "bark"
=      | excitedNoise(Cat) = "meow"
=      | excitedNoise(Chicken) = "cockadoodledoo";

- fun angryNoise(Dog) = "grrrrrr"
=      | angryNoise(Cat) = "hisssss"
=      | angryNoise(Chicken) = "squaaaack";
```

In an object-oriented setting, we have the opposite situation. A Java system equivalent to our original ML example would be

```
interface Animal {
    String happyNoise();
    String excitedNoise();
}

class Dog implements Animal {
    String happyNoise() { return "pant pant"; }
    String excitedNoise() { return "bark"; }
}

class Cat implements Animal {
    String happyNoise() { return "purrrrr"; }
    String excitedNoise() { return "meow"; }
}
```

Although the interface demands that everything of type `Animal` will have methods `happyNoise` and `excitedNoise` defined for it, the code for an operation like `happyNoise` is distributed among the classes. The result is a system where it is very easy to extend the data; you simply write a new class, without changing the other classes or the interface.

```
class Chicken implements Animal {
    String happyNoise() { return "cluck cluck"; }
    String excitedNoise() { return "cockadoodledoo"; }
}
```

The price is that we have made extending the functionality difficult. Adding a new operation now requires a change to the interface and to every class.

```
interface Animal {
    String happyNoise();
    String excitedNoise();
    String angryNoise();
}
```



```
class Dog implements Animal {
    String happyNoise() { return "pant pant"; }
    String excitedNoise() { return "bark"; }
    String angryNoise() { return "grrrrr"; }
}

class Cat implements Animal {
    String happyNoise() { return "purrrrr"; }
    String excitedNoise() { return "meow"; }
    String angryNoise() { return "hissss"; }
}

class Chicken implements Animal {
    String happyNoise() { return "cluck cluck"; }
    String excitedNoise() { return "cockadoodledoo"; }
    String angryNoise() { return "squaaaack"; }
}
```

We can lay out the types and the operations in a table to represent the system in a way independent of either paradigm.

	Dog	Cat	Chicken
happyNoise	pant pant	purrrr	cluck cluck
excitedNoise	bark	meow	cockadoodledoo
angryNoise	grrrrr	hissss	squaaaack

Relative to this table, functional programming packages things by rows, and adding a row to the table is convenient. Adding a column is easy in object-oriented programming, since a column is encapsulated by a class.

It is worth noting that object-oriented programming's most touted feature, inheritance, does not touch this problem. Adding a new operation like `angryNoise` by subclassing may allow us to leave the old classes untouched, but it does require writing three new classes and a new interface.

```
interface AnimalWithAngryNoise extends Animal {
    String angryNoise();
}

class DogWithAngryNoise extends Dog implements AnimalWithAngryNoise {
    String angryNoise() { return "grrrrr"; }
}

class CatWithAngryNoise extends Cat implements AnimalWithAngryNoise {
    String angryNoise() { return "hissss"; }
}

class ChickenWithAngryNoise extends Chicken implements AnimalWithAngryNoise {
    String angryNoise() { return "squaaaack"; }
}
```

Worse yet, all the code that depends on the old classes and interfaces must be changed to refer to the new (and oddly named) types.

This exemplifies a fundamental trade-off in designing a system. The Visitor pattern[8] is a way of importing the advantages (and liabilities) of the functional paradigm to object-oriented languages. A system that is easily extended by both data and functionality is a perennial riddle in software design.

Part VIII

Graph

Chapter 34

Graphs

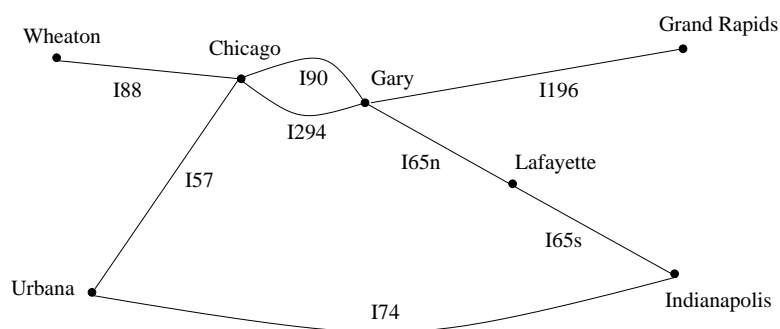
34.1 Introduction

We commonly use the word *graph* to refer to a wide range of graphics and charts which provide an illustration or visual representation of information, particularly quantitative information. In the realm of mathematics, you probably most closely associate graphs with illustrations of functions in the real-number plane. *Graph theory*, our topic in this part, is a field of mathematics that studies a very specific yet abstract notion of a graph. It has many applications throughout mathematics, computer science, and other fields, particularly for modeling systems and representing knowledge.

Unfortunately, the beginning student of graph theory will likely feel intimidated by the horde of terminology required; the slight differences of terms among sources and textbooks aggravates the situation. The student is encouraged to take careful stock of the definitions in these chapters, but also to enjoy the beauty of graphs and their uses. This chapter will consider the basic vocabulary of graph theory and a few results and applications. The following chapter will explore various kinds of paths through graphs. Finally, we will use graph theory as a framework for discussing isomorphism, a central concept throughout mathematics.

A *graph* $G = (V, E)$ is a pair of finite sets, a set V of *vertices* (singular *vertex*) and a set E of pairs of vertices called *edges*. We will typically write $V = \{v_1, v_2, \dots, v_n\}$ and $E = \{e_1, e_2, \dots, e_m\}$ where each $e_k = (v_i, v_j)$ for some v_i, v_j ; in that case, v_i and v_j are called *end points* of the edge e_k . Graphs are drawn so that vertices are dots and edges are line segments or curves connecting two dots.

As an example of a mathematical graph and its relation to everyday visual displays, consider the graph where the set of vertices is { Chicago, Gary, Grand Rapids, Indianapolis, Lafayette, Urbana, Wheaton } and the edges are direct highway connections between these cities. We have the following graph (with vertices and edges labeled). Notice how this resembles a map, simply more abstract (for example, it contains no accurate information about distance or direction).

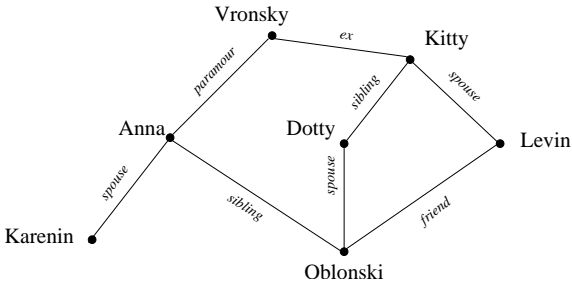


graph
vertex
edge
end points

We call the edges *pairs* of vertices for lack of a better term; a pair is generally considered a

two-tuple (in this case, it would be an element of $V \times V$); moreover, we write edges with parentheses and a comma, just as we would with tuples. However, we mean something slightly different. First, tuples are ordered. In our basic definition of graphs, we assume that the end points of an edge are unordered: we could write I57 as (Chicago, Urbana) or (Urbana, Chicago). Second, an edge as a pair of vertices is not unique. In the cities example, we have duplicate entries for (Chicago, Gary): both I90 and I294. This is why it is necessary to have two ways to represent an edge, a unique name as well as a descriptive one.

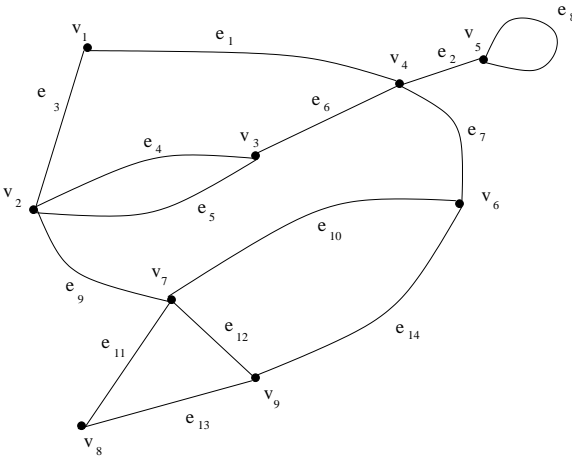
The kinship between graphs and relations should be readily apparent. Graphs, however, are more flexible. As a second example, this graph represents the relationships (close friendship, siblinghood, or romantic involvement—labeled on the drawing but not as names for the edges) among the main characters of *Anna Karenina*. $V = \{ \text{Karenin, Anna, Vronsky, Oblonsky, Dolly, Kitty, Levin} \}$. $E = \{ (\text{Karenin, Anna}), (\text{Anna, Vronsky}), (\text{Vronsky, Kitty}), (\text{Anna, Oblonsky}), (\text{Oblonsky, Dolly}), (\text{Dolly, Kitty}), (\text{Oblonsky, Levin}), (\text{Kitty, Levin}) \}$.



34.2 Definitions

incident
connects
adjacent
self-loop
parallel

An edge (v_i, v_j) is *incident* on its end points v_i and v_j ; we also say that it *connects* them. If vertices v_i and v_j are connected by an edge, they are *adjacent* to one another. If a vertex is adjacent to itself, that connecting edge is called a *self-loop*. If two edges connect the same two vertices, then those edges are *parallel* to each other. Below, e_1 is incident on v_1 and v_4 . e_{10} connects v_7 and v_6 . v_9 and v_6 are adjacent. e_8 is a self-loop. e_4 and e_5 are parallel.



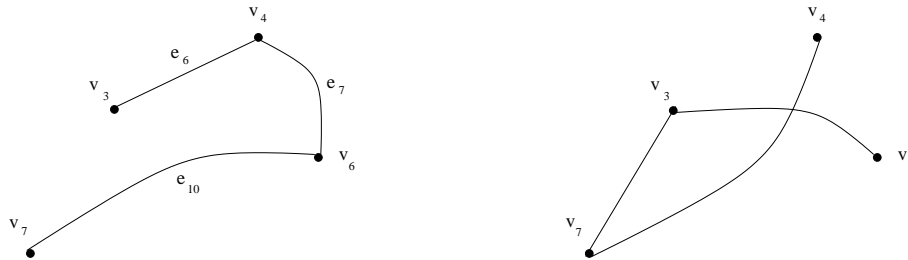
degree
subgraph
simple

The *degree* $\deg(v)$ of a vertex v is the number of edges incident on the vertex, with self-loops counted twice. $\deg(v_1) = 2$, $\deg(v_5) = 3$, and $\deg(v_2) = 4$. A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ (and, by definition of graph, for any edge $(v_i, v_j) \in E'$, $v_i, v_j \in V'$). A graph $G = (V, E)$ is *simple* if it contains no parallel edges or self-loops. The graph $(\{v_1, v_2, v_3, v_4, v_5\}, \{e_1, e_2, e_3, e_4, e_6\})$ is a simple subgraph of the graph shown.

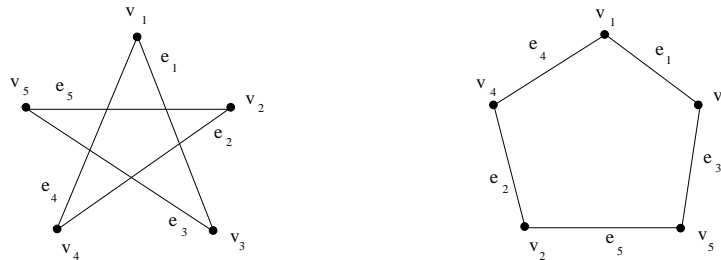
A simple graph $G = (V, E)$ is *complete* if for all $v_i, v_j \in V$, the edge $(v_i, v_j) \in E$. The subgraph $(\{v_7, v_8, v_9\}, \{e_{11}, v_{12}, v_{13}\})$ is complete. The *complement* of a simple graph $G = (V, E)$ is a graph $\bar{G} = (V, E')$ where for $v_i, v_j \in V$, $(v_i, v_j) \in E'$ if $(v_i, v_j) \notin E$; in other words, the complement has all the same vertices and all (and only) those possible edges that are not in the original graph. The complement of the subgraph $(\{v_3, v_4, v_6, v_7\}, \{e_6, e_7, e_{10}\})$ is $(\{v_3, v_4, v_6, v_7\}, \{(v_3, v_7), (v_7, v_4), (v_3, v_6)\})$, as shown below.

complete

complement



Even though we tend to think of a graph fundamentally as a picture, it is important to notice that a formal description of a graph is independent of the way it is drawn. Important to the essence of a graph is merely the names of vertices and edges and their connections (and in Chapter 36, we will see that even the names are not that important). The following two pictures show the same graph.



A *directed graph* is a graph where the edges are ordered pairs, that is, edges have directions. Pictorially, the direction of the edge is shown with arrows. Notice that a directed graph with no parallel edges is the same as a set together with a relation on that set. For this reason, we were able to use directed graphs to visualize relations in Part V. In a directed graph, we must differentiate between a vertex's *in-degree*, the number of edges towards it, and its *out-degree*, the number of edges away from it.

directed graph

in-degree

out-degree

34.3 Proofs

By now you should have achieved a skill level for writing proofs at which it is appropriate to ease up on the formality slightly. The definitions in graph theory do not avail themselves to proofs as detailed as those we have written for sets, relations, and functions, and graph theory proofs tend to be longer anyway. Do not be misled, nevertheless, into thinking that this is lowering the standards for logic and rigor; we will merely be stepping over a few obvious details for the sake of notation, length, and readability. The proof of the following proposition shows what sort of argumentation is expected for these chapters.

Theorem 34.1 (Handshake.) *If $G = (V, E)$ is a graph with $V = \{v_1, v_2, \dots, v_n\}$, then $\sum_{i=1}^n \deg(v_i) = 2 \cdot |E|$.*

Proof. By induction on the cardinality of E . First, suppose that G has no edges, that

is, $|E| = 0$. Then for any vertex $v \in V$, $\deg(v) = 0$. Hence $\sum_{i=1}^n \deg(v_i) = 0 = 2 \cdot 0 = 2 \cdot |E|$.

Hence there exists an $N \geq 0$ such that for all $m \leq N$, if $|E| = m$ then $\sum_{i=1}^n \deg(v_i) = 2 \cdot |E|$.

Now suppose $|E| = N + 1$, and suppose $e \in E$. Consider the subgraph of G , $G' = (V, E - \{e\})$. We will write the degree of $v \in V$ when it is being considered a vertex in G' instead of G as $\deg'(v)$. $|E - \{e\}| = N$, so by our inductive hypothesis $\sum_{i=1}^n \deg'(v_i) = 2 \cdot |E - \{e\}|$.

Suppose v_i, v_j are the end points of e . If $v_i = v_j$, then $\deg(v_i) = \deg'(v_i) + 2$; otherwise, $\deg(v_i) = \deg'(v_i) + 1$ and $\deg(v_j) = \deg'(v_j) + 1$; both by the definition of degree. For any other vertex $v \in V$, where $v \neq v_i$ and $v \neq v_j$, we have $\deg(v) = \deg'(v)$.

Hence $\sum_{i=1}^n \deg(v_i) = 2 + \sum_{i=1}^n \deg'(v_i) = 2 + 2 \cdot |E - \{e\}| = 2 + 2(|E| - 1) = 2 \cdot |E|$. \square

Here are the things left out of this proof.

- The third sentence makes the unjustified claim that having no edges implies every vertex has a degree of zero. This follows immediately from the definition of degree, and it is the best we can do without a formal notion of what “the number of edges” means. Keep in mind that the only formal mechanism we have developed for reasoning about quantity is cardinality.
- In the fourth sentence, substitution and rules of arithmetic are used without citation.
- The claim $|E - \{e\}| = N$ depends on Lemma 31.1 and the facts that E and $\{e\}$ are disjoint that $|\{e\}| = 1$.
- The sixth sentence of the second paragraph together with the last sentence claims that whether or not e is a self loop, it contributes two to the total sum of degrees. It is difficult to state this more formally.
- The last sentence uses arithmetic, algebra, and substitution uncited.

34.4 Game theory

Finally, we consider an application of graph theory. Graphs can be used to enumerate the possible outcomes in the playing of a game or manipulation of a puzzle, a method used in game theory, decision theory, and artificial intelligence. Consider the following puzzle:

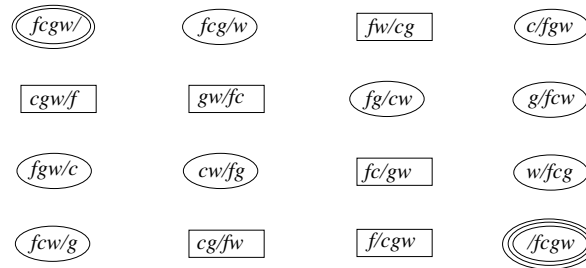
You must transport a cabbage, a goat, and a wolf across a river using a boat. The boat has only enough room for you and one of the other objects. You cannot leave the goat and the cabbage together unsupervised, or the goat will eat the cabbage. Similarly, the wolf will eat the goat if you are not there to prevent it. How can you safely transport all of them to the other side?

We will solve this puzzle by analyzing the possible “states” of the situation, that is, the possible places you, the goat, the wolf, and the cabbage can be, relative to the river; and the “moves” that can be made between the states, that is, your rowing the boat across the river, possibly with one of the objects. Let f stand for you, g for the goat, w for the wolf, and c for the cabbage. $/$ will show how the river separates all of these. For example, the initial state is $fgwc/$, indicating that you and all the objects are on one side of the river. If you were to row across the river by yourself, this would move the puzzle into the state gwc/f , which would be a failure. Our goal is to find a series of moves that will result in the state $/fgwc$.

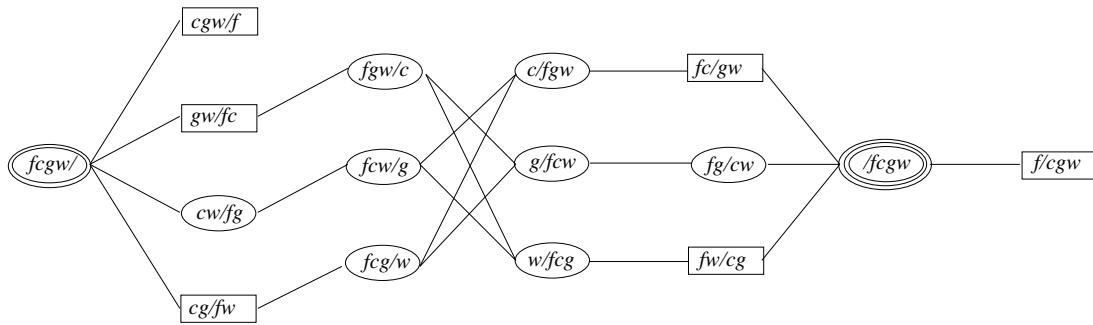
First, enumerate all the states.

$fcgw/$	fcg/w	fw/cg	c/fgw
cgw/f	gw/fc	fg/cw	g/fcw
fgw/c	cw/fg	fc/gw	w/fcg
fcw/g	cg/fw	f/cgw	$/fcgw$

Now, mark the starting state with a double circle, the winning state with a triple circle, each losing state with a square, and every other state with a single circle. These will be the vertices in our graph.



Finally, we draw edges between states to show what would happen if you cross the river carrying one or zero objects.



The puzzle is solved by finding a route through this graph (in the next chapter we shall see that the technical term for this is *path*) from the starting state to the finishing state, never passing through a losing state. One possible route informs you to transport them all by first taking over the goat, coming back (alone), transporting the cabbage, coming back with the goat, transporting the wolf, coming back (alone), and transporting the goat again. (One could argue that f/cgw is an unreachable state, since you would first need to win in order to lose in that way.)

Theoretically, this strategy could be used to write an unbeatable chess-playing program: let each vertex represent a legal position (or state) in a chess game, and let the edges represent how making a move changes the position. Then trace back from the positions representing checkmates against the computer and mark the edges that lead to them, so the computer will not choose them. However, the limitations of time and space once again hound us: There are estimated to be between 10^{43} and 10^{50} legal chess positions.

Exercises

1. Complete the on-line graph drills found at www.ship.edu/~deensl/DiscreteMath/flash/ch7/sec7_3/planargraphs.html
2. Find the degree of every vertex in the graph for the cabbage, goat, and wolf puzzle.
3. Draw the complement of the cabbage, goat, and wolf graph.
4. Draw a graph that would represent the moves in the puzzle if you were allowed to transport zero, one, or two objects at a time. Notice that the vertices are the same.
5. A puzzle related to the cabbage, goat, and wolf puzzle called Missionaries and Cannibals can be found at www.plastelina.net/games/game2.html. While there is nothing funny about missionaries being attacked, it is an interesting puzzle. Play that game, and use a graph to show all the states, including a winning solution.
6. Suppose there is a party attended by eleven people. Is it possible for each person to shake hands with exactly five other people? Explain. (Hint: Notice the name of Theorem 34.1.)
7. Prove that for any graph the number of vertices with an odd degree is even.
8. Prove that if a graph $G = (V, E)$ is complete, then $|E| = \frac{|V|(|V|-1)}{2}$.

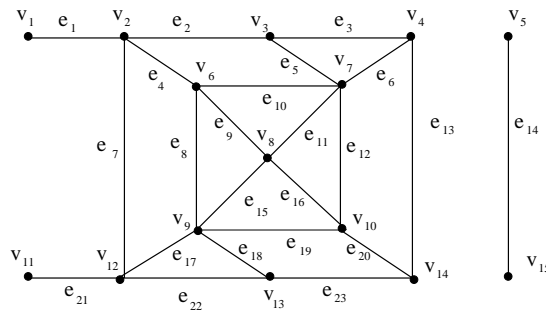
Chapter 35

Paths and cycles

35.1 Walks and paths

In all the following definitions, assume $G = (V, E)$ is a graph.

A *walk* from vertex v to vertex w , $v, w \in V$, is a sequence alternating between vertices in V and edges in E , written $v_0e_1v_1e_2\ldots v_{n-1}e_nv_n$ where $v_0 = v$ and $v_n = w$ and for all i , $1 \leq i < n$, $e_i = (v_{i-1}, v_i)$. v is called the *initial* vertex and w is called the *terminal* vertex. A walk is *trivial* if it contains only one vertex and no edges; otherwise it is *nontrivial*. The *length* of a walk is the number of edges (not necessarily distinct, since an edge may appear more than once). In the graph below, some examples of non-trivial walks are $v_1e_1v_2e_4v_6e_9v_8e_{11}v_7e_{10}v_6e_8v_9$ with length 6, $v_5e_{14}v_{15}$ with length 1, and $v_{11}e_{21}v_{12}e_{17}v_9e_{18}v_{13}e_{22}v_{12}e_{17}v_9e_{18}v_{13}e_{23}v_{14}$ with length 7.



A graph is *connected* if for all $v, w \in V$, there exists a walk in G from v to w . This graph is not connected, since no walk exists from v_5 or v_{15} to any of the other vertices. However, the subgraph excluding v_5 , v_{15} , and e_{14} is connected.

A *path* is a walk that does not contain a repeated edge. $v_1e_1v_2e_4v_6e_9v_8e_{11}v_7e_{10}v_6e_8v_9$ is a path, but $v_{11}e_{21}v_{12}e_{17}v_9e_{18}v_{13}e_{22}v_{12}e_{17}v_9e_{18}v_{13}$ is not. If the walk contains no repeated vertices, except possibly the initial and terminal, then the walk is *simple*. $v_1e_1v_2e_4v_6e_9v_8e_{11}v_7e_{10}v_6e_8v_9$ is not simple, since v_6 occurs twice. Its subpath $v_8e_{11}v_7e_{10}v_6e_8v_9$ is simple.

Propositions about walks and paths require careful use of the notation we use to denote a walk. Observe the process used in this example.

Theorem 35.1 *If $G = (V, E)$ is a connected graph, then between any two distinct vertices of G there exists a simple path in G .*

The first thing you must do is understand what this theorem is claiming. A quick read might mislead someone to think that this is merely stating the definition of *connected*. Be careful—being connected only means that any two vertices are connected by a walk, but this theorem claims that they are connected by a simple path, that is, without repeated edges or vertices. It turns out that whenever a walk exists, a simple path must also.

Lemma 35.1 *If $G = (V, E)$ is a graph, $v, w \in V$, and there exists a walk from v to w , then there exists a simple path from v to w .*

We will use a notation where we put subscripts on ellipses. The ellipses stand for subwalks which we would like to splice in and out of walks we are constructing.

Proof. Suppose $G = (V, E)$ is a graph, $v, w \in V$, and there exists a walk $c = v_0 e_1 v_1 \dots e_n v_n$ in G with $v_0 = v$ and $v_n = w$.

First, suppose that c contains a repeated edge, say $e_x = e_y$ for $0 \neq x < y \neq n$, so that we can write $c = v_0 e_1 v_1 \dots v_{x-1} e_x v_x \dots v_{y-1} e_y v_y e_{y+1} \dots e_n v_n$. Since $e_x = e_y$, then either $v_{x-1} = v_{y-1}$ and $v_x = v_y$ or $v_{x-1} = v_y$ and $v_x = v_{y-1}$. Then create a new walk c' in this way:

If $v_{x-1} = v_{y-1}$ and $v_x = v_y$, then let $c' = v_0 e_1 v_1 \dots v_{x-1} e_x v_x e_{y+1} \dots e_n v_n$. Otherwise (if $v_{x-1} = v_y$ and $v_x = v_{y-1}$), let $c' = v_0 e_1 v_1 \dots v_{x-1} e_{y+1} \dots e_n v_n$. Repeat this process until we obtain a walk c'' with no repeated edges. (If c had no repeated edges in the first place, then $c'' = c$.)

Next, suppose that c'' contains a repeated vertex, say, $v_x = v_y$ for $x < y$, so that we can write $c = v_0 e_1 v_1 \dots v_x e_x v_{x+1} \dots v_y e_y v_{y+1} \dots e_n v_n$. Then create a new walk $c''' = v_0 e_1 v_1 \dots v_x e_x v_{x+1} \dots e_n v_n$. Repeat this process until we obtain a walk c^{iv} with no repeated vertices.

Then c^{iv} is a walk from v to w , and since it repeats neither vertex nor edge, it is a simple path. \square

To meet the burden of this existence proof, we produced a walk that fulfills the requirements. This is a constructive proof, and it gives an algorithm for deriving a simple path from any walk in an undirected graph. It also makes a quick proof for the earlier theorem.

Proof (of Theorem 35.1). Suppose $G = (V, E)$ is a connected graph and $v, w \in V$. By definition of connected, there exists a walk from v to w . By Lemma 35.1, there exists a simple path from v to w . \square

35.2 Circuits and cycles

*closed
circuit
cycle*

If $v = w$ (that is, the initial and terminal vertices are the same), then the walk is *closed*. A *circuit* is a closed path. A *cycle* is a simple circuit. In the earlier example, $v_6 e_9 v_8 e_{11} v_7 e_{12} v_{10} e_{16} v_8 e_{15} v_9 e_8 v_6$ is a circuit, but not a cycle, since v_8 is repeated. $v_2 e_4 v_6 e_8 v_9 e_{17} v_{12} e_7 v_2$ is a cycle. If a circuit or cycle c comprises the entire graph (that is, for all $v \in V$ and $e \in E$, v and e appear in c), then we will say that the graph is a circuit or cycle.

Proofs of propositions about circuits and cycles become monstrous because the things we must demonstrate proliferate. To show that something is a cycle requires showing that it is closed, that it is simple, and that it is a path, in addition to the specific requirements of the present proposition.

Theorem 35.2 *If $G = (V, E)$ is a connected graph and for all $v \in V$, $\deg(v) = 2$, then G is a circuit.*

This requires us to show a walk that is a circuit and comprises the entire graph.

Proof. Suppose $G = (V, E)$ is a connected graph and for all $v \in V$, $\deg(v) = 2$.

First suppose $|V| = 1$, that is, there is only one vertex, v . Since $\deg(v) = 2$, this implies that there is only one edge, $e = (v, v)$. Then the circuit vev comprises the entire graph.

This looks like the beginning of a proof by induction, but actually it is a division into cases. We are merely getting a special case out of the way. We want to use the fact that there can be no self-loops, but that is true only if there are more than one vertex.

Next suppose $|V| > 1$. By Exercise 6, G has no self-loops. Then construct a walk c in this manner: Pick a vertex $v_1 \in V$ and an edge $e_1 = (v_1, v_2)$. Since $\deg(v_1) = 2$, e must exist, and since G contains no self-loops, $v_1 \neq v_2$. Since $\deg(v_2) = 2$, there exists another edge, $e_2 = (v_2, v_3) \in E$. Continue this process until we reach a vertex already visited, so that we can write $c = v_1 e_1 v_2 \dots e_{x-1} v_x$ where $v_x = v_i$ for some i , $1 \leq i < x$. We will reach such a vertex eventually because V is finite.

We have constructed a walk. We must show that it meets the requirements we are looking for.

Only one vertex in c is repeated, since reaching a vertex for the second time stops the building process. Hence c is simple.

Since we never repeat a vertex (until the last), each edge chosen leads to a new vertex, hence no edge is repeated in c , so c is a path.

We are always choosing the edge other than the one we took into a vertex, so $i \neq x - 1$.

Suppose $i \neq 1$. Since no other vertex is repeated, v_{i-1} , v_{i+1} , and v_{x-1} are distinct. Therefore, distinct edges (v_{i-1}, v_i) , (v_i, v_{i+1}) , and (v_{x-1}, v_i) all exist, and so $\deg(v_i) \geq 3$. Since $\deg(v_i) = 2$, this is a contradiction. Hence $i = 1$. Moreover, $v_1 = v_x$ and c is closed.

As a closed, simple path, c is a circuit.

Suppose that a vertex $v \in V$ is not in c , and let v' be any vertex in c . Since G is connected, there must be a walk, c' from v to v' , and let edge e' be the first edge in c' (starting from v') that is not in c , and let v'' be an endpoint in c' in c . Since two edges incident on v'' occur in c , accounting for e' means that $\deg(v'') \geq 3$. Since $\deg(v_i) = 2$, this is a contradiction. Hence there is no vertex not in c .

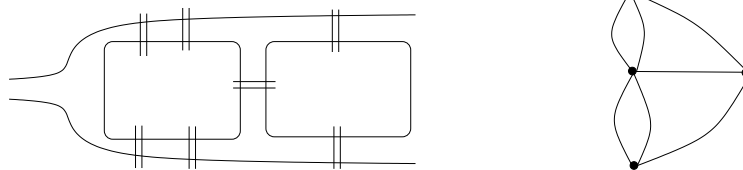
Suppose that an edge $e \in E$ is not in c , and let v be an endpoint of e . Since v is in the circuit, there exist distinct edges e_1 and e_2 in c that are incident on v , implying $\deg(v) \geq 3$. Since $\deg(v) = 2$, this is a contradiction. Hence there is no edge not in c .

Therefore, c is a circuit that comprises the entire graph, and G is a circuit. \square

The reasoning becomes very informal, especially the last two point about the circuit comprising the whole graph. However, make sure you see that the basic logic is still present: These are merely proofs of set emptiness.

35.3 Euler circuits and Hamiltonian cycles

Leonhard Euler proposed a problem based on the bridges in the town of Königsberg, Prussia. Two branches in of the Pregel River converge in the town, delineating it into a north part, a south part, an east part, and an island in the middle, as shown below. In Euler's time, the east part had one bridge to each the north part and the south part, the island had two bridges each to the north part and the south part, and one bridge connected the east part and the island. Supposing your house was in any of the four parts, is it possible to walk around town (beginning and ending at your house) and cross every bridge exactly once?



We can turn this into a graph problem by representing the information with a graph whose vertices stand for the parts of town and whose edges stand for the bridges, as displayed above. Let

$G = (V, E)$ be a graph. An *Euler circuit* of G is a circuit that contains every vertex and every edge. (Since it is a circuit, this also means that an Euler circuit contains every edge exactly once. Vertices, however, may be repeated.) The question now is whether or not this graph has an Euler circuit. We can prove that it does not, and so such a stroll about town is impossible.

Theorem 35.3 *If a graph $G = (V, E)$ has an Euler circuit, then every vertex of G has an even degree.*

Proof. Suppose graph $G = (V, E)$ has an Euler circuit $c = v_0 e_1 v_1 \dots e_n v_n$, where $v_0 = v_n$. For all $e \in E$, e appears in c exactly once, and for all $v \in V$, v appears at least once, by definition of Euler circuit. Suppose $v_i \in V$, $v_i \neq v_1$, and let x be the number of times v_i appears in c . Since each appearance of a vertex corresponds to two edges, $\deg(v_i) = 2x$, which is even by definition.

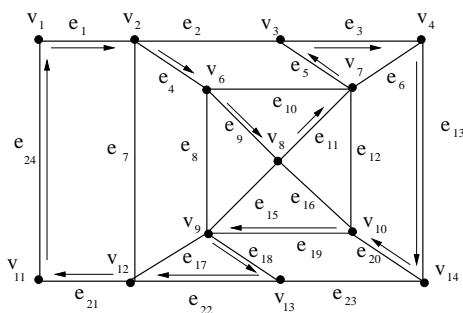
For v_1 , let y be the number of times it occurs in c besides as an initial or terminal vertex. These occurrences correspond to $2y$ incident edges. Moreover, e_1 and e_n are incident on v_1 , and hence $\deg(v_1) = 2y + 1 + 1 = 2(y + 1)$, which is even by definition. (If either e_1 and e_n are self-loops, they have already been accounted for once, and we are rightly counting them a second time.)

Therefore, all vertices in G have an even degree. \square

The northern, eastern, and southern parts of town each have odd degrees, so by the contrapositive of this theorem, no Euler circuit around town exists.

Hamiltonian cycle

Another interesting case is that of a *Hamiltonian cycle*, which for a graph $G = (V, E)$ is a cycle that includes every vertex in V . Since it is a cycle, this means that no vertex or edge is repeated; however, not all the edges need to be included. We reserve one Hamiltonian cycle proof for the exercises, but here is a Hamiltonian cycle in a graph similar to the one at the beginning of this chapter (with the disconnected subgraph removed).



Exercises

1. Complete the on-line graph drills found at www.ship.edu/~deensl/DiscreteMath/flash/ch7/sec7_1/euler.html and www.ship.edu/~deensl/DiscreteMath/flash/ch7/sec7_7/hamiltongraphs.html

2. Draw a graph that has a simple walk that is not a path.

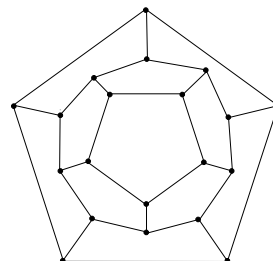
Prove. Assume $G = (V, E)$ is a graph.

3. If G is connected, then $|E| \geq |V| - 1$.
4. If G is connected, $|V| \geq 2$, and $|V| > |E|$, then G has a vertex of degree 1.
5. If G is not connected, then \overline{G} is connected.
6. If G is connected, for all $v \in V$, $\deg(v) = 2$, and $|V| > 1$, then G has no self-loops.
7. Every circuit in G contains a subwalk that is a cycle.
8. If for all $v \in V$, $\deg(v) \geq 2$, then G contains a cycle.
9. If $v, w \in V$ are part of a circuit c and G' is a subgraph of G formed by removing one edge of c , then there exists a path from v to w in G' .

10. If G has no cycles, then it has a vertex of degree 0 or 1.

11. If G has an Euler circuit, then it is connected.

12. Find a Hamiltonian cycle in the following graph (copy it and highlight the walk on your copy).



13. If G has a non-trivial Hamiltonian cycle, then G has a subgraph $G' = (V', E')$ such that

- G' contains every vertex of G ($V' = V$)
- G' is connected,
- G' has the same number of edges as vertices ($|V'| = |E'|$), and
- every vertex of G' has degree 2.

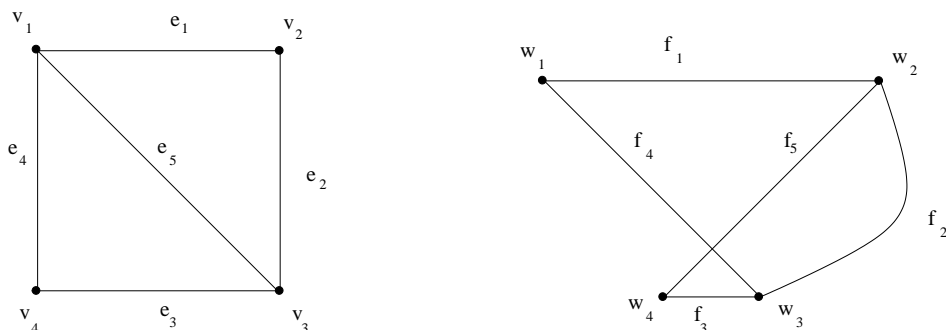
14. Reinterpret the Königsburg bridges problem by making a graph whose vertices represent the bridges and whose edges represent land connections between bridges. What problem are you solving now?

Chapter 36

Isomorphisms

36.1 Definition

We have already seen that the printed shape of the graph—the placement of the dots, the resulting angles of the lines, any curvature of the lines—is not of the essence of the graph. The only things that count are the names of the vertices and edges and the abstract shape, that is, the connections that the edges define. However, consider the two graph representations below, which illustrate the graphs $G = (V = \{v_1, v_2, v_3, v_4\}, E = \{e_1 = (v_1, v_2), e_2 = (v_2, v_3), e_3 = (v_3, v_4), e_4 = (v_4, v_1), e_5 = (v_1, v_3)\})$ and $G' = (W = \{w_1, w_2, w_3, w_4\}, F = \{f_1 = (w_1, w_2), f_2 = (w_2, w_3), f_3 = (w_3, w_4), f_4 = (w_3, w_1), f_5 = (w_4, w_2)\})$.



These graphs have much in common. Both have four vertices and five edges. Both have two vertices with degree two and two vertices with degree three. Both have a Hamiltonian cycle ($v_1e_1v_2e_2v_3e_3v_4e_4v_1$ and $w_1f_1w_2f_2w_3f_3w_4f_4w_1$, leaving out e_5 and f_2 , respectively) and two other cycles (involving e_5 and f_2). In fact, if you imagine switching the positions of w_1 and w_2 , with the edges sticking to the vertices as they move, and then doing a little stretching and squeezing, you could transform the second graph until it appears identical to the first.

In other words, these two really are the same graph, in a certain sense of sameness. The only difference is the arbitrary matter of names for the vertices and edges. We can formalize this by writing renaming functions, $g : V \rightarrow W$ and $h : E \rightarrow F$.

v	$g(v)$	e	$h(e)$
v_1	w_2	e_1	f_5
v_2	w_4	e_2	f_3
v_3	w_3	e_3	f_4
v_4	w_1	e_4	f_1
		e_5	f_2

The term for this kind of equivalence is *isomorphism*, from the Greek roots *iso* meaning “same”

isomorphism

isomorphic

and *morphē* meaning “shape.” This is a way to recognize identical abstract shapes of graphs, that two graphs are the same up to renaming. Let $G = (V, E)$ and $G' = (W, F)$ be graphs. G is *isomorphic* to G' if there exist one-to-one correspondences $g : V \rightarrow W$ and $h : E \rightarrow F$ such that for all $v \in V$ and $e \in E$, v is an endpoint of e iff $g(v)$ is an endpoint of $h(e)$. The two functions g and h , taken together, are usually referred to as the *isomorphism* itself, that is, there exists an isomorphism, namely g and h , between G and G' .

As we shall see, what characterizes isomorphisms is that they preserve properties, that is, there are many graph properties which, if they are true for one graph, are true for any other graph isomorphic to that graph. Graph theory is by no means the only area of mathematics where this concept occurs. Group theory (a main component of modern algebra) involves isomorphisms as functions that map from one group to another in such a way that operations are preserved. Isomorphisms define equivalences between matrices in linear algebra. The main concept of isomorphism is the independence of structure from data. If an isomorphism exists between two things, it means that they exist as parallel but equivalent universes, and everything that happens in one universe has an equivalent event in the other that keeps them in step.

36.2 Isomorphic invariants

isomorphic invariant

An *isomorphic invariant* is a property that is preserved through an isomorphism. In other words, proposition P is an isomorphic invariant if for any graphs G and G' , if $P(G)$ and G is isomorphic to G' , then $P(G')$. Some isomorphic invariants are simple and therefore quite easy to prove; others require subtle and complex proofs. To illustrate what is meant, consider this theorem.

Theorem 36.1 *For any $k \in \mathbb{N}$, the proposition $P(G) = “G \text{ has a vertex of degree } k”$ is an isomorphic invariant.*

This is not a difficult result to prove as long as one can identify what burden is required. Being an isomorphic invariant has significance only when two pieces are already in place: We have two graphs known to be isomorphic and that the proposition is true for one of those graphs.

Proof. Suppose $k \in \mathbb{N}$, $G = (V, E)$ is a graph which has a vertex $v \in V$ with degree k , and G' is a graph to which G is isomorphic.

Now we can assume and use the definition of isomorphic (those handy one-to-one correspondences must exist), and we must prove that G' has a vertex of degree k . The definition of degree also comes into play, especially in distinguishing between self-loops and other edges.

By definition of degree, there exist edges $e_1, e_2, \dots, e_n \in E$, non self-loops, and $e'_1, e'_2, \dots, e'_m \in E$, self-loops, that are incident on v , such that $k = n + 2m$.

By the definition of isomorphism, there exist one-to-one correspondences g and h with the isomorphic property.

Saying “with the isomorphic property” spares the trouble of writing out “for all $v' \in V$ ” etc, and instead more directly we claim

For each e_i , $1 \leq i \leq n$, $h(e_i)$ has $g(v)$ as one endpoint, and for each e'_j , $1 \leq j \leq m$, $h(e'_j)$ has $g(v)$ as both endpoints, and no other edge has $g(v)$ as an endpoint. Each $h(e_i)$ and $h(e'_j)$ is distinct since h is one-to-one. Hence

$$\deg(g(v)) = n + 2m = k$$

Therefore G' has a vertex, namely $g(v)$, with degree k . \square

36.3 The isomorphic relation

The proof in the previous section contained the awkward clause “ G' is a graph to which G is isomorphic.” This was necessary because our definitions directly allow only discussion of when one graph is isomorphic to another, not necessarily the same thing as, for example, “ G' is isomorphic to G .” However, your intuition should suggest to you that these in fact are the same thing, that if G is isomorphic to G' then G' is also isomorphic to G , and we may as well say “ G and G' are isomorphic to each other.”

If we think of isomorphism as a relation (the relation “is isomorphic to”), this means that the relation is symmetric. We can go further, and observe that isomorphism defines an equivalence class for graphs.

Theorem 36.2 *The relation R on graphs defined that $(G, G') \in R$ if G is isomorphic to G' is an equivalence relation.*

Proof. Suppose R is a relation on graphs defined that $(G, G') \in R$ if G is isomorphic to G' .

Reflexivity. Suppose $G = (V, E)$ is a graph. Let $g = i_V$ and $h = i_E$ (that is, the identity functions on V and E , respectively). By Exercise 8 of Chapter 25, g and h are one-to-one correspondences. Now suppose $v \in V$ and $e \in E$. Then $g(v) = v$ and $h(e) = e$, so if v is an endpoint of e then $g(v)$ is an endpoint of $h(e)$, and if $g(v)$ is an endpoint of $h(e)$, then v is an endpoint of e . Hence g and h are an isomorphism between G and itself, $(G, G) \in R$, and R is reflexive.

Symmetry. Suppose $G = (V, E)$ and $G' = (W, F)$ are graphs and $(G, G') \in R$, that is, G is isomorphic to G' . Then there exist one-to-one correspondences $g : V \rightarrow W$ and $h : E \rightarrow F$ with the isomorphic property. Since g and h are one-to-one correspondences, their inverses, $g' : W \rightarrow V$ and $h' : F \rightarrow E$, respectively, exist. Suppose $w \in W$ and $f \in F$. By definition of inverse, $g(g'(w)) = w$ and $h(h'(f)) = f$. Suppose w is an endpoint of f . By definition of isomorphism, $g'(w)$ is an endpoint of $h'(f)$. Then suppose $g'(w)$ is an endpoint of $h'(f)$. Also by definition of isomorphism, w is an endpoint of f . Hence g' and h' are an isomorphism from G' to G , $(G', G) \in R$, and R is symmetric.

Transitivity. Exercise 2.

Therefore, R is an equivalence relation. \square

There we have it—our “certain sense of sameness” is an example of mathematical equivalence. This is what abstract thinking is about, being able to ignore the detail to see what unifies. This proof, especially the part about symmetry, has much important review tucked into it. We had to notice that g and h were one-to-one correspondences before we could assume inverse functions. We see again the pattern of analysis (taking apart what it means for G to be isomorphic to G') and synthesis (constructing what it means for G' to be isomorphic to G). The isomorphic property is an “iff” property, so we were required to prove the matter of endpoints going either direction.

36.4 Final bow

We end this chapter, this part, and the entire book with a real workout.

Theorem 36.3 *Having a Hamiltonian cycle is an isomorphic invariant.*

Proof. Suppose $G = (V, E)$ and $G' = (W, F)$ are isomorphic graphs, and suppose that G has a Hamiltonian cycle, say $c = v_1 e_1 v_2 \dots e_{n-1} v_n$ (where $v_1 = v_n$). By the definition of isomorphism, there exist one-to-one correspondences g and h with the isomorphic property.

Suppose e_i is any edge in c , incident on v_i and v_{i+1} . By the definition of isomorphism, $h(e_i)$ is incident on $g(v_i)$ and $g(v_{i+1})$. This allows us to construct the walk $c' = g(v_1)h(e_1)g(v_2) \dots h(e_{n-1})g(v_n)$.

By definition of Hamiltonian cycle, every $v \in V$ occurs in c . Since g is onto, every vertex $w \in W$ occurs in c' , and since g is one-to-one, no vertex occurs more than once in c' except $g(v_1) = g(v_n)$. Also by definition of Hamiltonian cycle, no edge $e \in E$ occurs more than once in c . Since h is one-to-one, no edge $f \in F$ occurs more than once in c' .

Hence c' is a Hamiltonian cycle. Therefore, having a Hamiltonian cycle is an isomorphic invariant. \square

Exercises

1. Complete the on-line graph drills found at www.ship.edu/~deensl/DiscreteMath/flash/ch3/sec7_3/isomorphism.html
 2. Prove that the relation “is isomorphic to” over graphs is transitive.
 3. Prove that if graphs G and H are isomorphic, then their complements, \overline{G} and \overline{H} are isomorphic.
 4. G has n vertices, for $n \in \mathbb{N}$.
 5. G has m edges, for $m \in \mathbb{N}$.
 6. G has n vertices of degree k , for $k \in \mathbb{N}$.
 7. G has a circuit of length k , for $k \in \mathbb{N}$.
 8. G has a cycle of length k , for $k \in \mathbb{N}$.
 9. G is connected.
 10. G has an Euler circuit.
- Prove that the following are isomorphic invariants.

Bibliography

- [1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. McGraw Hill and the MIT Press, Cambridge, MA, second edition, 1996.
- [2] Mary Chase. *Harvey*. Dramatists Play Service, Inc., New York, 1971. Originally published in 1944.
- [3] G.K. Chesteron. *Orthodoxy*. Image Books, Garden City, NY, 1959.
- [4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. McGraw-Hill and MIT Press, second edition, 2001.
- [5] Sussana S. Epp. *Discrete Mathematics with Applications*. Thomson Brooks/Cole, Belmont, CA, third edition, 2004.
- [6] Matthias Felleisen and Daniel P Friedman. *The Little MLer*. MIT Press, Cambridge, MA, 1998.
- [7] H. W. Fowler. *A Dictionary of Modern English Usage*. Oxford University Press, Oxford, 1965. Originally published 1926. Revised by Ernest Gowers.
- [8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [9] Karel Hrbacek and Thomas Jech. *Introduction to Set Theory*. Marcel Dekker, New York, 1978. Reprinted by University Microfilms International, 1991.
- [10] Iu I Manin. *A course in mathematical logic*. Graduate texts in mathematics. Springer Verlag, New York, 1977. Translated from the Russian by Neal Koblitz.
- [11] George Pólya. *Induction and Analogy in Mathematics*. Princeton University Press, 1954. Volume I of *Mathematics and Plausible Reasoning*.
- [12] Michael Stob. Writing proofs. Unpublished, September 1994.
- [13] Geerhardus Vos. *Biblical Theology*. Banner of Truth, Carlisle, PA, 1975. Originally published by Eerdmans, 1948.
- [14] Samuel Wagstaff. Fermat’s last theorem is true for any exponent less than 1000000 (abstract). *AMS Notices*, 23(167):A–53, 1976.