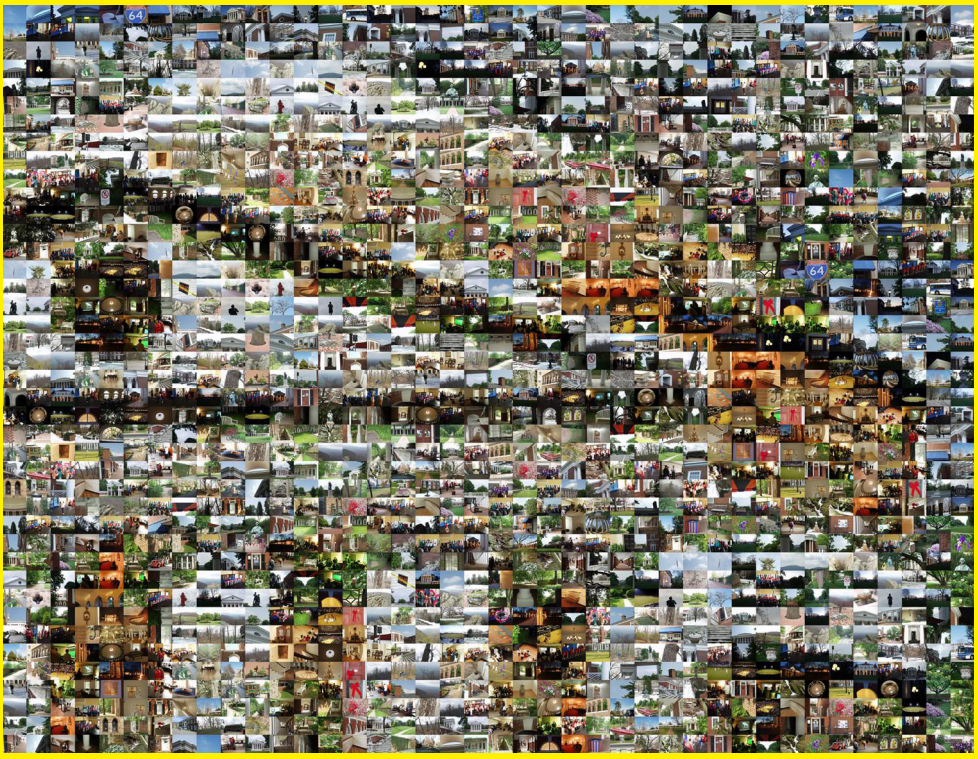
Computational Thinking

A Whirlwind Introduction
to the Third Millennial Liberal Art
from Ada and Euclid
to Quantum Computing
and the World Wide Web



David Evans
University of Virginia
January 2009

1

Computing

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

Edsger Dijkstra, 1972 Turing Award Lecture

The first million years of hominid tool development focused on developing tools to amplify, and eventually mechanize, our physical abilities to enable us to move faster, reach higher, and hit harder. We have developed tools that amplify physical force by the trillions and increase the speeds and distances we can travel by the thousands.

Tools that amplify intellectual abilities are much rarer. While many animals have developed tools to amplify their physical abilities, only humans have developed tools to substantially amplify our intellectual abilities, and it is those advances that have allowed humans to dominate the planet. The first key intellect amplifier was language. Language provided the ability to transmit our thoughts to others, as well as to use our own minds more effectively. The next key intellect amplifier was writing, which enabled the storage and transmission of thoughts over time and distance.

Computing is the ultimate mental amplifier—computers have the ability to mechanize any intellectual activity we can imagine. Automatic computing radically changes how humans solve problems, and even the kinds of problems we can imagine solving. Computing has changed the world more than any other invention of the past hundred years, and the power of automatic computing has come to pervade nearly all human endeavors. Yet, we are just at the beginning of the computing revolution; today's computing offers just a glimpse of the potential impact of computing.

There are two reasons why everyone should study computing:

1. Nearly all of the coolest, most exciting, and most important technologies of today and tomorrow are driven by computing.
2. Understanding computing illuminates deep insights and questions into the nature of our minds, our culture, and our universe.

Anyone who has submitted a query to Google, watched *Toy Story*, had LASIK eye surgery, made a cell phone call, seen a Cirque Du Soleil show, shopped with a credit card, or microwaved a pizza should be convinced of the first reason. None of these would be possible without the tremendous advances in computing over the past half century.

It may be true that you have to be able to read in order to fill out forms at the DMV, but that's not why we teach children to read. We teach them to read for the higher purpose of allowing them access to beautiful and meaningful ideas.
Paul Lockhart, *Lockhart's Lament*

Although this book will touch on on some exciting applications of computing, our primary focus is on the second reason, which may seem more surprising. Computing changes how we think about problems and how we understand the world. The goal of this book is to teach you that new way of thinking.

1.1 Processes, Procedures, and Computers

Computer science is the study of *information processes*. A process is a sequence of steps. Each step changes the state of the world in some small way, and the result of all the steps produces some goal state. For example, baking a cake, mailing a letter, and planting a tree are all processes. Because they involve physical things like sugar and dirt, however, they are not pure information processes. Computer science focuses on processes that involve abstract information, rather than physical things.

The boundaries between the physical world and pure information processes, however, are often fuzzy. Real computers operate in the physical world: they obtain input through physical actions by a user (e.g., hitting keys), and produce physical outputs (e.g., displaying an image on a screen). By focusing on abstract information, instead of the physical ways of representing and manipulating information, we can simplify computation to the point where we can understand and reason about it more easily.

A *procedure* is a description of a process. A simple process can be described just by listing the steps. The list of steps is the procedure; the act of following them is the process. If the description can be followed without any thought, we call it a *mechanical procedure*.

A mathematician is a machine for turning coffee into theorems.
Attributed to Paul Erdös

For example, here is a procedure for making coffee, adapted from the actual directions that come with a major coffeemaker:

1. Lift and open the coffeemaker lid.
2. Place a basket-type filter into the filter basket.
3. Add the desired amount of coffee and gently shake to level the coffee.
4. Fill the decanter with cold, fresh water to the desired capacity.
5. Pour the water into the water reservoir.
6. Close the lid.

7. Place the empty decanter on the warming plate.
8. Press the ON button.

There are lots of limitations of describing processes by just listing steps this way. First of all, natural languages are very imprecise and ambiguous. The steps described rely on the operator knowing lots of unstated assumptions such as which way the filter should go in the basket, how much coffee should go in the filter (but isn't our end goal to make coffee? here, we need to know that the directions mean coffee grounds), and that the coffeemaker needs to be plugged in to a power outlet and should be sitting on a flat surface. We could, of course, add lots more details to our procedure and make the language more precise than this. Even when a lot of effort is put into writing precisely and clearly, however, natural languages such as English are inherently ambiguous. This is why the United States tax code is 3.4 million words long, but lawyers can still spend years arguing over what it really means.

I have no idea what you're talking about when you say "ask".
Bill Gates, deposition in Microsoft anti-trust trial

Another problem with this way of describing a procedure is that the size of the description is proportional to the number of steps in the process. This is fine for simple processes with only a few steps, but the processes we want to execute on computers involve trillions of steps. This means we need more efficient ways to describe them than just listing each step one-by-one. The language we use to program computers enable a few statements to describe many different possible steps, and provide ways for defining repetition and changing the steps produced by a given procedure.

To program computers, we need tools that allow us to describe processes precisely and succinctly. Since the procedures will need to be carried out by a machine, every step needs to be described; we cannot rely on the operator having "common sense" (for example, to know how to fill the coffeemaker with water without explaining that water comes from a faucet, and how to turn the faucet on). Instead, we need mechanical procedures that can be followed without any thinking.

A *computer* is a machine that can:

1. Accept input. Input could be entered by a human typing at a keyboard, received over a network, or provided automatically by sensors attached to the computer.
2. Execute a mechanical procedure. Recall that a mechanical procedure is defined as a procedure where each step can be executed without any thought.
3. Produce output. Output could be printed data displayed to a human, but it could also be anything that effects the world outside the computer such as electrical signals that control how a device operates.

Computers exist in a wide range of forms, and thousands of computers are hidden in devices we use everyday but don't think of as computers such as cars, phones, TVs, microwave ovens, and access cards. Our primary focus in this book is on *universal computers*, which are computers that can perform *all* possible mechanical computations except for practical limits on space and time. We will explain what this means more precisely in Chapter 13.

1.2 Computing Power

How can we measure the power of a computing machine?

For physical machines, we can compare the power of different machines by measuring the amount of mechanical work they can perform within a given amount of time. This power can be captured with units like *horsepower* and *watt*. Physical power is not a very useful measure of computing power, though, since the amount of computing achieved for the same amount of energy varies greatly. Energy is consumed when a computer operates, not the desired output.

The two main properties we can measure about the power of a computing machine are:

1. *How much* information it can process?
2. *How fast* can it process?

We will defer considering the second property until later chapters, but consider the first question here.

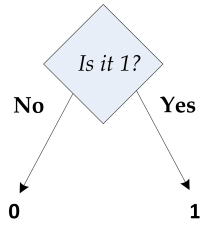
1.2.1 Information

Informally, we use *information* to mean knowledge. But to understand information quantitatively, as something we can measure, we need a more precise way to think about information.

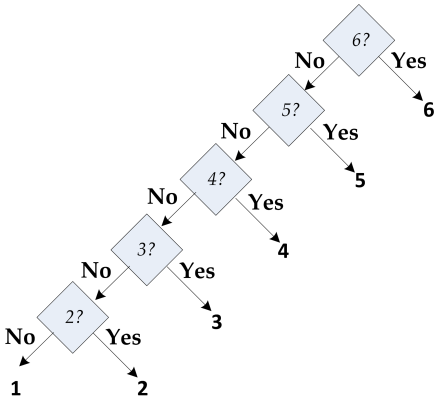
The way computer scientists measure information is based on how the knowledge of some thing changes as a result of obtaining the information.
bit The primary unit of information is a *bit*. One bit of information halves the amount of uncertainty. It is equivalent to answering a "yes" or "no" question, where either answer is equally likely beforehand. Before learning the answer, there were two possibilities; after learning the answer, there is one. We call a question with two possible answers a *binary question*. Since a bit can have two possible values, we often represent the values as **0** and **1**.

For example, suppose we perform a fair coin toss but do not reveal the result. Half of the time, the coin will land "heads", and the other half of the time the coin will land "tails". Without knowing any more information, our chances of guessing the correct answer are $\frac{1}{2}$. One bit of information would be enough to convey either "heads" or "tails"; we can use **0** to represent "heads" and **1** to represent "tails". So, the amount of information in a coin toss is one bit.

Similarly, one bit of information distinguishes between the values 0 and 1:



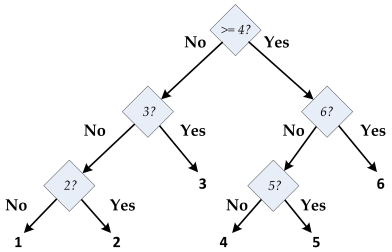
What about rolling a six-sided die? There are six equally likely possible outcomes, so without any more information we have a one in six chance of guessing the correct value. One bit is not enough to identify the actual number, since one bit can only distinguish between two possible values. We could use five binary questions like this:



This is quite inefficient, though, since in the worst case we need five questions to identify the value. Can we identify the value with fewer than 5 questions?

Our goal is to identify questions where the "yes" and "no" answers are a priori equally likely—that way, each answer provides the most information possible. This is not the case if we start with, "Is the value 6?", since that answer is expected to be "yes" only one time in six. Instead, we should

start with a question like, "Is the value at least 4?". Here, we expect the answer to be "yes" one half of the time, and the "yes" and "no" answers are equally likely. If the answer is "yes", we know the result is 4, 5, or 6. With two more bits, we can distinguish between these three values (note that two bits is actually enough to distinguish among *four* different values, so some information is wasted here). Similarly, if the answer to the first question is no, we know the result is 1, 2, or 3. We need two more bits to distinguish which of the three values it is. Thus, with three bits, we can distinguish all six possible outcomes.



Three bits can convey more information that just six possible outcomes, however. From the information tree, we can see there are some questions where the answer is not equally likely to be "yes" and "no" (for example, we expect the answer to "Is the value 3?" to be "yes" only one out of three times). This means we are not obtaining a full bit of information with each question.

Each bit doubles the number of possibilities we can distinguish, so with three bits we can distinguish between $2 * 2 * 2 = 8$ possibilities. In general, with n bits, we can distinguish between 2^n possibilities. Conversely, distinguishing among k possible values requires $\log_2 k$ bits. The logarithm is defined such that if $a = b^c$ then $\log_b a = c$. Since each bit has two possibilities, we use the logarithm base 2 to determine the number of bits needed to distinguish among a set of distinct possibilities. For our six-sided die, $\log_2 6 \approx 2.58$, so we need ≈ 2.58 binary questions. But, questions are discrete: we don't know how to ask $.58^{th}$ of a question, so we need to use three binary questions.

Figure 1.1 depicts a structure of binary questions for distinguishing among eight values. We call this structure a *tree*; we will see many useful applications of tree-like structures in this book. Computer scientists tend to draw trees upside down. The *root* is the top of the tree, and the *leaves* are the numbers at the bottom (**0**, **1**, **2**, . . ., **7**). There is a unique path from the root of the tree to each leaf. Thus, we can describe each of the eight possible values using the answers to the questions down the tree. For example, if the answers are "No", "No", and "No", we reach the leaf **0**; if the answers are

"Yes", "No", "Yes", we reach the leaf **5**. We can describe any non-negative integer using bits in this way, by just adding additional levels to the tree. For example, if we wanted to distinguish between 16 possible numbers, we would add a new question, "Is is $>= 8$?" to the top of the tree. If the answer is "No", we use the tree in Figure 1.1 to distinguished between 0 and 7. If the answer is "Yes", we use a tree similar to the one in Figure 1.1, but add 8 to each of the numbers in the questions and the leaves. The *depth* of the tree is the longest path from the root to any leaf. For the example tree, the depth is three. A binary tree of depth d can distinguish 2^d different values.

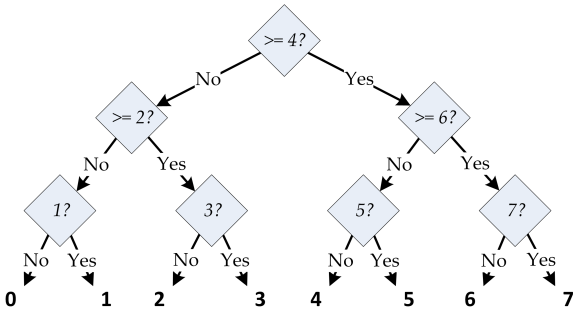


Figure 1.1. Using three bits to distinguish eight possible values.

One *byte* is defined as eight bits. Hence, one byte of information corresponds to eight binary questions, and can distinguish among 2^8 (256) different values. For larger amounts of information, we use metric prefixes, but instead of scaling by factors of 1000 they scale by factors of 2^{10} (1024). Hence, one *kilobyte* is 1024 bytes; one *megabyte* is 2^{20} (approximately one million) bytes; one *gigabyte* is 2^{30} (approximately one billion) bytes; and one *terabyte* is 2^{40} (approximately one trillion) bytes.

Exercise 1.1. Draw a binary tree for distinguishing among the sixteen numbers $0, 1, 2, \ldots, 15$ with the minimum possible depth.

Exercise 1.2. Draw a binary tree for distinguishing among the twelve months of the year with the minimum possible depth.

Excursion 1.1: How Much Data. How many bits are needed: (a) to uniquely identify any currently living human? (b) to uniquely identify any human who ever lived? (c) to identify any location on Earth within one square centimeter? (d) to uniquely identify any atom in the observable universe?

Exercise 1.3. The examples all use binary questions for which there are two possible answers. Suppose instead of basing our decisions on bits, we based it on *trits* where one trit can distinguish between three equally likely values. For each trit, we can ask a ternary question (a question with three possible answers).

a. [⋆] How many trits are needed to distinguish among eight possible values? (A convincing answer would show a ternary tree with the questions and answers for each node, and argue why it is not possible to distinguish all the values with a tree of lesser depth.)

b. [⋆] Devise a general formula for converting between bits and trits. That is, how many trits does it require to describe b bits of information?

Excursion 1.2: Guessing Numbers. The guess-a-number game starts with one player (the *chooser*) picking a number between 1 and 100 (inclusive) and secretly writing it down. The other player (the *guesser*) attempts to guess the number. After each guess, the chooser responds with "correct" (the guesser guessed the number and the game is over), "higher" (the actual number is higher than the guess), or "lower" (the actual number is lower than the guess).

a. Explain why the guesser can receive more than one bit of information for each response.

b. Assuming the chooser picks the number randomly (that is, all values between 1 and 100 are equally likely), what are the best first guesses? Explain why these guesses are better than any other guess. (Hint: there are two equally good first guesses.)

c. [⋆] What is the maximum number of guesses the second player should need to always find the number?

d. [⋆] What is the average number of guesses needed (assuming the chooser picks the number randomly as before)?

e. [⋆] Suppose instead of picking randomly, the chooser picks the number with the goal of maximizing the number of guesses the second player will need. What number should she pick?

f. [⋆] How should the guesser adjust her strategy if she knows the chooser is picking antagonistically?

g. [⋆⋆] What are the best strategies for both players in the adversarial guess-a-number game where chooser's goal is to pick a starting number that maximizes the number of guesses the guesser needs, and the guesser's goal is to guess the number using as few guesses as possible.

Excursion 1.3: Twenty Questions. The two-player game *twenty questions* starts with the first player (the *answerer*) thinking of an object, and declaring if the object is an animal, vegetable, or mineral (meant to include all non-living things). After this, the second player (the *questioner*), asks binary questions to try and guess the object the first player thought of. The first player answers each question "yes" or "no". The website *http://www.20q.net/* offers a web-based 20 questions game where a human acts as the answered and the computer as the questioner. The game is also sold as a $10 stand-alone toy (shown in the picture).

20Q Game

Image from ThinkGeek

a. How many different objects can be distinguished by a perfect questioner for the standard twenty questions game?

b. What does it mean for the questioner to play perfectly?

c. Try playing the 20Q game at *http://www.20q.net*. Did the computer guess your item?

d. Instead of just "yes" and "no", the 20Q game offers four different answers: "Yes", "No", "Sometimes", and "Unknown". (The website version of the game also has "Probably", "Irrelevant", and "Doubtful".) If all four answers were equally likely (and meaningful), how many items could be distinguished in 20 questions?

e. For an Animal, the first question 20Q asks is "Does it jump?" (note that 20Q will select randomly among a few different first questions). Is this a good first question?

f. [⋆] How many items do you think 20Q has data for?

g. [⋆] Speculate on how 20Q could build up its database.

1.2.2 Representing Data

We can use sequences of bits to represent all kinds of data. All we need to do is think of the right binary questions for which the bits give answers that allow us to represent each possible value. Next, we provide examples showing how bits can be used to represent numbers, poems, and pictures.

Numbers. In the previous section, we saw how to distinguish a set of items using a tree where each node asks a binary question, and the branches correspond to the "Yes" and "No" answers. A more compact way of writing down our decisions following the tree is to use **0** to encode a "No" answer, and **1** to encode a "Yes" answer. Then, we can describe a path to a leaf by a sequence of **0**s and **1**s—the "No", "No", "No" path to **0** is encoded as **000**, and the "Yes", "No", "Yes" path to **5** is encoded as **101**. This is known as *Binary Numbers*

the *binary number system*. Whereas the decimal number system uses ten as its base (there are ten decimal digits, and the positional values increase as powers of ten), the binary system uses two as its base (there are two binary digits, and the positional values increase as powers of two).

For example, the binary number **10010110** represents the decimal value 150. As in the decimal number system, the value of each binary digit depends on its position:

Binary:	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**
Value:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal Value:	128	64	32	16	8	4	2	1

There are only 10 types of people in the world: those who understand binary, and those who don't.
Infamous T-Shirt

By using more bits, we can represent larger numbers. With enough bits, we can represent any natural number this way. The more bits we have, the larger the set of possible numbers we can represent. As we saw with the binary decision trees, n bits can be used to represent 2^n different numbers.

Text. We can use a finite sequence of bits to describe *any* value that is selected from a well-defined finite set of possible values. One way to see this is to observe that we could give each item in the set a unique number, and then use that number to identify the item. Since we have seen that we can represent all the natural numbers with a sequence of bits, so once we have the mapping between each item in the set and a unique natural number, we can represent all of the item in the set. For the representation to be useful, though, we usually need a way to construct the corresponding number for any item directly since maintaining the set of all possible values is not possible.

For example, consider poems. The total number of possible poems is infinite, but for any given length there are a finite (albeit huge) number of possible poems. Enumerating all possible poems of length N is possible in theory, but practically infeasible. Even for haiku, which are short, 3-line poems, this is impossible; there are fewer atoms in the universe than the number of possible 50-letter poems.

So, instead of enumerating a mapping between all possible poems and the natural numbers, we need a process for converting any poem to a unique number that identifies that poem. Suppose we write our poem using the English alphabet. If we include lower-case letters (26), upper-case letters (26), and punctuation (space, comma, period, newline, semi-colon), we have 57 different discrete symbols to represent. We can assign a unique number to each symbol, and encode the corresponding number with six bits (this leaves seven values unused since six bits is enough to distinguish 64 values). For example, we could encode the alphabet as:

a	**000000**
b	**000001**
c	**000010**
d	**000011**
...	...
p	**001111**
q	**010000**
...	...
z	**011001**
A	**011010**
...	...
F	**011111**

G	**100000**
H	**100001**
...	...
Z	**110011**
space	**110100**
,	**110101**
.	**110110**
newline	**110111**
;	**111000**
unused	**111001**
...	...
unused	**111111**

Table 1.1. Encoding characters using bits.
This encoding is not the one typically used by computers. One commonly used encoding known as ASCII (the American Standard Code for Information Interchange) uses seven bits so that 128 different symbols can be encoded. The extra symbols are used to encode more special characters.

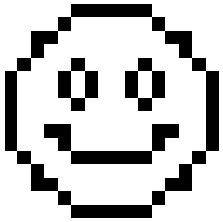
With the encoding in Table 1.1, the first bit answers the question, "Is it an uppercase letter after **F** or a special character?". When the first bit is **0**, the second bit answers the question, "Is it after **p**?".

Once we have a way of mapping each individual letter to a fixed-length bit sequence, we could write down any poem by just concatenating the bits encoding each letter. So, a poem that begins "The" would be encoded as **101101 000111 000100** (the spaces are used for clarity to separate the letters, but are not part of the actual encoding since we know each group of six bits encodes one letter in this encoding). We could write down any poem of length n that is written in the 57-symbol alphabet using this encoding using $6n$ bits. To convert the encoded poem back into English letters, we just need to invert the mapping.

How much information is really in **101101 000111 000100**? It contains 18 bits, so it encodes at most 18 bits of information. But, this is the *maximum* amount of information it could contain. The actual amount of information it encodes, however, is much less. This sequence would contain 18 bits of information only if all sequences of 18 bits are equally likely. In fact, if we already know it is the start of a poem written in English, it encodes far less information. For example, the Wordsworth's complete poetical works contains 887 poems, of which 97 poems start with "The" (that is, the first three letters, so this includes poems like *A Complaint* which starts, "There is a change—and I am poor"). The trigram "The" is the most common starting trigram for Wordsworth's poems, followed by "Whe" (28 poems) and "Wha" (20 poems). Hence, the probability that a poem starts with

"The" is approximately 10% (if Wordsworth is representative of English-language poets), where the probability if all 18-bit sequences were equally likely would be one in $2^{18} = 262,144$. Hence, learning that the poem starts with "The" conveys much less than 18 bits of information, even if our inefficient encoding uses 18 bits to encode the first three letters of the poem.

Rich Data. We can use bit sequences to represent complex data like pictures, movies, and audio recordings too. First, let's consider a simple black and white picture:



Since the picture is divided into discrete squares (known as *pixel*s), we could encode this as a sequence of bits by using one bit to encode the color of each pixel (for example, using **1** to represent black, and **0** to represent white). This image is 16x16, so has 256 pixels total. We could represent the image using a sequence of 256 bits (starting from the top left corner):

0000011111100000
0000100000010000
0011000000001100
0010000000000100
...

What about complex pictures that are not divided into discrete squares or a fixed number of colors, like Van Gogh's *Starry Night*?

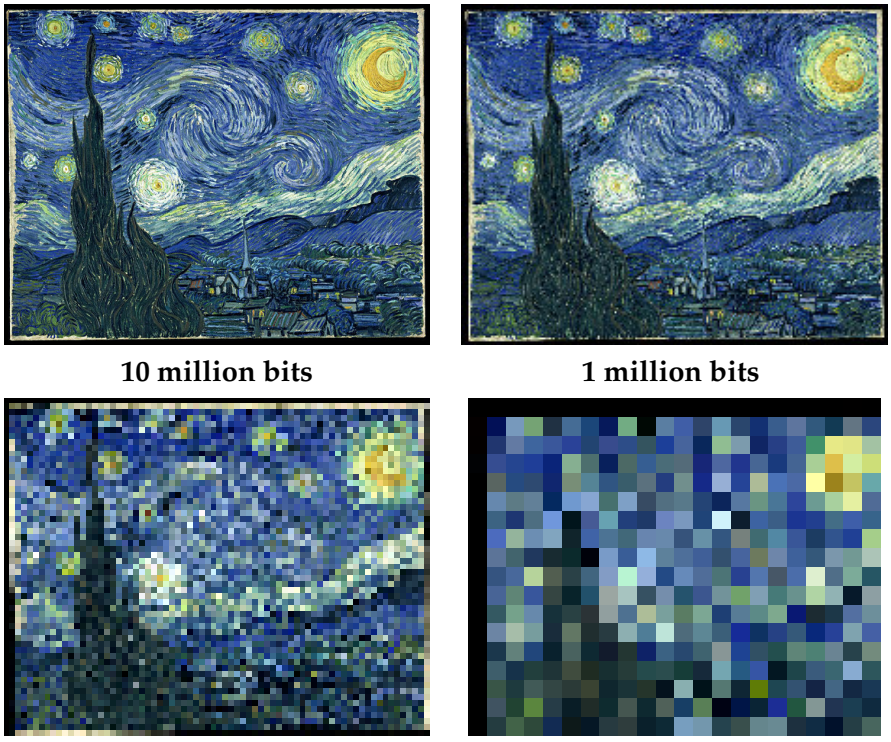
Different wavelengths of electromagnetic radiation have different colors. There are arguably[1] infinitely many different colors, corresponding to different wavelengths of visible light. For example, light with wavelengths between 625 and 730 nanometers appears red. But, each wavelength of light has a slightly different color; for example, light with wavelength 650 nanometers would be a different color (albeit imperceptible to humans) from light of wavelength 650.0000001 nanometers. So, if there are infinitely many colors, there is no way to map each color to a unique, finite bit sequence, and there is no way to completely describe a picture with a finite bit sequence.

On the other hand, the human eye and brain have limits. We cannot actually perceive infinitely many different colors; at some point the wavelengths are close enough that we cannot distinguish them. Ability to distinguish colors varies, but most humans can perceive several million different colors. Then, we can map each distinguishable color to a unique bit sequence. One way to represent color is to break it into its three primary components (red, green, and blue), and record the intensity of each component. The more bits available to represent a color, the more different colors that can be represented.

So, with bits we can approximate the color at each point. How many different points are there? If space in the universe is continuous, there are infinitely many points. But, as with color, once the points get smaller than a certain size they are imperceptible. We can approximate the picture by dividing the canvas into small regions and sampling the average color of

[1] This comes down to the question of whether the space-time of the universe is really continuous or discrete. Certainly in our common perception it seems to be continuous—we can imagine dividing a time interval in two. In reality, this may not be the case, but if the universe is discrete it is discrete at an extremely tiny scale (e.g., less than 10^{-40} of a second).

each region. The smaller the sample regions, the more bits we will have and the more detail that will be visible in the image. With enough bits to represent color, and enough sample points, we can represent any image as a sequence of bits.



10 million bits **1 million bits**

100 000 bits **10 000 bits**

Figure 1.2. Information and image quality.

The first Van Gogh image uses 10 million bits. Figure 1.2 shows four versions of the Van Gogh painting, each using a different number of bits to encode the image. For these images, we use 24-bit color so colors are represented by three 8-bit values, one each for the red, green, and blue intensities. This means 2^{24} (over 16 million) different colors that can be represented.

As with the poems, however, there is much less information in the pictures than the raw number of bits. For the first picture in Figure 1.2, there are approximately 10 million bits—the image is 722x576 pixels, so there are 415 872 pixels; with 24-bits for each pixel, there are 9 980 928 bits (of course, depending on the resolution of your printer or screen, you may not be seeing all the pixels). But, each picture bit does not provide one full bit of information. For example, the color of a pixel is often similar to the color of the pixels next to it. This is not always true, of course, but it is true most of

the time. Hence, without learning the color value for this pixel we have a much better that one in 2^{24} chance of guessing this pixel's color. Thus, the actual amount of information we learn from the 24 bits of color information for this pixel are much less than 24 bits.

When images are stored on computers they use *compression* to reduce the number of bits needed to store the image. Compression algorithms can map the 1 million bits in the first image to a smaller number of bits. When the image is viewed, a decompression algorithm performs the inverse mapping, producing the full image from the compressed data. For the first image, one of the most commonly used image compression algorithms (JPEG), reduces the size of the 10 million bit image to just over 1 million bits. JPEG is a *lossy* compression algorithm, meaning that when the compressed image is decompressed it may not produce exactly the original image, but rather a good approximation of it.

compression

Summary and Preview. We can use sequences of bits to represent any natural number exactly, and hence, represent any member of a finite set of values using a sequence of bits. The more bits we use the more different values that can be represented; with n bits we can represent 2^n different values. We can also use sequences of bits to represent rich data like images, audio, and video. Since the world we are trying to represent is continuous there are infinitely many possible values, and we cannot represent these objects exactly with any finite sequence of bits; but, since human perception is limited, with enough bits we can represent any of these adequately well. Finding good ways to represent data is a constant challenge in computing. Manipulating sequences of bits is awkward, so we need ways of thinking about sequences of bits at higher levels of abstraction. Chapter 5 focuses on ways to manage complex data.

1.2.3 Growth of Computing Power

The number of bits a computer can store gives an upper limit on the amount of information it can process. Looking at the number of bits different computers can store over time gives us a rough indication of how computing power has increased. Here, we consider two machines: the Apollo Guidance Computer and a modern laptop.

Apollo Guidance Computer

The Apollo Guidance Computer was developed in the early 1960s to control the flight systems of the Apollo. In some respects, it can be considered the first *personal computer*, since it was designed to be used in real-time by a single operator (an astronaut in the Apollo capsule). Most earlier computers required a full room, and were far too expensive to be devoted to a single user; instead, they processed jobs submitted by many users in turn.

AGC User Interface

Since the Apollo Guidance Computer was designed to fit in the Apollo capsule, it needed to be small and light. Its volume was about a cubic foot and it weighed 70 pounds. The AGC was the first computer built using integrated circuits, miniature electronic circuits that can perform simple logical operations. The AGC used about 4000 integrated circuits, each one being able to perform a single logical operation and costing $1000. The AGC consumed a significant fraction of all integrated circuits produced in the mid-1960s, and the project spurred the growth of the integrated circuit industry.

The AGC had 552 960 bits of memory (of which only 61 440 bits were modifiable, the rest were fixed). (Compare the roughly half a million bits needed to perform rendezvous in space with the van Gogh pictures.) The smallest USB flash memory you can buy today (from SanDisk in December 2008) is the 1 gigabyte Cruzer for $9.99; 1 gigabyte (GB) is 2^{30} bytes or approximately 8.6 billion bits, about 140 000 times the amount of memory in the AGC (and all of the Cruzer memory is modifiable). A typical low-end laptop today has 2 gigabytes of RAM (fast memory close to the processor that loses its state when the machine is turned off) and 250 gigabytes of hard disk memory (slow memory that persists when the machine is turned off); for under $600 today we get a computer with over 4 million times the amount of memory the AGC had.

Moore's law is a violation of Murphy's law. Everything gets better and better.
Gordon Moore

To improve by a factor of 4 million involves doubling 23 times in about 46 years. So, the amount of computing power approximately doubled every two years between the AGC in the early 1960s and a modern laptop today (2009). This property of exponential improvement in computing power is known as *Moore's Law*. One of the pioneers in integrated circuit technology, Gordon Moore, who co-founded Intel, observed in 1965 than the number of components that can be built in integrated circuits for the same cost was approximately doubling every year (various revisions to Moore's observation have put the doubling rate at approximately 18 months instead of one year). This growth has been driven by the growth of the computing industry, increasing the resources available for designing integrated circuits. Another driver is that the current technology can be used to design the next technology generation. Improvement in computing power has followed this exponential growth remarkably closely over the past 40 years, although there is no law, of course, that this growth can continue forever.

Although our comparison between the AGC and a modern laptop shows an impressive factor of 4 million improvement, it is much slower than Moore's law would suggest. Instead of 23 doublings in power since 1963, there should have been 30 doublings (using the 18 month doubling rate). This would produce an improvement of one billion times instead of just 4 million. The reason is our comparison is very unequal relative to cost: the

AGC was the world's most expensive small computer of its time, reflecting many millions of dollars of government funding. Computing power available for similar funding today is well over a billion times more powerful than the AGC.

1.3 Science, Engineering, and Liberal Art

Much ink and many bits have been spent on debating whether computer science is an art, an engineering discipline, or a science. The confusion stems from the nature of computing as a new field that does not fit well into existing silos. In fact, computer science fits well into all three categories, and it is useful to approach computing from all three perspectives.

Science. Traditional science is about understanding nature through observation. The goal of science is to develop general and predictive theories—that is, theories that allow us to understand aspects of nature deeply enough to make accurate quantitative predications about them. For example, Newton's law of universal gravitation makes predictions about how masses will move. The more general a theory is the better; a key, as yet unachieved, goal of science is to find a universal law that can describe all physical behavior at scales from the smallest particle to the largest galaxy clusters, and all the bosons, muons, dark matter, and black holes in between. Science deals with real things (like bowling balls, planets, and electrons) and attempts to make progress towards theories that predict how these real things will behave in different situations.

Computer science typically focuses on artificial things like numbers, graphs, functions, and lists. Instead of dealing with physical things in the real world, most of computer science concerns abstract things in a virtual world. The numbers we use in computations, of course, can represent properties of physical things in the real world, and with enough bits we can model real things with arbitrary precision. But, since our focus is on the abstract, artificial things rather than the real, physical things, computer science is traditionally not a natural science but a more abstract field like mathematics. Like mathematics, computing is an essential tool for modern science, but when we study computing on artificial things it is not a natural science itself.

On the other hand, computing exists all over nature. A long term goal of computer science is to develop theories that allow us to better understand how nature computes. One direct example of computing in nature comes from biology. Complex life exists because nature can perform sophisticated computing. Sometimes people analogize DNA as a "blueprint", but it is

really much better thought of as a program. Whereas a blueprint describes what a building should be when it is finished, giving the dimensions of walls and how they fit together, the DNA of an organism needs to encode a process for making that organism. This can be seen most clearly by looking at embryology. In the steps from a single cell to a human baby, the embryo goes through forms that seem to resemble our evolutionary ancestors (see Neil Shubin's *Your Inner Fish* for a wonderful exposition on this).

Surely this is not the most sensible "make-a-human" process if one were designing it from scratch. Thus, the human genome is not a blueprint that describes the body plan of a human, it is a program that turns a single cell into a complex human. The process of evolution (which itself is an information process) produces new programs, and hence new species, through the process of natural selection on mutated DNA sequences. Understanding how both these processes work is one of the most interesting and important open scientific questions, and it involves deep questions in computer science, as well as biology, chemistry, and physics.

The scientific questions we consider in this book focus on the question of what can and cannot be computed. This is both a theoretical question (what can be computed by a given theoretical model of a computer), and a pragmatic one (what can be computed by physical things in our universe). Answering the second question requires deeper understanding of our physical universe (in particular, quantum physics) than is currently known, as well as advances in computer science.

Scientists study the world as it is; engineers create the world that never has been.
Theodore von Kármán

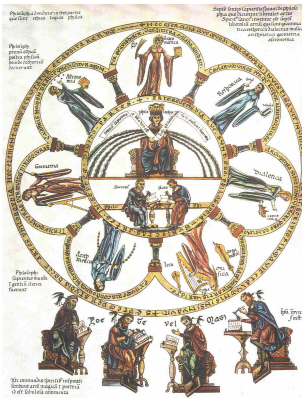
Engineering. Engineering is about making useful things. Engineering is often distinguished from crafts in that engineers use scientific principles to create their designs, and focus on designing under practical constraints. As William Wulf (University of Virginia professor and President of the National Academy of Engineering) and George Fisher (CEO of Kodak) wrote:

> *Whereas science is analytic in that it strives to understand nature, or what is, engineering is synthetic in that it strives to create. Our own favorite description of what engineers do is "design under constraint". Engineering is creativity constrained by nature, by cost, by concerns of safety, environmental impact, ergonomics, reliability, manufacturability, maintainability–the whole long list of such "ilities". To be sure, the realities of nature is one of the constraint sets we work under, but it is far from the only one, it is seldom the hardest one, and almost never the limiting one.* [WF02]

Computer scientists do not typically face the main constraints faced by civil and mechanical engineers—computer programs are massless, odorless, and tasteless, so the kinds of physical constraints like gravity and ten-

sile strength that impose limits on bridge designs are not relevant to most computer scientists. As we saw from the Apollo Guidance Computer comparison, what practical constraints there are on computing power change rapidly — the one billion times improvement in computing power is unlike any change in physical materials (for example, the highest strength density material available today, carbon nanotubes, are perhaps 300 times stronger than the best material available 50 years ago). Although we may need to worry about manufacturability and maintainability of storage media (such as the disk we use to store a program), our primary focus as computer scientists is on the abstract bits themselves, not how they are stored.

There are, however, many constraints that computer scientists face. A primary constraint is the capacity of the human mind—there is a limit to how much information a human can keep in mind at one time. As computing systems get more complex, there is no way for a human to understand the entire system at once. To build complex systems, we need techniques for managing complexity. The primary tool computer scientists use to manage complexity is *abstraction*. Abstraction is a way of giving a name to something in a way that allows us to hide unnecessary details. By using carefully designed abstractions, we can construct complex systems with reliable properties while limiting the amount of information a human designer needs to keep in mind at any one time.



Herrad von Landsberg, ***Septem Artes Liberales***

Liberal Art. The notion of the *liberal arts* emerged during the middle ages to distinguish education for the purpose of expanding the intellects of free people from the *illiberal arts* such as medicine and carpentry that were pursued for economic purposes. The liberal arts were intended for people who did not need to learn an art to make a living, but instead had the luxury to pursue purely intellectual activities for their own sake. The traditional seven liberal arts started with the *Trivium* (three roads), focused on language (the quotes defining each liberal art are from Sister Miriam Joseph [Jos02])

- Grammar — "the art of inventing symbols and combining them to express thought"
- Rhetoric — "the art of communicating thought from one mind to another, the adaptation of language to circumstance"
- Logic — "the art of thinking"

The Trivium was followed by the *Quadrivium* (four roads), focused on numbers:

- Arithmetic — "theory of number"
- Geometry — "theory of space"

- Music — "application of the theory of number"
- Astronomy — "application of the theory of space"

All of these have strong connections to computer science, and we will touch on each of them to some degree in this book. Language is essential to computing since we use the tools of language to describe information processes. In Chapter 2, we investigate the structure of language and throughout this book we consider how to efficiently use and combine symbols to express meanings. Rhetoric encompasses communicating thoughts between minds. In computing, we are not typically communicating between minds, but we see many forms of communication between entities — interfaces between components of a program, as well as protocols used to enable multiple computing systems to communicate (for example, the HTTP protocol defines how a web browser and web server interact), and communication between computer programs and humans through their user interfaces. The primary tool for understanding what computer programs mean, and hence, for constructing programs with particular meanings, is logic. Hence, the traditional trivium liberal arts of language and logic permeate computer science.

The connections between computing and the quadrivium arts are also pervasive. We have already seen how computing uses sequences of bits to represent numbers, and computing requires fast algorithms for performing arithmetic. In Chapter 17, we will see how numbers can be represented using even more fundamental building blocks and arithmetical primitives can be defined starting from scratch. Geometry is essential for computer graphics, and graph theory is also important for computer networking. The harmonic structures in music have strong connections to the recursive definitions we will see in Chapter 4 and later chapters (see Hofstadter's *Gödel, Escher, Bach* for lots of interesting examples of connections between computing and music). Unlike the other six liberal arts, astronomy is not directly connected to computing, but computing is an essential tool for doing modern astronomy.

Hence, although learning about computing qualifies as an illiberal art (that is, it has substantial economic benefits for those who learn it well), computer science also covers at least six of the traditional seven liberal arts.

1.4 Summary and Roadmap

Computer scientists think about problems differently. When confronted with a problem, a computer scientist does not just attempt to solve it. Instead, computer scientists think about a problem as a mapping between

its inputs and desired outputs, develop a systematic sequence of steps for solving the problem for any possible input, and consider how the number of steps required to solve the problem scales as the input size increases.

The nature of the computer forces solutions to be expressed precisely in a language the computer can interpret. This means a computer scientist needs to understand how languages work, and exactly what they mean. The next chapter focuses on language. Natural languages like English are too complex and inexact for this, so we need to invent and use new languages that are simpler, more structured, and less ambiguously defined than natural languages.

The computer frees the human from having to actually carry out the steps needed to solve the problem. Without complaint, boredom, or rebellion, it dutifully executes the exact steps the program specifies. And it executes them at a remarkable rate - billions of simple steps in each second on a typical laptop. This changes not just the time it takes to solve a problem, but qualitatively changes the kinds of problems we can solve, and the kinds of solutions worth considering. Problems like sequencing the human genome, simulating the global climate, and making a photomosaic not only could not have been solved without computing, but perhaps could also not have even been envisioned.

The rest of this book presents a whirlwind introduction to computer science. We do not cover any topics in great depth, but rather provide a broad picture of what computer science is, how to think like a computer scientist, and how to solve problems. Much of the book will revolve around three very powerful ideas that are prevalent throughout computing:

Recursive definitions. A recursive definition defines something in terms of smaller instances of itself. This is a powerful way of solving problems by breaking a problem into solving a simple instance of the problem, and showing how to solve a larger instance of the problem by using a solution to a smaller instance. We introduce recursive definitions in how they are used to define infinite languages in Chapter 2, show how they can be used to solve problems in Chapter 4, illustrate how complex data structures can be built from recursive definitions in Chapter 5, and in later chapters see how language interpreters themselves can be defined recursively.

Higher order procedures. Computers are distinguished from other machines in that their behavior can be changed by a program. Programs themselves are just bits though, so we can write programs to generate other programs. With higher order procedures, we treat procedures just like any other data. Procedures can be passed as inputs and generated as outputs. Considering procedures as data is both a powerful problem solving tool, and a useful way of thinking about the power and fundamental limits of computing.

We introduce the use of procedures as inputs and outputs in Chapter 4, see how generated procedures can be packaged with state to model objects in Chapter 12, and use procedures at data to reason about the fundamental limits of computing in Chapter 13.

Abstraction. Abstraction is a way of hiding details by giving things names. We use abstraction to manage complexity and to enable things to be reused. Good abstractions hide unnecessary details so they can be used to build complex systems without needing to understand all the details of the abstraction at once. We introduce procedural abstraction in Chapter 4, data abstraction in Chapter 5, abstraction using objects in Chapter 12, and many other examples of abstraction throughout this book.

These themes will recur throughout the rest of the book, as we explore the art and science of how to instruct computing machines to perform useful tasks, reasoning about the resources needed to execute a particular procedure, and understanding the fundamental and practical limits on what computers can do.

2

Language

"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean - nothing more nor less."

"The question is," said Alice, "whether you can make words mean so many different things."

Lewis Carroll, *Through the Looking Glass*

The most powerful tool we have for communication is language. This is true whether we are considering communication between two humans, communication between a human programmer and a computer, or communication between multiple computers. In computing, we use language to describe procedures and use tools to turn descriptions of procedures in language that are easy for humans to read and write into executing processes. This chapter considers what a language is, how language works, and introduces the techniques we will use to define languages.

2.1 Surface Forms and Meanings

A *language* is a set of surface forms, s, meanings, m, and a mapping between the surface forms in s and their associated meanings.[1] In the earliest human languages, the surface forms were sounds. But, the surface forms can be anything that can be perceived by the communicating parties. We will focus on languages where the surface forms are text.

A *natural language* is a language spoken by humans, such as English. Natural languages are very complex since they have evolved over many thousands years of individual and cultural interaction. We will be primarily concerned with designed languages that are created by humans for some deliberate purpose (in particular, languages created for expressing procedures to be executed by computers).

A simple communication system could be described by just listing a table of surface forms and their associated meanings. For example, this table

[1] Thanks to Charles Yang for this definition.

describes a communication system between traffic lights and drivers:

Surface Form	Meaning
Green	Go
Yellow	Caution
Red	Stop

Rotary traffic signal

Communication systems involving humans are notoriously imprecise and subjective. A driver and a police officer may disagree on the actual meaning of the *Yellow* symbol, and may even disagree on which symbol is being transmitted by the traffic light at a particular time. Communication systems for computers demand precision: we want to know what our programs will do, so it is important that every step they make is understood precisely and unambiguously. The method of defining a communication system by listing a table of

$$< Symbol, Meaning >$$

pairs can work adequately only for trivial communication systems. The number of possible meanings that can be expressed is limited by the number of entries in the table. It is impossible to express any *new* meaning using the communication system: all meanings must already be listed in the table!

Languages and Infinity

A real language must be able to express *infinitely* many different meanings. This means it must provide infinitely many surface forms; hence, there must be a system for generating new surface forms and a way of inferring the meaning of each generated surface form. No finite representation such as a printed table can contain all the surface forms and meanings in an infinite language.

One way humans can generate infinitely large sets is to use repeating patterns. For example, most humans would recognize the notation:

1, 2, 3, . . .

as the set of all natural numbers. We interpret the "..." as meaning keep doing the same thing for ever. In this case, it means keep adding one to the preceding number. Thus, with only a few numbers and symbols we can describe a set containing infinitely many numbers.

The repeating pattern technique might be sufficient to describe some languages with infinitely many meanings. For example, this table defines an infinite language:

Surface Form	Meaning
I will run today.	Today, I will run.
I will run the day after today.	One day after today, I will run.
I will run the day after the day after today.	Two days after today, I will run.
...	...

Although we can describe some infinite languages in this way, it is entirely unsatisfactory.[2] The set of surface forms must be produced by simple repetition. Although we can express new meanings using this type of language (for example, we can always add one more "the day after" to the longest previously produced surface form), the new surface forms and associated meanings are very similar to previously known ones.

2.2 Language Construction

To define more expressive infinite languages, we need a richer system for constructing new surface forms and associated meanings. We need ways of describing languages that allow us to describe an infinitely large set of surface forms and meanings with a compact notation. The approach we will use is to define a language by defining a set of rules that produce all strings in the language (and no strings that are not in the language).

Components of Language

A language is composed of:

- *primitives* — the smallest units of meaning. A primitive cannot be broken into smaller parts that have relevant meanings.
- *means of combination* — rules for building new language elements by combining simpler ones.

In English, the primitives are the smallest meaningful units, known as *morphemes*. The means of combination are rules for building words from morphemes, and for building phrases and sentences from words.

Since we have rules for producing new words not all words are primitives. For example, we can create a new word by adding *anti-* in front of an ex-

[2] Languages that can be defined using simple repeating patterns in this way are known as *regular languages*. We will define this more precisely in Chapter 17, and see that they are actually a bit more interesting than it seems here.

isting word. The meaning of the new word is (approximately) "against the meaning of the original word".

For example, the verb *freeze* means to pass from a liquid state to a solid state; *antifreeze* is a substance designed to prevent freezing. An English speaker who knew the meaning of *freeze* and *anti-* could roughly guess the meaning of *antifreeze* even if she has never heard the word before.[3]

Note that the primitives are defined as the smallest units of *meaning*, not based on the surface forms. Both *anti* and *freeze* are morphemes; they cannot be broken into smaller parts with meaning. We can break *anti-* into two syllables, or four letters, but those sub-components do not have meanings that could be combined to produce the meaning of the morpheme.

This property of English means anyone can invent a new word, and use it in communication in ways that will probably be understood by listeners who have never heard the word before. There can be no longest English word, since for whatever word you claim to be the longest, I can create a longer one (for example, by adding *anti-* to the beginning of your word).

Means of Abstraction

In addition to primitives and means of combination, powerful languages have an additional type of component that enables economic communication: *means of abstraction*.

Means of abstraction allow us to give a simple name to a complex entity. In English, the means of abstraction are *pronouns* like "she", "it", and "they". The meaning of a pronoun depends on the context in which it is used. It abstracts a complex meaning with a simple word. For example, the *it* in the previous sentence abstracts "the meaning of a pronoun", but the *it* in the sentence before that one abstracts "a pronoun". In natural languages, means of abstraction tend to be awkward (English has *she* and *he*, but no gender-neutral pronoun for abstracting a person), and confusing (it is often unclear what a particular *it* is abstracting). Languages for programming computers need to have powerful and clear means of abstraction.

The next three sections introduce three different ways to define languages. The first system, *production systems*, is very powerful[4] but not widely used for defining languages today because it is too difficult (meaning it can be impossible) to determine if a given string is in the language. The next two, *recursive transition networks* and *replacement grammars* are less powerful than production systems, but more useful because they are simpler to reason about.

[3] Guessing that it is a verb meaning to pass from the solid to liquid state would also be reasonable. This shows how imprecise and ambiguous natural languages are; for programming computers, we need the meanings of constructs to be clearly determined.

[4] In fact, it is exactly as powerful as a Turing machine (see Chapter 13).

We focus on languages where the surface forms can easily be written down and interpreted linearly. This means the surface forms will be sequences of characters. A character is a symbol selected from a finite set of symbols known as an *alphabet*. A typical alphabet comprises the letters, numerals, and punctuation symbols used in English. We will refer to a sequence of zero or more characters as a *string*. Hence, our goal in defining the surface forms of a textual language is to define the set of strings in the language. To define a language, we need to define a system that produces all strings in the language, and no other strings. The problem of associating meanings with those strings is much more difficult; we consider it in various ways in later chapters.

Exercise 2.1. [⋆] Merriam-Webster's word for the year for 2006 was *truthiness*, a word invented and popularized by Stephen Colbert. Its definition is, "truth that comes from the gut, not books". Identify the morphemes that are used to build *truthiness*, and explain, based on its composition, what *truthiness* should mean.

Exercise 2.2. According to the Guinness Book of World Records, the longest word in the English language is *floccinaucinihilipilification*, meaning "The act or habit of describing or regarding something as worthless".

a. [⋆] Break *floccinaucinihilipilification* into its morphemes. Show that a speaker familiar with the morphemes could understand the word.

b. [⋆] Prove Guinness wrong by demonstrating a longer English word. An English speaker (familiar with the morphemes) should be able to deduce the meaning of your word.

Excursion 2.1: Wordsmithing. Embiggening your vocabulary with anticromulent words that ecdysiasts can grok.

a. [⋆] Invent a new English word by combining common morphemes.

b. [⋆] Get someone else to use the word you invented.

c. [⋆⋆⋆] Get Merriam-Webster to add your word to their dictionary.

2.3 Production Systems

Production systems were invented by Emil Post, an American logician in the 1920's. A *Post production system* consists of a set of production rules. Each production rule consists of a pattern to match on the left side, and a replacement on the right side. From an initial string, rules that match are

applied to produce new strings.

For example, Douglas Hofstadter describes the following Post production system known as the *MIU-system* in *Gödel, Escher, Bach* (each rule is described first formally, and the quoted text below is paraphrased from the description from *GEB*):

As for any claims I might make perhaps the best I can say is that I would have proved *Gödel's Theorem in 1921 — had I been Gödel.*
Emil Post, from postcard to Gödel, October 1938.

Rule I: $x\mathsf{I} ::\Rightarrow x\mathsf{IU}$
"If you possess a string whose last letter is I, you may produce the string with U added at the end."
Rule II: $\mathsf{M}x ::\Rightarrow \mathsf{M}xx$
"If you have Mx, you may produce Mxx."
Rule III: $x\mathsf{III}y ::\Rightarrow x\mathsf{U}y$
"If III occurs in one of the strings in your collection, you may produce a new string with U in place of III."
Rule IV: $x\mathsf{UU}y ::\Rightarrow xy$
"If UU occurs inside your string, you can produce a string with it removed."

The rules use the variables x and y to match any sequence of symbols. On the left side of a rule, x means match a sequence of zero or more symbols. On the right side of a rule, x means produce whatever x matched on the left side of the rule. We refer to this process as *binding*. The variable x is initially unbound — it may match any sequence of symbols. Once it is matched, though, it is *bound* and refers to the sequence of symbols that were matched.

For example, consider applying Rule II to MUM. To match Rule II, the first M matches the M at the beginning of the left side of the rule. After that the rule uses x, which is currently unbound. We can bind x to UM to match the rule. The right side of the rule produces Mxx. Since x is bound to UM, the result is MUMUM.

Given these four rules, we can start from a given string and apply the rules to produce new strings. For example, starting from MI we can apply the rules to produce MUIUIU:

1.	MI	Initial String
2.	MII	Apply Rule II with x bound to I
3.	MIIII	Apply Rule II with x bound to II
4.	MIIIIIIII	Apply Rule II with x bound to IIII
5.	MUIIIII	Apply Rule III with x bound to M and y bound to IIII
6.	MUIIIIIU	Apply Rule I with x bound to MUIIII
7.	MUIUIU	Apply Rule III with x bound to MUI and y bound to IU

Note that at some steps we have many choices about which rule to apply, and what bindings to use when we apply the rule. For example, at step 5 we could have instead bound x to MUII and y to the empty string to produce MUIIU.

Exercise 2.3. *MIU-system* productions.

a. [⋆] Using the *MIU-system*, show how M can be derived starting from MI?

b. [⋆] Using the *MIU-system*, how many different strings can be derived starting from UMI?

c. [⋆] Using the *MIU-system*, how many different strings can be derived starting from MI?

d. [⋆] (Based on *GEB*) Using the *MIU-system*, is it possible to produce MU starting from MI?

Exercise 2.4. [⋆] Devise a Post production system that can produce all the surface forms in the { "I will run today.", "I will run the day after today.", "I will run the day after the day after today.", . . . } language.

2.4 Recursive Transition Networks

Although Post production systems are powerful enough to generate complex languages, they are more awkward to use than we would like. In particular, applying a rule requires making decisions about binding variables in the left side of a rule. This makes it hard to reason about the strings in a language, and hard to determine whether or not a given string is in the language (see Exercise 3). *Recursive transition networks* (RTNs) are a less powerful[5] way of defining a language, but provide a clearer way of understanding the set of strings in a defined language.

A recursive transition network is defined by a graph of nodes and edges. The edges are labeled with output symbols. One of the nodes is designated the start node (indicated by an arrow pointing into that node). One or more of the nodes may be designated as final nodes (indicated by an inner circle). A string is in the language if there is a path from the start node to a final node in the graph such that when all the output symbols on the path edges are collected along the path they form the string.

[5] We mean less powerful in a formal sense: there are languages that can be described by a Post production system that cannot be defined by any recursive transition network. Section 2.6 discusses the relative power of different language definition mechanisms.

For example, Figure 2.1 shows a simple recursive transition network with three nodes and four edges.

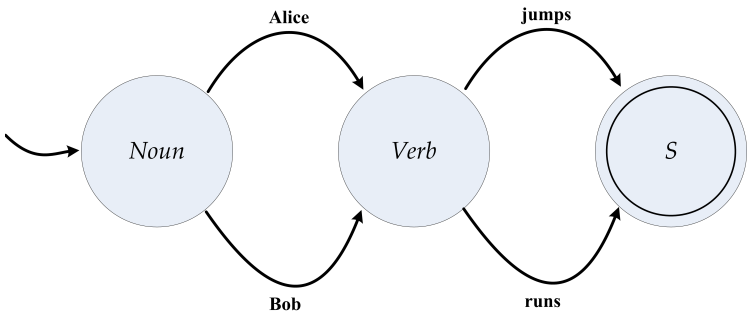


Figure 2.1. Simple recursive transition network.

This network can produce four different sentences. Starting in the node marked *Noun*, we have two possible edges to follow; each edge outputs a different symbol, and leads to the node marked *Verb*. From that node, we have two possible edges, each leading to the node marked *S*, which is a final node. Since there are no edges out of *S*, this ends the string. Hence, we can produce four strings corresponding to the four different paths from the start to final node: "Alice jumps", "Alice runs", "Bob jumps", and "Bob runs".

This way of defining a language is more efficient than just listing all strings in a table, since the number of possible strings increases with the number of possible paths in the graph. For example, adding one more edge from *Noun* to *Verb* with label "Colleen", would add two new strings to the language.

The expressive power of recursive transition networks really increases once we add edges that form cycles in the graph. This is where the *recursive* in the name comes from. Once a graph has a cycle, there are *infinitely* many possible paths through the graph, since we can always go around the cycle one more time. Consider what happens when we add a single edge to the previous network:

Now, we can produce infinitely many different strings! We can follow the "and" edge back to the *Noun* node, to produce strings like "Alice runs and Bob jumps and Alice jumps and Alice runs" with as many conjuncts as we want.

Exercise 2.5. [⋆] Draw a recursive transition network that defines the language of the whole numbers: 0, 1, 2, ...

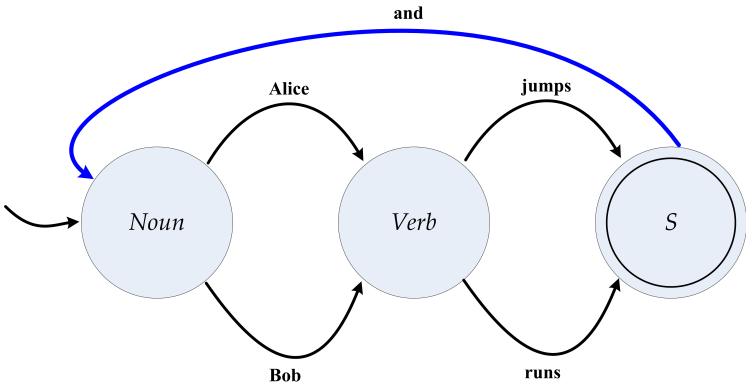


Figure 2.2. RTN with a cycle.

Exercise 2.6. Recursive transition networks.

a. [⋆] What is the smallest number of edges needed for a recursive transition network that can produce exactly 8 strings?

b. [⋆] What is the smallest number of nodes needed for a recursive transition network that can produce exactly 8 strings?

c. [⋆] What is the smallest number of edges needed for a recursive transition network that can produce exactly n strings?

Exercise 2.7. [⋆⋆] Is it possible to define the language of the *MIU-system* using a recursive transition network? Either draw a network that matches the language produced by the *MIU-system*, or explain what it is impossible to do so.

2.4.1 Subnetworks

In the RTNs we have seen so far, the labels on the output edges are direct outputs known as *terminals*: following an edge just produces the symbol on that edge. We can make more expressive RTNs by allowing edge labels to also name *subnetworks*. A subnetwork is identified by the name of its starting node. When an edge with a subnetwork label is followed, instead of outputting one symbol, the network traversal jumps to the subnetwork node. Then, it can follow any path from that node to a final node. Upon reaching a final node, the network traversal jumps back to complete the edge.

For example, consider the network shown in Figure 2.3. It describes the same language as the RTN in Figure 2.1, but uses subnetworks for *Noun* and *Verb*. To produce a string, we start in the *Sentence* node. The only

edge out from *Sentence* is labeled *Noun*. To follow the edge, we jump to the *Noun* node, which is a separate subnetwork. Now, we can follow any path from *Noun* to a final node (in this cases, outputting either "Alice" or "Bob" on the path toward *EndNoun*.

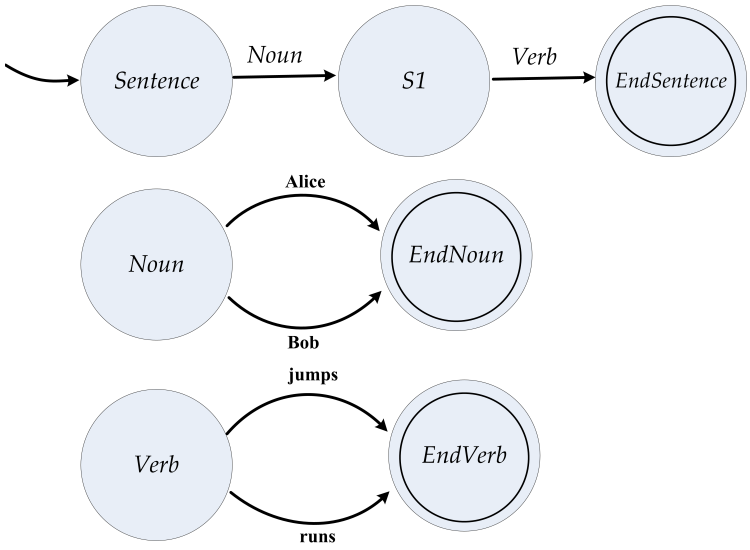


Figure 2.3. Recursive transition network with subnetworks.

Suppose we replace the *Noun* subnetwork with the more interesting version shown in Figure 2.4.

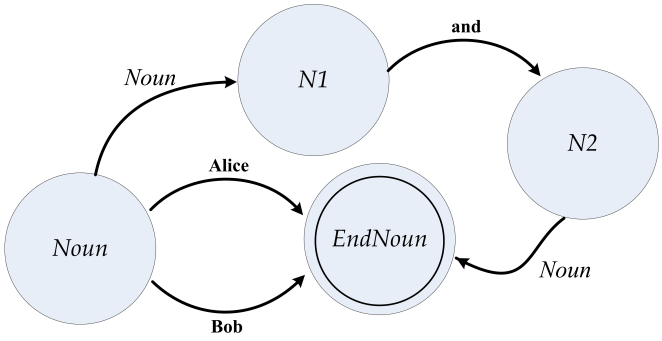


Figure 2.4. Alternate *Noun* subnetwork.

The new subnetwork includes an edge from *Noun* to *N1* labeled with *Noun*. So, if we follow this edge, control jumps back into the *Noun* node, for another path through the *Noun* subnetwork. Starting from *Noun*, we can generate complex phrases like "Alice and Bob" or "Alice and Bob and Alice" (note there are two different paths that generate this phrase).

To keep track of paths through RTNs without subnetworks, a single marker

suffices. We can start with the marker on the start node, and move it along the path through each node to the final node. To keep track of paths on an RTN with subnetworks, though, is more complicated. We need to keep track of the where we are in the current network, but also where we need to jump to when a final node is reached. Since we can enter subnetworks within subnetworks, we need a way to keep track of arbitrarily many jump points.

A data structure for keeping track of this is known as a *stack*. We can think of a stack like a stack of trays in a cafeteria. At any point in time, only the *top* tray on the stack can be reached. We can take the top tray off the stack, after which the next tray is now on top. This operation is called *popping* the stack. We can push a new tray on top of the stack, which makes the old top of the stack now one below the new top. This operation is called *pushing*.

stack

Using a stack, we can follow a path through an RTN using this procedure:[6]

1. Initially, push the starting node on the stack.
2. Pop a node, N, off the stack.
3. If N is a final node, check if the stack is empty. If the stack is empty, **stop**. Otherwise, go back to step 2.
4. Select an edge from the RTN that starts from node N. Use D to represent the destination of that edge, and s to output symbol on the edge.
5. Push D on the stack.
6. If s is a node (that is, the name of a subnetwork), push s on the stack. Otherwise, output s.
7. Go back to step 2.

For the example, we start by pushing *Sentence* on the stack. In step 2, we pop the stack, so the current node, N, is *Sentence*. Since it is not a final node, we do nothing for step 3. In step 4, we choose an edge starting from *Sentence*. There is only one edge to choose, and it leads to the node labeled *S1*. In step 5, we push *S1* on the stack. The label on the edge is *Noun*, which is a node, so we push *Noun* on the stack. The stack now contains two items: $[Noun, S1]$. As directed by step 7, we go back to step 2 and continue by popping the top node, *Noun*, off the stack. It is not a final node, so we continue to step 4, and select the edge from *Noun* to *N1*. Since *N1* is not a final node, we continue to step 5 and push *N1* on the stack. The label on the edge is *Noun*, which is a node, so we push *Noun* on the stack in step 6. At this point, the stack contains three nodes: $[Noun, N1, S1]$.

We continue in the same manner, following the steps in the procedure as we

[6]For simplicity, this procedure assumes we always stop when a final node is reached. We can, however, have RTNs in which there are edges out of final nodes (as in Figure 2.2) and it is possible to either stop or continue from a final node.

keep track of a path through the network. Next, we pop *Noun*, and select the edge labeled "Alice", pushing *EndNoun* on the stack. Returning to step 2, we pop the *EndNoun*, which is a final node. Now, we are at the point to jump back to where we entered the subnetwork. This is the *N1* node, which is now on top of the stack. Continuing in step 2, we pop *N1*, and follow the edge labeled "and", continuing to node *N2*. This leads to another pass through the *Noun* subnetwork, after which we reach the *EndNoun* node. After continuing to step 3, the stack now contains just [*S1*]. In this manner, we can follow a path through the network, using the stack to keep track of the nodes to return to after finishing each subnetwork.

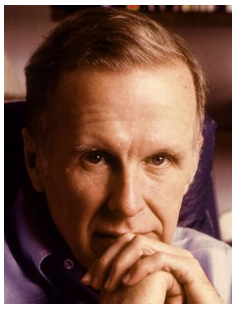
Exercise 2.8. Traversing RTNs.

a. [⋆] Show the sequence of stack values used in generating the string "Alice and Bob and Alice runs".

b. [⋆] Identify a string that cannot be produced with a stack that can hold no more than four elements.

Exercise 2.9. [⋆] The procedure given for traversing RTNs assumes that a subnetwork path always stops when a final node is reached, so cannot follow all possible paths for an RTN where there are edges out of a final node. Describe a procedure that can be used for RTNs where there are edges from final nodes.

2.5 Replacement Grammars



John Backus

Another way to define a language is to use a grammar.[7] This is the most common way languages are defined by computer scientists today, and the way we will use for the rest of this book.

A *grammar* is a set of rules for generating all strings in the language. The grammars we will use are a simple notation known as *Backus-Naur Form* (BNF). BNF was invented by John Backus in the late 1950s. Backus led efforts at IBM to define and implement Fortran, the first widely used high-level programming language. Fortran enabled computer programs to be written in a language more like familiar algebraic formulas than low-level machine instructions, enabling programs to be written more quickly and reliably (in the next chapter, we describe programming languages and how they are implemented). In defining the Fortran language, Backus and his team used ad hoc English descriptions to define the language. Backus de-

[7] You are probably already somewhat familiar with grammars from your time in what was previously known as "grammar school"!

veloped the replacement grammar notation to precisely describe the language of a later programming language, Algol (1958). Peter Naur adapted the notation for the report describing the Algol language, and it was subsequently known as Backus-Naur Form at the suggestion of Donald Knuth to recognize both Backus' and Naur's contributions.

I flunked out every year. I never studied. I hated studying. I was just goofing around. It had the delightful consequence that every year I went to summer school in New Hampshire where I spent the summer sailing and having a nice time.
John Backus

Rules in a Backus-Naur Form grammar are of the form:

$$\textit{nonterminal} ::\Rightarrow \textit{replacement}$$

These rules are similar to Post production rules, except that the left side of a rule is always a single symbol, known as a *nonterminal* since it can never appear in the final generated string. Whenever we can match the nonterminal on the left side of a rule, we can replace it with what appears on the right side of the matching rule. This method of defining languages is exactly as powerful as recursive transition networks (the follow subsection considers why), but easier to write down.

The right side of a rule contains one or more symbols. These symbols may include nonterminals, which will be replaced using replacement rules before generating the final string. They may also be *terminals*, which are symbols that never appear as the left side of a rule. When we describe grammars, we use *italics* to represent nonterminal symbols, and **bold** to represent terminal symbols. Once a terminal is reached, no more replacements can be done on it.

We can generate a string in the language described by a replacement grammar by starting from a designated start symbol (e.g., *sentence*), and at each step selecting a nonterminal in the working string, and replacing it with the right side of a replacement rule whose left side matches the nonterminal. Unlike Post production systems, there are no variables to bind in BNF grammar rules. We simply look for a nonterminal that matches the left side of a rule.

Here is an example BNF grammar:

1. *Sentence* ::⇒ *Noun Verb*
2. *Noun* ::⇒ **Alice**
3. *Noun* ::⇒ **Bob**
4. *Verb* ::⇒ **jumps**
5. *Verb* ::⇒ **runs**

Starting from *Sentence*, we can generate four different sentences using the

replacement rules: "Alice jumps", "Alice runs", "Bob jumps", and "Bob runs".

Derivation

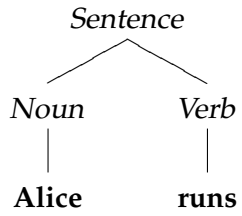
A *derivation* shows how a grammar generates a given string. Here is the derivation of "Alice runs":

Sentence ::⇒ *Noun* *Verb*	using Rule 1
::⇒ **Alice** *Verb*	replacing *Noun* using Rule 2
::⇒ **Alice runs**	replacing *Verb* using Rule 5

Parse Tree

We can represent a grammar derivation as a tree, where the root of the tree is the starting nonterminal (*Sentence* in this case), and the leaves of the tree are the terminals that form the derived sentence. Such a tree is known as a *parse tree*.

Here is the parse tree for the derivation of "Alice runs":



From this example, we can see that BNF notation offers some compression over just listing all strings in the language, since a grammar can have multiple replacement rules for each nonterminal. Adding another rule like,

6. *Noun* ::⇒ **Colleen**

to the grammar would add two new strings ("Colleen runs" and "Colleen jumps") to the language.

Recursive Grammars

The real power of BNF as a compact notation for describing languages, though, comes once we start adding *recursive* rules to our grammar. A grammar is recursive if there is a way to start from a given nonterminal, and follow a sequence of one or more replacement rules to generate a production that contains the same nonterminal.

Suppose we add the rule,

7. *Sentence* ::⇒ *Sentence* **and** *Sentence*

to our example grammar. Now, how many sentences can we generate?

Infinitely many! For example, we can generate "Alice runs and Bob jumps" and "Alice runs and Bob jumps and Colleen runs". We can also generate "Alice runs and Alice runs and Alice runs and Alice runs", with as many repetitions of "Alice runs" as we want. This is very powerful: it means a compact grammar can be used to define a language containing infinitely many strings.

Example 2.1: Whole Numbers. Here is a grammar that defines the language of the whole numbers (0, 1, ...):

$$
\begin{array}{lll}
\textit{Number} & ::\Rightarrow & \textit{Digit MoreDigits} \\
\textit{MoreDigits} & ::\Rightarrow & \\
\textit{MoreDigits} & ::\Rightarrow & \textit{Number} \\
\textit{Digit} & ::\Rightarrow & \mathbf{0} \\
\textit{Digit} & ::\Rightarrow & \mathbf{1} \\
\textit{Digit} & ::\Rightarrow & \mathbf{2} \\
\textit{Digit} & ::\Rightarrow & \mathbf{3} \\
\textit{Digit} & ::\Rightarrow & \mathbf{4} \\
\textit{Digit} & ::\Rightarrow & \mathbf{5} \\
\textit{Digit} & ::\Rightarrow & \mathbf{6} \\
\textit{Digit} & ::\Rightarrow & \mathbf{7} \\
\textit{Digit} & ::\Rightarrow & \mathbf{8} \\
\textit{Digit} & ::\Rightarrow & \mathbf{9}
\end{array}
$$

Note that the second rule says we can replace *MoreDigits* with nothing. This is sometimes written as ϵ to make it clear that the replacement is empty:

$$\textit{MoreDigits} ::\Rightarrow \epsilon$$

This is a very important rule in the grammar—without it *no* strings could be generated; with it *infinitely* many strings can be generated. The key is that we can only produce a string when all nonterminals in the string have been replaced with terminals. Without the $\textit{MoreDigits} ::\Rightarrow \epsilon$ rule, the only rule we would have with *MoreDigits* on the left side is the third rule:

$$\textit{MoreDigits} ::\Rightarrow \textit{Number}$$

The only rule we have with *Number* on the left side is the first rule, which replaces *Number* with *Digit MoreDigits*. Every time we go through this

replacement cycle, we replace *MoreDigits* with *Digit MoreDigits*. We can produce as many *Digit*s as we want, but without the *MoreDigits* ::⇒ ϵ rule we can never stop.

Circular vs. Recursive Definitions

This is the difference between a *circular* definition, and a *recursive* definition. Without the stopping rule, *MoreDigits* would be defined in a circular way. There is no way to start with *MoreDigits* and generate a production that does not contain *MoreDigits* (or a nonterminal that eventually must produce *MoreDigits*). With the *MoreDigits* ::⇒ ϵ rule, however, we have a way to produce something terminal from *MoreDigits*. This is known as a *base case* — a rule that turns an otherwise circular definition into a meaningful, recursive definition.

Figure 2.5 shows a parse tree for the derivation of **150** from *Number*.

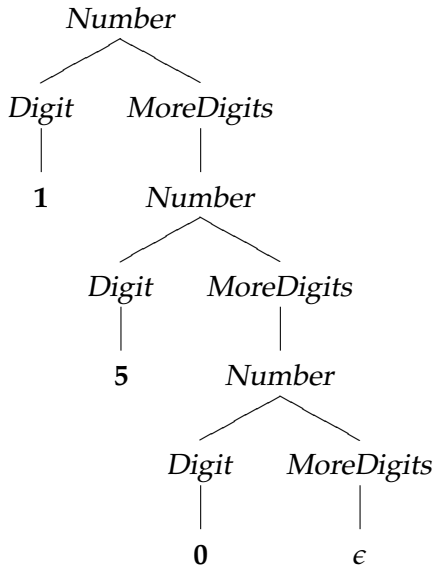


Figure 2.5. Deriviation of 150 from *Number*.

It is common to have many grammar rules with the same left side nonterminal. For example, the whole numbers grammar has ten rules with *Digit* on the left side to produce the ten terminal digits. Each of these is an alternative rule that can be used when the production string contains the nonterminal *Digit*. A compact notation for these types of rules is to use the vertical bar (|) to separate alternative replacements. For example, we could write the ten *Digit* rules compactly as:

Digit ::⇒ **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Exercise 2.10. [⋆] The grammar for whole numbers is complicated because we do not want to include the empty string in our language. Devise a simpler grammar that defines the language of the whole numbers including the empty string.

Exercise 2.11. Suppose we replaced the first rule (*Number* ::⇒ *Digit MoreDigits*) in the whole numbers grammar with this rule:

$$\textit{Number} ::\Rightarrow \textit{MoreDigits Digit}$$

a. [⋆] How does this change the parse tree for the derivation of **150** from *Number*? Draw the parse tree that results from the new grammar.

b. [⋆] Does this change the language? Either show some string that is in the language defined by the modified grammar but not in the original language (or vice versa), or argue that both grammars can generate exactly the same sets of strings.

Exercise 2.12. [⋆] The grammar for whole numbers we defined allows strings with non-standard leading zeros such as "000" and "00005". Devise a grammar that produces all whole numbers (including "0"), but no strings with unnecessary leading zeros.

Excursion 2.2: Valid Dates. [⋆] Devise a grammar that defines the language of dates (e.g., "December 7, 1941"). Your language should include all valid dates, but no invalid dates (that is, "September 29, 2007" and "February 29, 2008" are in the language, but "February 29, 2009" is not).

2.6 Power of Language Systems

We claimed that recursive transition networks and BNF replacement grammars are *equally* powerful. Here, we explain more precisely what that means and prove that the two systems are, in fact, equivalent in power.

First, what does it mean to say two systems are equally powerful? The purpose of a language description mechanism is to define a set of strings comprising a language. Hence, the power of a language description mechanism is determined by the set of languages (that is, a set of sets of strings) it can define.

One approach to consider is counting the number of languages that can

be defined. Even the simplest mechanisms can define infinitely many languages, however, so just counting the number of languages does not distinguish well between the different language description mechanisms. For example, even with the table listing all surface forms in the language (as introduced in Section 2.1) we can define infinitely many different languages. There is no limit on the number of entries in the table, so we can always add one more entry containing a new surface form to define a new language. Similarly, we can argue that both RTNs and BNFs can describe infinitely many different languages. We can always add a new edge to an RTN to increase the number of strings in the language, or add a new replacement rule to a BNF that replaces a nonterminal with a new terminal symbol.

Instead, we need to consider the particular languages that each mechanism can define. A system A is more powerful that another system B if we can use A to define every language that can be defined by B, and there is some language L that can be defined using A that cannot be defined using B. This matches our intuitive interpretation of *more powerful* — A is more powerful than B if it can do everything B can do and more. The set diagrams in Figure 2.6 depict three possible scenarios.

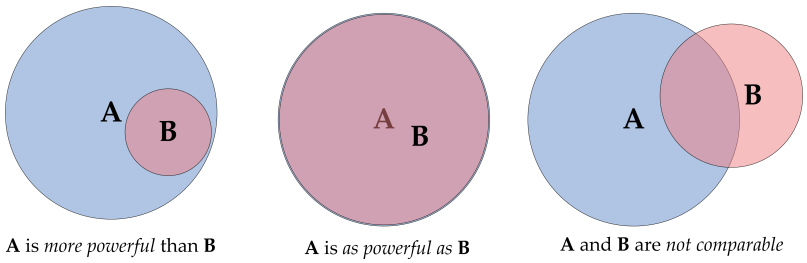


Figure 2.6. System power relationships.

In the leftmost picture, the set of languages that can be defined by B is a proper subset of the set of languages that can be defined by A. Hence, A is more powerful than B. In the center picture, the sets are equal. This means every language that can be defined by A can also be defined by B, and every language that can be defined by B can also be defined by A, and the systems are equally powerful. In the rightmost picture, there are some elements of A that are not elements of B, but there are also some elements of B that are not elements of A. This means we cannot say either one is more powerful; A can do some things B cannot do, and B can do some things A cannot do.

So, to determine the relationship between RTNs and BNFs, we need to understand if there are languages that can be defined by an RTN that cannot be defined by a BNF and if there are languages that can be defined by a

BNF that cannot be defined by an RTN.

First, we will prove that there are no languages that can be defined by a BNF that cannot be defined by an RTN. This is equivalent to showing that *every* language that can be defined by a BNF grammar has a corresponding RTN. Since there are infinitely many languages that can be defined by BNF grammars, we obviously cannot prove this by enumerating each language and showing the corresponding RTN. Instead, we use a proof technique commonly used in computer science: *proof by construction*. We show that given any BNF grammar we can construct a corresponding RTN. That is, define an algorithm that takes as input a BNF grammar, and produces as output an RTN that defines the same language as the input BNF grammar.

Our general strategy is to construct a subnetwork corresponding to each nonterminal. For each rule where the nonterminal is on the left side, the right hand side is converted to a path through that node's subnetwork. Here is our algorithm for converting a BNF grammar to an equivalent RTN:

1. For each nonterminal X in the grammar, construct two nodes, *StartX* and *EndX*, where *EndX* is a final node. Make the node *StartS* the start node of the RTN, where S is the start nonterminal of the grammar.
2. For each rule in the grammar, add a corresponding path through the RTN. All BNF rules have the form X ::⇒ *replacement* where X is a nonterminal in the grammar and *replacement* is a sequence of zero or more terminals and nonterminals: $[R_0, R_1, \ldots, R_n]$.
   (a) If the replacement is empty, make *StartX* a final node.
   (b) If the replacement has just one element, R_0, add an edge from *StartX* to *EndX* with edge label R_0.
   (c) Otherwise:
      i. Add an edge from *StartX* to a new node labeled $X_{i,0}$ (where i identifies the grammar rule), with edge label R_0.
      ii. For each remaining element R_j in the replacement add an edge from $X_{i,j-1}$ to a new node labeled $X_{i,j}$ with edge label R_j. (For example, for element R_1, a new node $X_{i,1}$ is added, and an edge from $X_{i,0}$ to $X_{i,1}$ with edge label R_1.)
      iii. Add an edge from $X_{i,n-1}$ to *EndX* with edge label R_n.

Following this procedure, we can convert any BNF grammar into an RTN that defines the same language. Hence, we have proved that RTNs are at least as powerful as BNF grammars.

To complete the proof that BNF grammars and RTNs are equally powerful ways of defining languages, we also need to show that a BNF can define

every language that can be defined using an RTN. This part of the proof can be done using a similar strategy: by showing a procedure that can be used to construct a BNF equivalent to any input RTN. We leave the details as an exercise for especially ambitious readers.

Excursion 2.3: Power comparison. [⋆] Prove that BNF grammars are not more powerful than Post production systems.

Excursion 2.4: BNF-RTN equivalence. [⋆] Prove that BNF grammars are as powerful as RTNs by devising a procedure that can construct a BNF grammar that defines the same language as any input RTN.

3

Programming

The Analytical Engine has no pretensions whatever to originate any thing. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with.

Augusta Ada, Countess of Lovelace,
in *Notes on the Analytical Engine*, 1843

What distinguishes a computer from other tools is its *programmability*. Without a program, a computer is an overpriced and not very effective door stopper. With the right program, though, a computer can be a tool for communicating across the continent, discovering a new molecule that can cure cancer, writing and recording a symphony, or managing the logistics of a retail empire.

Golden Gate Bridge

Programming is the act of writing instructions that make the computer do something useful. It is an intensely creative activity, involving aspects of art, engineering, and science. Good programs are written to be executed efficiently by computers, but also to be read and understood by humans. The best programs are delightful in ways similar to the best architecture, elegant in both form and function.

The ideal programmer would have the vision of Issac Newton, the intellect of Albert Einstein, the memory of Joshua Foer, the courage of Amelia Earhart, the determination of Michael Jordan, the pragmatism of Abraham Lincoln, the creativity of Miles Davis, the aesthetic sense of Maya Lin, the wisdom of Benjamin Franklin, the foresight of Garry Kasparov, the hindsight of Edward Gibbon, the writing talents of William Shakespeare, the oratorical skills of Martin Luther King, the audacity of John Roebling, the humility of Socrates, and the self-confidence of Grace Hopper.

Fortunately, it is not necessary to possess all of those rare qualities to be a good programmer! Indeed, anyone who is able to master the intellectual challenge of learning a language (which, presumably, anyone who has gotten this far has done at least for English) can become a good programmer. Since programming is a new way of thinking, many people find it challenging and even frustrating at first. Because the computer does exactly what

it is told, any small mistake in a program may prevent it from working as intended. With a bit of patience and persistence, however, the tedious parts of programming become easier, and you will be able to focus your energies on the fun and creative problem solving parts.

In the previous chapter, we explored the components of language and mechanisms for defining languages. In this chapter, we explain why natural languages are not a satisfactory way for defining procedures and introduce languages for programming computers and how they are used to define procedures.

3.1 Problems with Natural Languages

Natural languages, such as English, work adequately (most, but certainly not all, of the time) for human-human communication, but are not well-suited for human-computer or computer-computer communication. Why can't we use natural languages to program computers?

Next, we survey several of the reasons for this, focusing on specifics from English, although all natural languages suffer from all of these problems to varying degrees.

Complexity. Although English may seem simple to you now, it took many years of intense effort (most of it subconscious) for you to learn it. Despite using it for most of your waking hours for many years (assuming you are a native English speaker), you only know a small fraction of the entire language. The Oxford English Dictionary contains 615,000 words, of which a typical native English speaker knows about 40,000.

Ambiguity. Not only do natural languages have huge numbers of words, most words have many different meanings. To understand which meaning is intended requires knowing the context, and sometimes pure guesswork.

For example, what does it mean to be paid *biweekly*? According to the American Heritage Dictionary [Her07], *biweekly* has two definitions:

1. *Happening every two weeks.*
2. *Happening twice a week; semiweekly.*

Merriam-Webster's Dictionary takes the opposite approach [Onl08]:

1. *occurring twice a week*
2. *occurring every two weeks : fortnightly*

So, depending on which definition is intended, someone who is paid biweekly could either be paid once or four times every two weeks! One would not want the correct behavior of a payroll management program to depend on how biweekly is interpreted.

Even if we can agree on the definition of every word, the meaning of a sentence is often ambiguous. Here is one of my favorite examples, taken from the instructions with a shipment of ballistic missiles from the British Admiralty:

> *It is necessary for technical reasons that these warheads be stored upside down, that is, with the top at the bottom and the bottom at the top. In order that there be no doubt as to which is the bottom and which is the top, for storage purposes, it will be seen that the bottom of each warhead has been labeled 'TOP'.* [Par88]

Irregularity. Because natural languages evolve over time as different cultures interact and speakers misspeak and listeners mishear, natural languages end up a morass of irregularity. Nearly all grammar rules have exceptions. For example, English has a rule that we can make a word plural by appending an *s*. The new word means "more than one of the original word's meaning". This rule works for most words: *word* $\mapsto$ *words*, *language* $\mapsto$ *languages*, *person* $\mapsto$ *persons*.[1] It does not work for *all* words, however. The plural of *goose* is *geese* (and *gooses* is not an English word), the plural of *deer* is *deer* (and *deers* is not an English word), and the plural of *beer* is controversial (and may depend on whether you speak American English or Canadian English). These irregularities can be charming for a natural language, but they are a constant source of difficulty for non-native speakers attempting to learn a language. There is no sure way to predict when the rule can be applied, and it is necessary to memorize each of the irregular forms.

Uneconomic. It requires a lot of space to express a complex idea in a natural language. Many superfluous words are needed for grammatical correctness, even though they do not contribute to the desired meaning. Since natural languages evolved for everyday communication, they are not well suited to describing the precise steps and decisions needed in a computer program.

I didn't have time to write a short letter, so I wrote a long one instead.
Mark Twain

As an example, consider a procedure for finding the maximum of two numbers. In English, we could describe it like this:

> *To find the maximum of two numbers, compare them. If the first number is greater than the second number, the maximum is the first number. Otherwise, the maximum is the second number.*

[1] Or is it *people*? What is the singular of *people*? What about *peeps*? Can you only have one *peep*?

Perhaps shorter descriptions are possible, but any much shorter description probably assumes the reader already knows a lot. By contrast, we can express the same steps in the Scheme programming language in very concise way: (**define** (*bigger a b*) (**if** (> *a b*) *a b*)). (Don't worry if this doesn't make sense yet—it should by the end of this chapter.)

Limited means of abstraction. Natural languages provide small, fixed sets of pronouns to use as means of abstraction, and the rules for binding pronouns to meanings are often unclear. As discussed in Section 2.2, the means of abstraction available in English are particularly poor. Since programming often involves using simple names to refer to complex things, we need more powerful means of abstraction than natural languages provide.

3.2 Programming Languages

For programming computers, we want languages that are simple, unambiguous, regular, economical, and that provide more powerful means of abstraction. A *programming language* is a language that is designed to be read and written by humans to create programs that can be executed by computers.

Programming languages come in many flavors. It is difficult to simultaneously satisfy all the goals, in particular simplicity is often at odds with economy and powerful means of abstraction. Every feature that is added to a language to increase its expressiveness incurs a cost in reducing simplicity and regularity.

Another reason there are many different programming languages is that they are at different *levels of abstraction*. Some languages provide programmers with detailed control over machine resources, such as selecting a particular location in memory where a value is stored. Other languages hide most of the details of the machine operation from the programmer, allowing them to focus on higher-level actions.

Ultimately, we want a program the computer can execute. This means at the lowest level we need languages the computer can understand directly. At this level, the program is just a sequence of bits encoding machine instructions. Code at this level is not easy for humans to understand or write, but it is easy for a processor to execute quickly. The machine code encodes instructions that direct the processor to take simple actions like moving data from one place to another, performing simple arithmetic, and jumping around to find the next instruction to execute.

For example, the bit sequence `1110101111111110` encodes an instruction

in the Intel x86 instruction set (used on most PCs) that tells the processor to jump backwards two locations. Since two locations is the amount of space needed to hold this instruction, jumping back two locations actually jumps back to the beginning of this instruction. Hence, it gets stuck running forever without making any progress. The computer's processor is designed to execute very simple instructions like this one. This means each instruction can be executed very quickly. A typical modern processor can execute *billions* of instructions in a single second.[2]

Grace Hopper, 1952

Image courtesy Computer History Museum

Until the early 1950s, all programming was done at the level of simple instructions. The problem with instructions at this level is that they are not easy for humans to write and understand, and you need many simple instructions before you have a useful program.

In the early 1950s, Admiral Grace Hopper developed the first compilers. A *compiler* is a computer program that generates other programs. It can translate an input program written in a high-level language that is easier for humans to create into a program in a machine-level language that is easier for a computer to execute.

An alternative to a compiler is an interpreter. An *interpreter* is a tool that translates between a higher-level language and a lower-level language, but where a compiler translates an entire program at once and produces a machine language program that can be executed directly, an interpreter interprets the program a small piece at a time while it is running. This has the advantage that we do not have to run a separate tool to compile a program before running it; we can simply enter our program into the interpreter and run it right away. This makes it easy to make small changes to a program and try it again, and to observe the state of our program as it is running.

Nobody believed that I had a running compiler and nobody would touch it. They told me computers could only do arithmetic.
Grace Hopper

A disadvantage of using an interpreter instead of a compiler is that because the translation is happening while the program is running, the program may execute much slower than a similar compiled program would. Another advantage of compilers over interpreters is that since the compiler translates the entire program it can also analyze the program for consistency and detect certain types of programming mistakes automatically instead of encountering them when the program is running (or worse, not detecting them at all and producing unintended results). This is especially important when writing large, critical programs such as flight control software — we want to detect as many problems as possible in the flight control software before the plane is flying!

[2] When a computer is marketed as a "2GHz processor" that means the processor executes 2 billion cycles per second. This does not map directly to the number of instructions it can execute in a second, though, since some instructions take several cycles to execute.

3.3 Scheme

For now, we are more concerned with interactive exploration than with performance and detecting errors early, so we use an interpreter instead of a compiler. The programming system we use for the first part of this book is depicted in Figure 3.1.

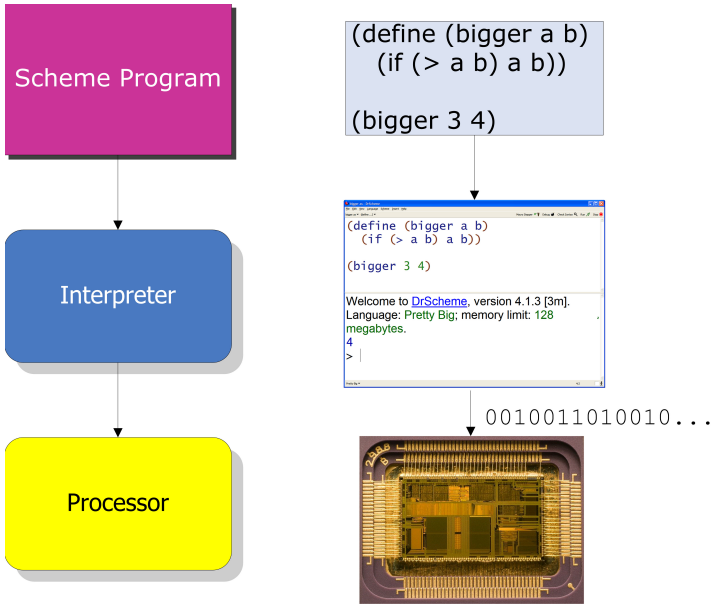


Figure 3.1. Running a Scheme program.

The input to our programming system is a program written in a programming language named *Scheme*.[3] Scheme was developed at MIT in the 1970s by Guy Steele and Gerald Sussman, based on the LISP programming language that was developed by John McCarthy in the 1950s. A Scheme interpreter interprets a Scheme program and executes it on the machine processor.

Although Scheme is not widely used in industry, it is a great language for learning about computing and programming. The primary advantage of using Scheme to learn about computing is its simplicity and elegance. The language is simple enough that you will learn nearly the entire language by the end of this chapter (we defer describing a few aspects until Chapter 11), and by the end of this book you will know enough to implement your own Scheme interpreter. By contrast, some programming languages that are widely used in industrial programming such as C++ and Java re-

[3]Originally, it was named "Schemer", but the machine used to develop it only supported 6-letter file names, so the name was shortened to "Scheme".

quire thousands of pages to describe, and even the world's experts in those languages do not agree on exactly what all programs mean.

Although almost everything we describe should work in all Scheme interpreters, for the examples in this book we assume the DrScheme programming environment which is freely available from *http://www.drscheme.org/*. DrScheme includes interpreters for many different languages, so you must select the desired language using the Language menu. The selected language defines the grammar and evaluation rules that will be used to interpret your program. For all the examples in this book, we use the language named Pretty Big.

3.4 Expressions

Scheme programs are composed of expressions and definitions (Section 3.5). An *expression* is a syntactic element that has a *value*. The act of determining the value associated with an expression is called *evaluation*. A Scheme interpreter, such as the one provided in DrScheme, is a machine for evaluating Scheme expressions. If you enter an expression to a Scheme interpreter, it responds by displaying the value of that expression.

Expressions may be primitives. Scheme also provides means of combination for producing complex expressions from simple expressions. The next subsections describe primitive expressions and application expressions. Section 3.6 describes expressions for making procedures and Section 3.7 describes expressions that can be used to make decisions.

3.4.1 Primitives

An expression can be replaced with a primitive:

$$\textit{Expression} ::\Rightarrow \textit{PrimitiveExpression}$$

As with natural languages, primitives are the smallest units of meaning. Hence, the value of a primitive is its pre-defined meaning.

Scheme provides many different primitives. Three useful types of primitives are described next: numbers, Booleans, and primitive procedures.

Numbers. Numbers represent numerical values. Scheme provides all the kinds of numbers you are familiar with, and they mean almost exactly what

you think they mean.[4]

Example numbers include:

```
150        0     −12
3.14159    3/4   999999999999999999999
```

Numbers evaluate to their value. For example, the value of the primitive expression 150 is 150.

Booleans. Booleans represent truth values. There are two primitives for representing true and false:

$$\textit{PrimitiveExpression} ::\Rightarrow \texttt{true} \mid \texttt{false}$$

Unsurprisingly, the meaning of true is true, and the meaning of false is false.[5]

function

Primitive Procedures. Scheme provides primitive procedures corresponding to many common functions. Mathematically, a *function* is a mapping from inputs to outputs. A function has a *domain*, the set of all inputs that it accepts. For each input in the domain, there is exactly one associated output. For example, + is a procedure that takes zero or more inputs, each of which must be a number. The output it produces is the sum of the values of the inputs. (We cover how to apply a function in the next subsection.)

Table 3.1 describes some of the primitive procedures.

[4]The details of managing numbers on computers are complex, and we do not consider them here.

[5]In the DrScheme interpreter, #t and #f are used as the primitive truth values; they mean the same thing as true and false. So, when you evaluate something that evaluates to true, it will appear as #t in the interactions window.

Symbol	Description	Inputs	Output
+	add	zero or more numbers	sum of the input numbers (0 if there are no inputs)
*	multiply	zero or more numbers	product of the input numbers (1 if there are no inputs)
−	subtract	two numbers	the value of the first number minus the value the second number
/	divide	two numbers	the value of the first number divided by the value of the second number
zero?	is zero?	one number	true if the input value is 0, otherwise false
=	is equal to?	two numbers	true if the input values have the same value, otherwise false
<	is less than?	two numbers	true if the first input value has lesser value than the second input value, otherwise false
>	is greater than?	two numbers	true if the first input value has greater value than the second input value, otherwise false
<=	is less than or equal to?	two numbers	true if the first input value is not greater than the second input value, otherwise false
>=	is greater than or equal to?	two numbers	true if the first input value is not less than the second input value, otherwise false

Table 3.1. Selected Scheme Primitive Procedures.
All of these primitive procedures operate on numbers. The first four are the basic arithmetic operators; the rest are comparison procedures. Some of these procedures are defined for more inputs than just the ones shown here. For example, the subtract procedure also works on one number, producing its negation.

3.4.2 Application Expressions

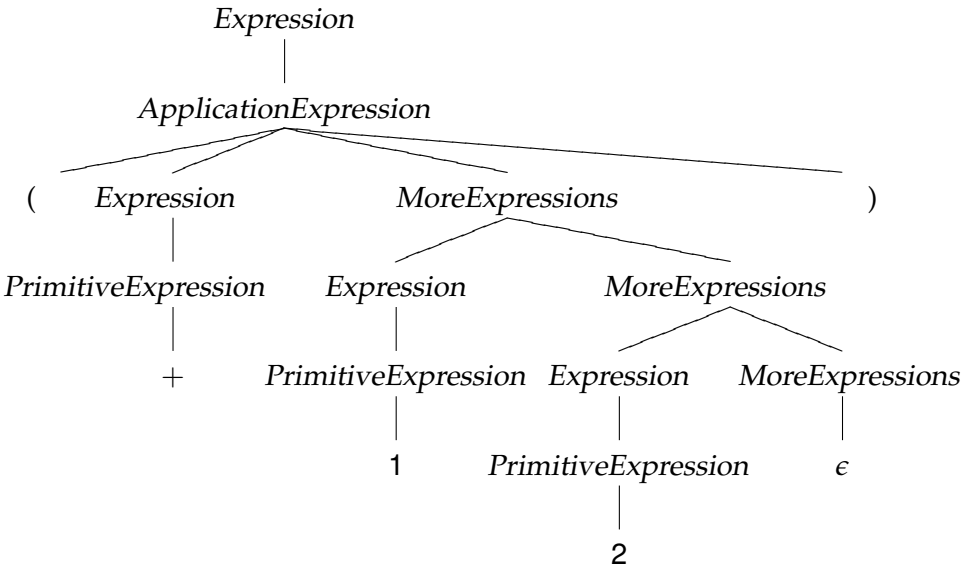
Most of the actual work done by a Scheme program is done by application expressions. The grammar rule for application is:

$$
\begin{array}{lll}
\textit{Expression} & ::\Rightarrow & \textit{ApplicationExpression} \\
\textit{ApplicationExpression} & ::\Rightarrow & (\textit{Expression MoreExpressions}) \\
\textit{MoreExpressions} & ::\Rightarrow & \epsilon \mid \textit{Expression MoreExpressions}
\end{array}
$$

This rule generates a list of one or more expressions surrounded by parentheses. The value of the first expression should be a procedure. All of the primitive procedures are procedures; in Section 3.6, we will see how to create new procedures. The remaining expressions are the inputs to the procedure.

For example, the expression (+ 1 2) is an *ApplicationExpression*, consisting of three subexpressions. Although this example is probably simple enough that you can probably guess that it evaluates to 3, we will demonstrate in detail how it is evaluated by breaking down into its subexpressions using the grammar rules. The same process will allow us to understand how *any* expression is evaluated.

Here is a parse tree for the expression (+ 1 2):

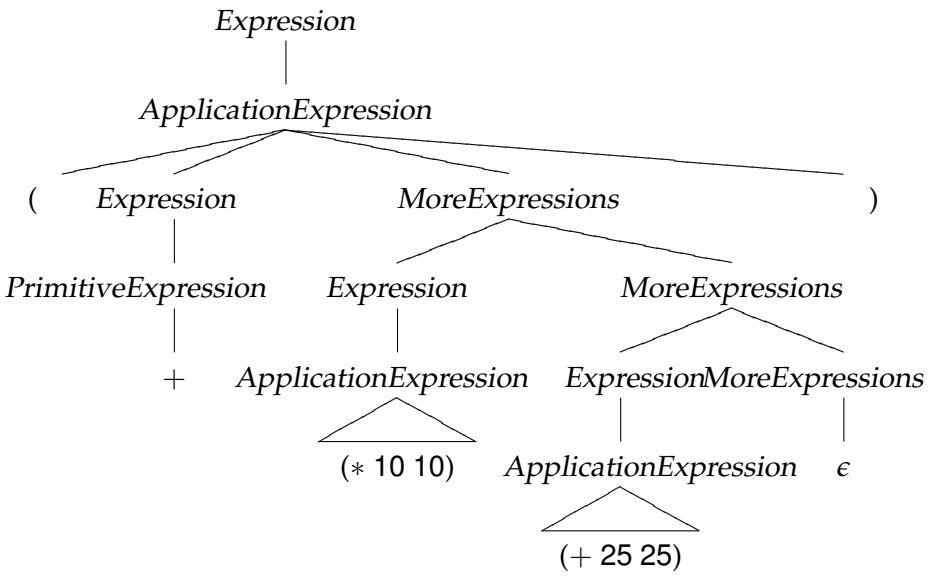


Following the grammar rules, we replace *Expression* with *ApplicationExpression* at the top of the parse tree. Then, we replace *ApplicationExpres-*

sion with (*Expression MoreExpressions*). The *Expression* term is replaced *PrimitiveExpression*, and finally, the primitive addition procedure $+$. This is the first subexpression of the application, so it is the procedure to be applied. The *MoreExpressions* term produces the two operand expressions: 1 and 2, both of which are primitives that evaluate to their own values. The application expression is evaluated by applying the value of the first expression (the primitive procedure $+$) to the inputs given by the values of the other expressions. Following the meaning of the primitive procedure, $(+\ 1\ 2)$ evaluates to 3 as expected.

As with any nonterminal, the *Expression* nonterminals in the application expression can be replaced with anything that appears on the right side of an expression rule, including the application expression rule. Hence, we can build up complex expressions like $(+\ (*\ 10\ 10)\ (+\ 25\ 25))$.

The parse tree is:



This tree is similar to the previous tree, except instead of the subexpressions of the first application expression being simple primitive expressions, they are now application expressions. (Instead of showing the complete parse tree for the nested application expressions, we use triangles.)

To evaluate the output application, we need to evaluate all the subexpressions. The first subexpression, $+$, evaluates to the primitive procedure. The second subexpression, $(*\ 10\ 10)$, evaluates to 100, and the third expression, $(+\ 25\ 25)$, evaluates to 50. Now, we can evaluate the original expression using the values for its three component subexpressions: $(+\ 100\ 50)$ evaluates to 150.

Exercise 3.1. Draw a parse tree for the Scheme expression

$$(+\ 100\ (*\ 5\ (+\ 5\ 5)))$$

and show how it would be evaluated.

Exercise 3.2. Predict how each of the following Scheme expressions is evaluated. After making your prediction, try evaluating the expression in DrScheme. If the result is different from your prediction, explain why the Scheme interpreter evaluates the expression as it does.

a. 150

b. $(+\ 150)$

c. $(+\ (+\ 100\ 50)\ (*\ 2\ 0))$

d. $(=\ (+\ 100\ 50)\ (*\ 15\ (+\ 5\ 5)))$

e. $(zero?\ (-\ 150\ (+\ 50\ 50\ (+\ 25\ 25))))$

f. $+$

g. [⋆] $(+\ +\ <)$

Exercise 3.3. For each problem, construct a Scheme expression that calculates the result and try evaluating it in DrScheme.

a. How many seconds are there in a year?

b. For how many seconds have you been alive?

c. For what fraction of your life have you been in school?

Exercise 3.4. Construct a Scheme expression to calculate the distance in inches that light travels during the time it takes the processor in your computer to execute one cycle.

A meter is defined as the distance light travels in $1/299792458^{th}$ of a second in a vacuum. One meter is 100 centimeters, and one inch is defined as 2.54 centimeters. Your processor speed is probably given in *gigahertz* (GHz), which are 1,000,000,000 hertz. One hertz means once per second, so 1GHz means the processor executes 1,000,000,000 cycles per second. On a Windows machine, you can find the speed of your processor by opening the Control Panel (select it from the Start menu) and selecting System. Note that Scheme performs calculations exactly, so the result will be displayed as a fraction. To see a more useful answer, use (*exact->inexact Expression*) to convert the value of the expression to a decimal representation.

3.5 Definitions

Scheme provides a simple, yet powerful, mechanism for abstraction. We can introduce a new name using a definition:

$$\textit{Definition} ::\Rightarrow \text{ (}\textbf{define}\ \textit{Name}\ \textit{Expression}\text{)}$$

After a definition, the name in the definition is now associated with the value of the expression in the definition.[6] A definition is not an expression since it does not evaluate to a value.

A name can be any sequence of letters, digits, and special characters (such as $-$, $>$, *?*, and *!*) that starts with a letter or special character. Examples of valid names include *a*, *Ada*, *Augusta-Ada*, *gold49*, *!yuck*, and *yikes!*\%@\#. We don't recommend using some of these names in your programs, however! A good programmer will pick names that are easy to read, pronounce, and remember, and that are not easily confused with other names.

After a name has been bound to a value by a definition, that name may be used in an expression:

$$\begin{array}{ll} \textit{Expression} & ::\Rightarrow \textit{NameExpression} \\ \textit{NameExpression} & ::\Rightarrow \textit{Name} \end{array}$$

The value of a *NameExpression* is the value associated with the name.

For example, below we define *speed-of-light* to be the speed of light in meters per second, define *seconds-per-hour* to be the number of seconds in an hour, and use them to calculate the speed of light in kilometers per hour:

```
> (define speed-of-light 299792458)
> speed-of-light
299792458
> (define seconds-per-hour (* 60 60))
> (/ (* speed-of-light seconds-per-hour) 1000)
1079252848 4/5
```

[6] Alert readers should be worried that we need a more precise definition of the meaning of definitions to know what it means for a value to be associated with a name. This one will serve us well for now, but we will provide a more precise explanation of the meaning of a definition in Chapter 11.

3.6 Procedures

In Chapter 1 we defined a procedure as a description of a process. Scheme provides a way to define procedures that take inputs, carry out a sequence of actions, and produce an output. In Section 3.4.1, we saw that Scheme provides some primitive procedures. To construct complex programs, however, we need to be able to create our own procedures.

Procedures are similar to mathematical functions in that they provide a mapping between inputs and outputs, but they are different from mathematical functions in two key ways:

- State — in addition to producing an output, a procedure may access and modify state. This means that even when the same procedure is applied to the same inputs, the output produced may vary. Because mathematical functions do not have external state, when the same function is applied to the same inputs it always produces the same result. State makes procedures much harder to reason about. In particular, it breaks the substitution model of evaluation we introduce in the next section. We will ignore this issue until Chapter 11, and focus until then only on procedures that do not involve any state.

- Resources — unlike an ideal mathematical function, which provides an instantaneous and free mapping between inputs and outputs, a procedure requires resources to execute before the output is produced. The most important resources are *space* (memory) and *time*. A procedure may need space to keep track of intermediate results while it is executing. Each step of a procedure requires some time to execute. Predicting how long a procedure will take to execute, and finding the fastest procedure possible for solving some problem, are core problems in computer science. We will consider this throughout this book, and in particular in Chapter 9. Even knowing if a procedure will finish (that is, ever produce an output) is a challenging problem. In Chapter 13 we will see that it is impossible to solve in general.

For the rest of this chapter, however, we will view procedures as idealized mathematical functions: we will consider only procedures that involve no state, and we will not worry about the resources our procedures require.

3.6.1 Making Procedures

Scheme provides a general mechanism for making a procedure:

Expression	::⇒	*ProcedureExpression*
ProcedureExpression	::⇒	(**lambda** (*Parameters*) *Expression*)
Parameters	::⇒	ϵ | ***Name*** *Parameters*

Evaluating a *ProcedureExpression* produces a procedure that takes as inputs the *Parameters* following the **lambda**.[7] You can think of **lambda** as meaning "make a procedure". The body of the procedure is the *Expression*, which is not evaluated until the procedure is applied.

Note that a *ProcedureExpression* can replace an *Expression*. This means anywhere an *Expression* is used we can create a new procedure. This is very powerful since it means we can use procedures as inputs to other procedures and create procedures that return new procedures as their output!

Here are some example procedures:

- (**lambda** (*x*) (∗ *x x*)) — a procedure that takes one input, and produces the square of the input value as its output.
- (**lambda** (*a b*) (+ *a b*)) — a procedure that takes two inputs, and produces the sum of the input values as its output.
- (**lambda** () 0) — a procedure that takes no inputs, and produces 0 as its output.
- (**lambda** (*a*) (**lambda** (*b*) (+ *a b*))) — a procedure that takes one input (*a*), and produces as its output a procedure that takes one input and produces the sum of that input at *a* as its output. We can think of this procedure as a procedure that makes an adding procedure.

3.6.2 Substitution Model of Evaluation

For a procedure to be useful, we need to apply it. In Section 3.4.2, we saw the syntax and evaluation rule for an *ApplicationExpression* when the procedure to be applied is a primitive procedure. The syntax for applying a constructed procedure is identical to the syntax for applying a primitive procedure:

Expression	::⇒	*ApplicationExpression*
ApplicationExpression	::⇒	(*Expression MoreExpressions*)
MoreExpressions	::⇒	ϵ | *Expression MoreExpressions*

[7] Scheme uses **lambda** to make a procedure because it is based on LISP which is based on Lambda Calculus (see Chapter 17).

To understand how constructed procedures are evaluated, we need a new evaluation rule. In this case, the first *Expression* evaluates to a procedure that was created using a *ProcedureExpression*, so we can think of the *ApplicationExpression* as:

ApplicationExpression ::⇒
((**lambda** (*Parameters*)*Expression*) *MoreExpressions*)

(The underlined part is the replacement for the *ProcedureExpression*.)

To evaluate the application, we evaluate the *MoreExpressions* in the application expression. These expressions are known as the *operands* of the application. The resulting values are the input to the procedure. There must be exactly one expression in the *MoreExpressions* corresponding to each name in the parameters list. Next, evaluate the expression that is the body of the procedure. Whenever any parameter name is used inside the body expression, the name evaluates to the value of the corresponding input. This is similar to the way binding worked in Post production systems (Section 2.3). When a value is matched with a procedure parameter, that parameter is bound to the value. When the parameter name is evaluated, the result is the bound value.

Example 3.1: Square. Consider evaluating the following expression, which apples the squaring procedure to 2:

((**lambda** (*x*) (∗ *x x*)) 2)

It is an *ApplicationExpression* where the first sub-expression is the *ProcedureExpression*, (**lambda** (*x*) (∗ *x x*)). To evaluate the application, we evaluate all the subeexpressions and apply the value of the first subexpression to the values of the remaining subexpressions. The first subexpression evaluates to a procedure that takes one parameter named *x* and has the expression body (∗ *x x*). There is one operand expression, the primitive 2, that evaluates to 2.

To evaluate the application we bind the first parameter, *x*, to the value of the first operand, 2, and evaluate the procedure body, (∗ *x x*). After substituting the parameter values, we have (∗ 2 2). This is an application of the primitive multiplication procedure. Evaluating the application results in the value 4.

The procedure in our example, (**lambda** (*x*) (∗ *x x*)), is a procedure that takes a number as input and as output produces the square of that number. We

can use the definition mechanism (from Section 3.5) to give this procedure a name so we can reuse it:

(**define** *square* (**lambda** (*x*) (∗ *x x*)))

This defines the name *square* as the procedure. After this, we can use *square* to produce the square of any number:

```
> (square 2)
4
> (square 1/4)
1/16
> (square (square 2))
16
```

Example 3.2: Make adder. For the make an adding procedure example,

((**lambda** (*a*) (**lambda** (*b*) (+ *a b*))) 3)

produces a procedure that adds 3 to its input. Applying that procedure,

(((**lambda** (*a*) (**lambda** (*b*) (+ *a b*))) 3) 4)

evaluates to 7. By using **define**, we can give these procedures sensible names:

```
(define make-adder
        (lambda (a)
                (lambda (b) (+ a b))))
```

Then,

(**define** *add-three* (*make-adder* 3))

defines *add-three* as a procedure that takes one parameter and outputs the value of that parameter plus 3.

Abbreviated Procedure Definitions. Since we commonly need to define new procedures, Scheme provides a condensed notation for defining a procedure[8]:

[8]The condensed notation also includes a begin expression, which is a special form. We will not need the begin expression until we start dealing with procedures that have side-effects. We describe the **begin** special form in Chapter 11.

$$\textit{Definition} ::\Rightarrow \ (\textbf{define}\ (\textit{Name}\ \textit{Parameters})\ \textit{Expression})$$

This incorporates the **lambda** invisibly into the definition, but means exactly the same thing. For example,

(**define** *square* (**lambda** (*x*) (∗ *x x*)))

can be written equivalently as:

(**define** (*square x*) (∗ *x x*))

The two definitions mean exactly the same thing.

Exercise 3.5. Define a procedure, *cube*, that takes one number as input and produces as output the cube of that number.

Exercise 3.6. Define a procedure, *compute-cost*, that takes as input two numbers, the first represents that price of an item, and the second represents the sales tax rate. The output should be the total cost, which is computed as the price of the item plus the sales tax on the item, which is its price times the sales tax rate. For example, (*compute-cost* 13 0.05) should evaluate to 13.65.

3.7 Decisions

We would like to be able to make procedures where the actions taken depend on the input values. For example, we may want a procedure that takes two numbers as inputs and evaluates to the maximum value of the two inputs. To define such a procedure we need a way of making a decision. A *predicate* is a test expression that is used to determine which actions to take next. Scheme provides the **if** expression for determining actions based on a predicate.

The *IfExpression* replacement has three *Expression* terms. For clarity, we give each of them names as denoted by the subscripts:

$$\begin{aligned} \textit{Expression} &::\Rightarrow \textit{IfExpression} \\ \textit{IfExpression} &::\Rightarrow (\textbf{if}\ \textit{Expression}_{\text{Predicate}} \\ &\qquad\qquad \textit{Expression}_{\text{Consequent}} \\ &\qquad\qquad \textit{Expression}_{\text{Alternate}}) \end{aligned}$$

The evaluation rule for an *IfExpression* is to first evaluate $\textit{Expression}_{\text{Predicate}}$, the predicate expression. If it evaluates to any non-false value, the value of

the *IfExpression* is the value of $Expression_{\text{Consequent}}$, the consequent expression, and the alternate expression is not evaluated at all. If the predicate expression evaluates to false, the value of the *IfExpression* is the value of $Expression_{\text{Alternate}}$, the alternate expression, and the consequent expression is not evaluated at all. The predicate expression determines which of the two following expressions is evaluated to produce the value of the *IfExpression*.

Note that if the value of the predicate is *anything* other than false, the consequent expression is used. For example, if the predicate evaluates to true, to a number, or to a procedure the consequent expression is evaluated.

The if-expression is a *special form*. This means that although it looks syntactically identical to an application (that is, it could be an application of a procedure named **if**), it is not evaluated as a normal application would be. Instead, we have a special evaluation rule for if-expressions. The reason a special rule is needed is because we do not want all the subexpressions to be evaluated. With the normal application rule, all the subexpressions are evaluated, and then the procedure resulting from the first subexpression is applied to the values resulting from the others. With the if special form evaluation rule, the predicate expression is always evaluated, but only one of the following subexpressions is evaluated depending on the result of evaluating the predicate expression.

This means an if-expression can evaluate to a value even if evaluating one of its subexpressions would produce an error. For example,

(**if** (> 3 4) (∗ + +) 7)

evaluates to 7 even though evaluating the subexpression (∗ + +) would produce an error. Because of the special evaluation rule for if-expressions, the consequent expression is never evaluated.

Example 3.3: Bigger. Now that we have procedures, decisions, and definitions, we can understand the *bigger* procedure from the beginning of the chapter. The definition,

(**define** (*bigger a b*) (**if** (> *a b*) *a b*))

is a condensed procedure definition. It is equivalent to:

(**define** *bigger* (**lambda** (*a b*) (**if** (> *a b*) *a b*)))

This defines the name *bigger* as the value of evaluating the procedure expression (**lambda** (*a b*) (**if** (> *a b*) *a b*)). This is a procedure that takes two

inputs, named *a* and *b*. Its body is an if-expression with predicate expression (> *a b*). The predicate expression compares the value that is bound to the first parameter, *a*, with the value that is bound to the second parameter, *b*, and evaluates to true if the value of the first parameter is greater, and false otherwise. According to the evaluation rule for an if-expression, if the predicate evaluates to any non-false value (in this case, true), the value of the if-expression is the value of the consequent expression, *a*. If the predicate evaluates to false, the value of the if-expression is the value of the alternate expression, *b*. Hence, our *bigger* procedure takes two numbers as inputs and produces as output the greater of the two inputs.

Exercise 3.7. Follow the evaluation and application rules to evaluate the following Scheme expression:

(*bigger* 3 4)

where *bigger* is the maximum procedure defined as,

(**define** *bigger* (**lambda** (*a b*) (**if** (> *a b*) *a b*)))

It is very tedious to follow all of the steps (that's why we normally rely on computers to do it!), but worth doing once to make sure you understand the evaluation rules.

Exercise 3.8. Define a procedure, *xor*, that implements the logical exclusive-or operation. The *xor* function takes two inputs, and outputs true if exactly one of those outputs has a true value. Otherwise, it outputs false. For example, (*xor* true true) should evaluate to false and (*xor* (< 3 5) (= 8 8)) should evaluate to true.

Exercise 3.9. Define a procedure, *abs*, that takes a number as input and produces the absolute value of that number as its output. For example, (*abs* 3) should evaluate to 3, (*abs* −150) should evaluate to 150, and (*abs* 0) should evaluate to 0.

Exercise 3.10. [⋆] Define a procedure, *bigger-magnitude*, that takes two inputs, and produces as output the value of the input with the maximum magnitude (that is, absolute distance from zero). For example,

(*bigger-magnitude* 5 −7)

should evaluate to −7, and

(*bigger-magnitude* 9 −3)

should evaluate to 9.

Exercise 3.11. [⋆] Define a procedure, *biggest*, that takes three inputs, and produces as output the maximum value of the three inputs. For example,

(*biggest* 5 7 3)

should evaluate to 7. Try to find at least two different ways to define *biggest*, one using *bigger*, and one without using it.

3.8 Summary

At this point, we have covered enough of Scheme to write useful programs (even if the programs we have seen so far seem rather dull). In fact (as we will see in Chapter 17), we have covered enough to express *every* possible computation! We just need to combine the constructs we know in more complex ways to perform more interesting computations. The next chapter, and much of the rest of this book, focuses on ways to combine the constructs for making procedures, making decisions, and applying procedures in more powerful ways.

Here we summarize the grammar rules and evaluation rules. Each grammar rule has an associated evaluation rule. This means that any Scheme fragment that can be described by the grammar also has an associated meaning that can be produced by combining the evaluation rules corresponding to the grammar rules.

Program	::⇒	ϵ | *ProgramElement Program*
ProgramElement	::⇒	*Expression* | *Definition*

A program is a sequence of expressions and definitions.

Definition	::⇒	(**define** ***Name*** *Expression*)

A definition evaluates the expression, and associates the value of the expression with the name.

Definition	::⇒	(**define** (***Name*** *Parameters*) *Expression*)

Abbreviation for (**define** ***Name*** (**lambda** *Parameters*) *Expression*)

Expression ::⇒ *PrimitiveExpression* | *NameExpression* | *ApplicationExpression* | *ProcedureExpression* | *IfExpression*

The value of the expression is the value of the replacement expression.

PrimitiveExpression ::⇒ ***Number*** | true | false | *primitive procedure*

Evaluation Rule 1: Primitives. A primitive expression evaluates to its pre-defined value.

NameExpression ::⇒ ***Name***

Evaluation Rule 2: Names. A name evaluates to the value associated with that name.

ApplicationExpression ::⇒ (*Expression MoreExpressions*)

Evaluation Rule 3: Application. To evaluate an application expression:

a. **Evaluate** all the subexpressions;

b. Then, **apply** the value of the first subexpression to the values of the remaining subexpressions.

MoreExpressions ::⇒ ϵ | *Expression MoreExpressions*
ProcedureExpression ::⇒ (**lambda** (*Parameters*) *Expression*)

Evaluation Rule 4: Lambda. Lambda expressions evaluate to a procedure that takes the given parameters and has the expression as its body.

Parameters ::⇒ ϵ | ***Name*** *Parameters*

IfExpression ::⇒ (**if** $Expression_{\text{Predicate}}$ $Expression_{\text{Consequent}}$ $Expression_{\text{Alternate}}$)

Evaluation Rule 5: If. To evaluate an if-expression, (a) evaluate the predicate expression; then, (b) if the value of the predicate expression is a false value then the value of the if-expression is the value of the alternate expression; otherwise, the value of the if-expression is the value of the consequent expression.

The evaluation rule for an application (Rule 3b) uses **apply** to perform the application. We define **apply** using the two application rules:

- **Application Rule 1: Primitives.** If the procedure to apply is a primitive procedure, just do it.
- **Application Rule 2: Constructed Procedures.** If the procedure to apply is a constructed procedure, **evaluate** the body of the procedure with each parameter name bound to the corresponding input expression value.

Note that **evaluate** in the Application Rule 2 means use the evaluation rules above to evaluate the expression. Thus, the evaluation rules are defined using the application rules, which are defined using the evaluation rules! This appears to be a circular definition, but as with the grammar examples, it has a base case. There are some expressions we can evaluate without using the application rules (e.g., primitive expressions, name expressions), and some applications we can evaluate without using the evaluation rules (when the procedure to apply is a primitive). Hence, the process of evaluating an expression will sometimes finish and when it does we end with the value of the expression.[9]

[9]This does not guarantee it will *always* finish, however! We will see in some examples in the next chapter where evaluation never finishes.

4

Problems and Procedures

A great discovery solves a great problem, but there is a grain of discovery in the solution of any problem. Your problem may be modest, but if it challenges your curiosity and brings into play your inventive faculties, and if you solve it by your own means, you may experience the tension and enjoy the triumph of discovery.
George Pólya, *How to Solve It*

Computers are tools for performing computations to solve problems. In this chapter, we consider what it means to solve a problem and explore some strategies for constructing procedures that solve problems.

4.1 Solving Problems

Traditionally, a problem is an obstacle to overcome or some question to answer. Once the question is answered or the obstacle circumvented, the problem is solved and we can declare victory and move on to the next one.

When we talk about writing programs to solve problems, though, we usually have a larger goal. We don't just want to solve *one* instance of a problem, we want a procedure that can solve *all* instances of a problem. For example, we don't just want to find the best route between Charlottesville and Washington, we want to find a procedure that can find the best route between any two locations on a map. The inputs to the procedure are the map, the start location, and the end location, and the procedure should generate the best route as its output.[1] There are infinitely many possible inputs that each specify different instances of the problem of finding the best route between two locations on a map; a general solution to the problem is a procedure that can always find the best route for any possible inputs.

A *problem* is defined by its inputs and the desired property of the output. *problem*
Recall from Chapter 1 that a procedure is a precise description of a process. To define a procedure that can solve a problem, we need to define a pro-

[1] Actually finding a general procedure that does this is a challenging and interesting problem, that we will return to in Chapter 18.

cedure that takes inputs describing the problem instance and produces a different information process depending on the actual values of its inputs.

A procedure takes zero or more inputs, and produces one output or no outputs[2], as shown in Figure 4.1.

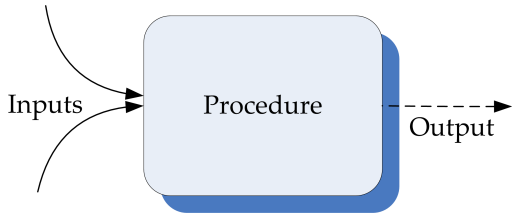


Figure 4.1. A procedure maps inputs to an output.

Our goal in solving a problem is to devise a procedure that takes inputs that define a problem instance, and produces as output the solution to the problem. This means every application of the procedure must eventually finish evaluating and produce an output value.

algorithm

A procedure is guaranteed to always finish is called an *algorithm*. The name algorithm is a Latinization of the name of the Persian mathematician and scientist, Muhammad ibn Mūsā al-Khwārizmī, who published a book in 825 on calculation with Hindu numerals. Al-Khwārizmī was also the responsible for defining algebra.

al-Khwārizmī

Although the name algorithm was not adopted until after al-Khwārizmī's book, algorithms go back much further than that. The ancient Babylonians had algorithms for finding square roots more than 3500 years ago (see Example 4).

There is no magic wand for solving all problems, but at its core most problem solving involves breaking problems you do not yet know how to solve into simpler and simpler problems until you find problems simple enough that you already know how to solve them. The trick is to find the right subproblems so that they can be combined to solve the original problem. This approach of solving problems by breaking them into simpler parts is known as *divide and conquer*.

divide and conquer

The following sections describe a few techniques for solving problems, and illustrate them with some simple examples. We will use these same problem-solving techniques over and over throughout this book. In these

[2] Although procedures can produce more than one output, we limit our discussion here to procedures that produce no more than one output. In the next chapter, we introduce ways to construct complex data, so any number of output values can be packaged into a single output.

examples, we limit ourselves to simple data: all of the problems have one or more numbers as their input, and produce one number as output. In the next chapter, we introduce structured data and revisit these problem solving techniques.

4.2 Composing Procedures

One way to divide a problem is to split it into steps where the output of the first step is the input to the second step, and the output of the second step is the solution to the problem. Each step can be defined by one procedure, and the two procedures can be combined to create one procedure that solves the problem.

Figure 4.2 shows a composition of two functions, f and g. The output of f is used as the input to g.

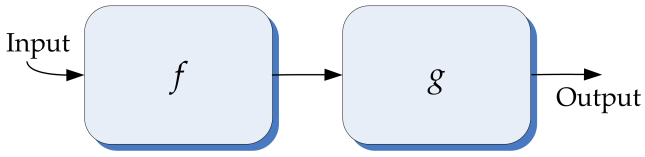


Figure 4.2. Composition.

We can express this composition with the Scheme expression (g (f x)) where x is the input. The written order appears to be reversed from the picture in Figure 4.2. This is because we apply a procedure to the values of its subexpressions: the values of the inner subexpressions must be computed first, and then used as the inputs to the outer applications. So, although f appears to the right of g in the expression, the subexpression (f x) is evaluated first since the evaluation rule for the outer application expression is to first evaluate all the subexpressions.

We can define a procedure that implements the composed procedure by making x a parameter:

(**define** *fog* (**lambda** (x) (g (f x))))

This defines *fog* as a procedure that takes one input and produces as output the composition of f and g applied to the input parameter. The works for any two procedures, as long as both procedures take a single input parameter.

For example, we could compose the *square* and *cube* procedures from Chapter 3 as:

(**define** *sixth-power* (**lambda** (*x*) (*cube* (*square x*))))

Then, (*sixth-power* 2) evaluates to 64.

4.2.1 Procedures as Inputs and Outputs

higher-order procedure

So far, all the procedure inputs and outputs we have seen have been numbers. But, the subexpressions of an application can be any expression including a procedure. A *higher-order procedure* is a procedure that takes other procedures as inputs or that produces a procedure as its output. Higher-order procedures give us the ability to write procedures that behave differently based on the procedures that are passed in as inputs.

For example, we can create a generic composition procedure by making *f* and *g* parameters:

(**define** *fog* (**lambda** (*f g x*) (*g* (*f x*))))

The *fog* procedure takes three parameters. The first two are both procedures that take one input. The third parameter is a value that can be the input to the first procedure.

For example,

```
> (fog square cube 2)
64
> (fog (lambda (x) (+ x 1)) square 2)
9
```

In the second example the first parameter is the procedure produced by the lambda-expression (**lambda** (*x*) (+ *x* 1)). This procedure takes a number as input and produces as output that number plus one. We define *inc* (short for increment) as this procedure:

(**define** *inc* (**lambda** (*x*) (+ *x* 1)))

A more useful composition procedure would separate the input value, *x*, from the composition. The *fcompose* procedure takes two procedures as inputs and produces as output a procedure that is their composition:[3]

[3]We name our composition procedure *fcompose* to avoid collision with the built-in *compose* procedure that behaves similarly.

```
(define fcompose
   (lambda (f g) (lambda (x) (g (f x)))))
```

The body of the *fcompose* procedure is a lambda expression that makes a procedure! Hence, the result of applying *fcompose* to two procedures is not a simple value, but a procedure. The resulting procedure can then be applied to a value.

Here are some examples using *fcompose*:

```
> (fcompose inc inc)
#<procedure>
> ((fcompose inc inc) 1)
3
> (define sixth-power (fcompose square cube))
> (sixth-power 3)
729
> (((fcompose inc square) 2)
9
> ((fcompose square inc) 2)
5
```

Note that the order in which procedures are composed matters. For example, (((*fcompose inc square*) 2)) evaluates to 9 since the input is incremented first, then squared; but, ((*fcompose square inc*) 2) evaluates to 5.

Exercise 4.1. Evaluating procedures. For each expression, give the value to which the expression evaluates. Assume *fcompose* and *inc* are defined as above.

a. (*fcompose* (**lambda** (*x*) (∗ *x* 2)) (**lambda** (*x*) (/ *x* 2)))

b. ((*fcompose* (**lambda** (*x*) (∗ *x* 2)) (**lambda** (*x*) (/ *x* 2))) 150)

c. ((*fcompose* (*fcompose inc inc*) *inc*) 2)

Exercise 4.2. Suppose we define *self-compose* as a procedure that composes a procedure with itself:

```
(define (self-compose f) (fcompose f f))
```

Explain how (((*fcompose self-compose self-compose*) *inc*) 1) is evaluated.

Exercise 4.3. Define a procedure *fcompose3* that takes three procedures as input, and produces as output a procedure that is the composition of the three input procedures. For example, ((*fcompose3 abs inc square*) −5) should evaluate to 36. Try to define *fcompose3* two different ways: once without using *fcompose*, and once using *fcompose*.

Exercise 4.4. The *fcompose* procedure only works when both input procedure take one input. Define a *f2compose* procedure that composes two procedures where the first procedure takes two inputs, and the second procedure takes one input. For example, ((*f2compose add abs*) 3 −5) should evaluate to 2.

4.3 Recursive Problem Solving

In the previous section, we used functional composition to break a problem into two procedures that can be composed to produce the desired output. A particularly useful variation on this is when we can break a problem into a smaller version of the original problem.

The goal is to be able to feed the output of one application of the procedure back into the same procedure as its input for the next application, as shown in Figure 4.3.

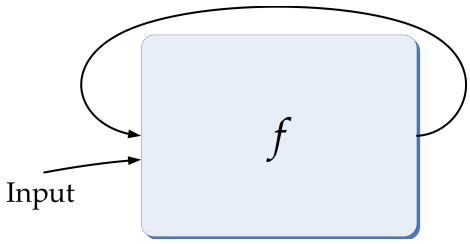


Figure 4.3. Circular Composition.

Here's a corresponding Scheme procedure:

(**define** *f* (**lambda** (*n*) (*f n*)))

Of course, this doesn't work very well![4] Every time an application of *f* is evaluated, it results in another application of *f* to evaluate. This never stops, stop so no output is ever produced and the interpreter will keep evaluating applications of *f* until it is stopper or runs out of memory.

[4]Curious readers should try entering this definition into a Scheme interpreter and evaluating (*f* 0). If you get tired of waiting for an output, in DrScheme you can click the **Stop** button in the upper right corner to interrupt the evaluation.

What we need is a way of making progress and eventually stopping, instead of going around in circles. To make progress, each subsequent application should have a smaller input. Then, the applications can stop when the input to the procedure is simple enough that we know the output directly. This is called the *base case*, similarly to the grammar rules in Section 2.5. In our grammar examples, the base case involved replacing the nonterminal with nothing (e.g., *MoreDigits* ::⇒ ϵ) or with a terminal (e.g., *Noun* ::⇒ **Alice**). In recursive procedures, the base case will provide a solution for some input for which the problem is so simple we already know the answer. When the input is a number, this is often (but not necessarily) when the input is zero or one.

base case

To define a recursive procedure, we need to use an if-expression to test if the input matches the base case input. If it does, the consequent expression is the known answer for the base case. Otherwise, we enter the recursive case and apply the procedure again but with a different input. Each time we apply the procedure we need to make progress towards reaching the base case. This means, the input has to change in a way that gets closer to the base case input. If the base case is for 0, and the original input is a positive number, one way to get closer to the base case input is to subtract 1 from the input value with each recursive application.

This evaluation spiral is depicted in Figure 4.4. With each subsequent recursive call, the input gets smaller, eventually reaching the base case. For the base case application, a result is returned to the previous application. This is passed back up the spiral to produce the final output.

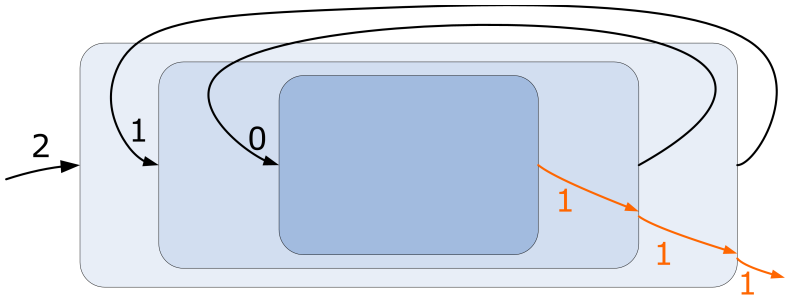


Figure 4.4. Recursive Composition.

Here is the corresponding procedure:

```
(define g
  (lambda (n)
    (if (= n 0) 1 (g (- n 1)))))
```

Unlike the earlier circular f procedure, if we apply g to any non-negative

integer it will eventually produce an output. For example, consider evaluating (g 2). When we evaluate the first application, the value of the parameter n is 2, so the predicate expression ($= n$ 0) evaluates to false and the value of the procedure body is the value of the alternate expression, (g ($-$ n 1)). The subexpression, ($-$ n 1) evaluates to 1, so the result is the result of applying g to 1. As with the previous application, this leads to the application, (g ($-$ n 1)), but this time the value of n is 1, so ($-$ n 1) evaluates to 0. The next application leads to the application, (g 0). This time, the predicate expression evaluates to true and we have reached the base case. The consequent expression is just 1, so no further applications of g are performed and this is the result of the application (g 0). This is returned as the result of the (g 1) application in the previous recursive call, and then as the output of the original (g 2) application.

We can think of the recursive evaluation as winding until the base case is reached, and then unwinding the outputs back to the original application. For this procedure, the output is not very interesting: no matter what positive number we apply g to, the eventual result is 1. To solve interesting problems with recursive procedures, we need to accumulate results as the recursive applications wind or unwind. Examples 1 and 2 illustrate recursive procedures that accumulate the result during the unwinding process. Example 3 illustrates a recursive procedure that accumulates the result during the winding process.

Example 4.1: Factorial. How many different arrangements are there of a deck of 52 playing cards? The top card in the deck can be any of the 52 cards, so there are 52 possible choices for the top card. The second card can be any of the cards except for the card that is the top card, so there are 51 possible choices for the second card. The third card can be any of the 50 remaining cards, and so on, until the last card for which there is only one choice remaining. To determine the total number of possible arrangements we need to multiply the number of choices for each card:

$$52 * 51 * 50 * \cdots * 2 * 1$$

This is known as the *factorial* function (denoted in mathematics using the exclamation point, e.g., 52!). It is defined as:

$$n! = \begin{cases} 1 & : \quad n = 0 \\ n * (n-1)! & : \quad n > 0 \end{cases}$$

The mathematical definition of factorial is recursive, so it is natural that we can define a recursive procedure that computes factorials:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

We can now determine the number of deck arrangements by evaluating (*factorial* 52) to get a sixty-eight digit number starting with an 8.

The *factorial* procedure has structure very similar to our earlier definition of the useless recursive *g* procedure. The only difference is the alternative expression for the if-expression: in *g* we used (*g* (− *n* 1)); in *factorial* we added the outer application of ∗: (∗ *n* (*factorial* (− *n* 1))). Instead of just evaluating to the result of the recursive application, we are now combining the output of the recursive evaluation with the input *n* using a multiplication application.

Exercise 4.5. How many different ways are there of choosing an unordered 5-card hand from a 52-card deck?

This is an instance of the "*n* choose *k*" problem (also known as the binomial coefficient): how many different ways are there to choose a set of *k* items from *n* items. There are *n* ways to choose the first item, $n-1$ ways to choose the second, ..., and $n-k+1$ ways to choose the k^{th} item. But, since the order does not matter, some of these ways are equivalent. The number of possible ways to order the *k* items is *k*!, so we can compute the number of ways to choose *k* items from a set of *n* items as:

$$\frac{n*(n-1)*\cdots*(n-k+1)}{k!} = \frac{n!}{(n-k)!k!} \tag{4.1}$$

a. [⋆] Define a procedure *choose* that takes two inputs, *n* (the size of the item set) and *k* (the number of items to choose), and outputs the number of possible ways to choose *k* items from *n*.

b. [⋆] Compute the number of possible 5-card hands that can be dealt from a 52-card deck.

c. [⋆] Compute the likelihood of being dealt a flush (5 cards all of the same suit). In a standard 52-card deck, there are 13 cards of each of the four suits. Hint: divide the number of possible flush hands by the number of possible hands.

Exercise 4.6. When Karl Gauss was in elementary school, his teacher assigned the class the task of summing the integers from 1 to 100 (e.g., $1+2+3+\cdots+100$) to keep them busy. Being the (future) "Prince of Mathematics", Gauss developed the formula for calculating this sum, that

Karl Gauss

is now known as the *Gauss sum*. Had he been a computer scientist, however, and had access to a Scheme interpreter in the late 1700s, he might have instead defined a recursive procedure to solve the problem.

a. [⋆] Define a recursive procedure, *gauss-sum*, that takes a number n as its input parameter, and evaluates to the sum of the integers from 1 to n as its output. For example, (*gauss-sum* 100) should evaluate to 5050.

b. [⋆] Define a higher-order procedure, *accumulate*, that can be used to make both *gauss-sum* and *factorial*. The *accumulate* procedure should take two inputs: the first is the function used for accumulation (e.g., $*$ for *factorial*, $+$ for *gauss-sum*); the second is the base case value (that is, the value of the function when the input is 0). With your *accumulate* procedure, ((*accumulate* $+$ 0) 100) should evaluate to 5050 and ((*accumulate* $*$ 1) 3) should evaluate to 6.

Hint: since your procedure should produce a procedure as its output, it could start like this:

```
(define (accumulate f base)
  (lambda (n)
    ...
```

Example 4.2: Maximum. In Chapter 3, we saw a procedure, *max*, that takes two inputs and evaluates to the greater input. Here, we consider the problem of defining a procedure that takes as its input a procedure, a low value, and a high value, and outputs the maximum value the input procedure produces when applied to an integer value between the low value and high value input. We will name the inputs *f*, *low*, and *high*. To find the maximum, the *find-maximum* procedure should evaluate the input procedure *f* at every integer value between the *low* and *high*, and produce as output the greatest value found.

To define the procedure, we need to think about this in a way that allows us to combine results from simpler problems to find the result. For the base case, we need to identify a case so simple we already know the answer. Consider the case when *low* and *high* are equal. Then, there is only one value to use, and we know the value of the maximum is (*f low*). So, the base case is (**if** (= *low high*) (*f low*) ...).

How do we make progress towards the base case? Suppose the value of *high* is equal to the value of *low* plus 1. Then, the maximum value is either the value of (*f low*) or the value of (*f* (+ *low* 1)). We could select it using the *max* procedure:

```
(max (f low) (f (+ low 1)))
```

Of course, we can extend this to the case where *high* is equal to the value of *low* plus 2:

```
(max (f low) (max (f (+ low 1)) (f (+ low 2))))
```

The second operand for the outer *max* evaluation is the maximum value of the input procedure between the low value plus one and the high value input. If we name the procedure we are defining *find-maximum*, then this is the result of (*find-maximum f* (+ *low* 1) *high*). This works whether *high* is equal to (+ *low* 1), or (+ *low* 2), or any other value greater than *high*! Putting things together, we have our recursive definition of *find-maximum*:

```
(define (find-maximum f low high)
  (if (= low high)
    (f low)
    (max (f low)
        (find-maximum f (+ low 1) high)))))
```

Here are a few examples:

```
> (find-maximum (lambda (x) x) 1 20)
20
> (find-maximum (lambda (x) (- 10 x)) 1 20)
9
> (find-maximum (lambda (x) (* x (- 10 x))) 1 20)
25
```

Exercise 4.7. [⋆] To find the maximum of a continuous function, we need to evaluate at all numbers in the range, not just the integers. There are infinitely many numbers between any two numbers, however, so this is impossible. We can approximate this, however, by evaluating the function at many numbers in the range.

Define a procedure *find-maximum-continuous* that takes as input a function *f*, a low range value *low*, a high range value *high*, and an increment *inc*, and produces as output the maximum value of *f* in the range between *low* and *high* where *f* is evaluated at *low*, (+ *low inc*), (+ *low inc inc*), . . ., *high*.

As the value of increment decreases, we expect to find a more accurate maximum value. For example,

```
(find-maximum-continuous (lambda (x) (* x (- 5.5 x))) 1 10 1)
```

should evaluate to 7.5. And,

(*find-maximum-continuous* (**lambda** (*x*) (∗ *x* (− 5.5 *x*))) 1 10 0.0001)

should evaluate to 7.5625.

Exercise 4.8. [⋆] The *find-maximum* procedure we defined evaluates to the maximum value of the input function in the range, but does not provide the input value that produces that maximum output value. Define a procedure that finds the input in the range that produces the maximum output value.

Exercise 4.9. [⋆] Define a *find-area* procedure that takes as input a function *f*, a low range value *low*, a high range value *high*, and an increment *inc*, and produces as output an estimate for the area under the curve produced by the function *f* between *low* and *high* using the *inc* value to determine how many points to evaluate.

Example 4.3: Euclid's Algorithm. In Book 7 of the *Elements*, Euclid describes an algorithm for finding the greatest common divisor of two non-zero integers. The greatest common divisor is the greatest integer that divides both of the input numbers without leaving any remainder. For example, the greatest common divisor of 150 and 200 is 50 since (/ 150 50) evaluates to 3 and (/ 200 50) evaluates to 4, and there is no number greater than 50 which can divide both 150 and 200 without leaving a remainder.

The *modulo* primitive procedure takes two integers as its inputs and evaluates to the remainder when the first input is divided by the second input. For example, (*modulo* 6 3) evaluates to 0 and (*modulo* 7 3) evaluates to 1.

Euclid's algorithm stems from two properties of integers:

1. If (*modulo a b*) evaluates to 0 then *b* is the greatest common divisor of *a* and *b*.
2. If (*modulo a b*) evaluates to a non-zero integer *r*, then the greatest common divisor of *a* and *b* is the greatest common divisor of *b* and *r*.

We can define a recursive procedure for finding the greatest common divisor closely following Euclid's algorithm:

```
(define (gcd a b)
  (if (= (modulo a b) 0)
      b
      (gcd b (modulo a b))))
```

The structure of the definition is similar to the *factorial* definition: the procedure body is an if-expression and the predicate tests for the base case. For the *gcd* procedure, the base case corresponds to the first property above. It

occurs when b divides a evenly, and the consequent expression is b. The alternate expression, (*gcd b* (*modulo a b*)), is the recursive application.

It differs from the alternate expression in the *factorial* definition in that there is no outer application expression (the $*$ application). We do not need to combine the result of the recursive application with some other value as was done in the *factorial* definition, the result of the recursive application is the final result. Unlike the *factorial* and *find-maximum* examples, the *gcd* procedure produces the result in the base case, and no further computation is necessary to produce the final result. When no further evaluation is necessary to get from the result of the recursive application to the final result, a recursive definition is said to be *tail recursive*. *tail recursive*

Exercise 4.10. Show the structure of the applications of *gcd* in evaluating (*gcd* 6 9).

Exercise 4.11. [⋆] Provide a convincing argument why the evaluation of (*gcd a b*) will always finish when the inputs are both positive integers.

Exercise 4.12. [⋆] Provide an alternate definition of *factorial* that is tail recursive. To be tail recursive, the expression containing the recursive application cannot be part of another application expression.

Hint: define a *factorial-helper* procedure that takes an extra parameter, and then define *factorial* as:

(**define** (*factorial n*) (*factorial-helper n* 1))

Exercise 4.13. [⋆] Provide an alternate definition of *find-maximum* that is tail recursive.

Exercise 4.14. [⋆⋆] Provide a convincing argument why it is always possible to transform a recursive procedure into an equivalent procedure that is tail recursive.

Example 4.4: Square Roots. One of the earliest known algorithms is a method for computing square roots. It is known as Heron's method after the Greek mathematician Heron of Alexandria who lived in the first century AD who described the method, although it was also known to the Babylonians many centuries earlier. Issac Newton developed a more general method for estimating functions based on their derivatives known as Netwon's method, of which Heron's method is a specialization.

Square root is a mathematical function that take a number, a, as input and outputs a value x such that $x^2 = a$. For many numbers (for example, 2), the square root is irrational, so the best we can hope for with is a good approx-

Heron of Alexandria

imation. In this example, we will define a procedure *find-sqrt* that takes the target number as input and outputs an approximation for its square root.

Heron's method works by starting with an arbitrary guess, g_0. Then, with each iteration, compute a new guess (g_n is the n^{th} guess) that is a function of the previous guess (g_{n-1}) and the target number (a):

$$g_n = \frac{g_{n-1} + \frac{a}{g_{n-1}}}{2}$$

As n increases, g_n will get closer and closer to the square root of a.

The definition is recursive since we compute g_n as a function of g_{n-1}, so we can define a recursive procedure that computes Heron's method. First, we define a procedure for computing the next guess from the previous guess and the target:

```
(define (heron-next-guess a g)
  (/ (+ g (/ a g)) 2))
```

Next, we define a recursive procedure to compute the n^{th} guess using Heron's method. It takes three inputs: the target number, a, the number of guesses to make, n, and the value of the first guess, g.

```
(define (heron-method a n g)
  (if (= n 0)
      g
      (heron-method a (- n 1) (heron-next-guess a g))))
```

To start, we need a value for the first guess. The choice doesn't really matter — the method works with any starting guess (but will reach a closer estimate quicker if the starting guess is good). We will use 1 as our starting guess. So, we can define a *find-sqrt* procedure that takes two inputs, the target number and the number of guesses to make, and outputs an approximation of the square root of the target number.

```
(define (find-sqrt a guesses)
  (heron-method a guesses 1))
```

Heron's method converges to a good estimate very quickly. For example,

```
> (square (find-sqrt 2 0))
1
> (square (find-sqrt 2 1))
2 1/4
> (square (find-sqrt 2 2))
```

```
2 1/144
> (square (find-sqrt 2 3))
2 1/166464
> (square (find-sqrt 2 4))
2 1/221682772224
> (square (find-sqrt 2 5))
2 1/393146012008229658338304
> (exact->inexact (find-sqrt 2 5))
1.4142135623730951
```

The actual square root of 2 is 1.414213562373095048 . . . so our estimate is correct to 16 digits after only five guesses.

Users of square roots don't really care about the method used to find the square root (or how many guesses are used). Instead, what is important to a square root user is how accurate the estimate is. Can we change our *find-sqrt* procedure so that instead of taking the number of guesses to make as its second input it takes a minimum tolerance value?

Since we don't know the actual square root value (otherwise, of course, we could just return that), we need to measure tolerance as how close the square of the approximation is to the target number. Hence, we can stop when the square of the guess is close enough to the target value.

```
(define (close-enough? a tolerance g)
  (<= (abs (- a (square g))) tolerance))
```

The stopping condition for the recursive definition is now when the guess is close enough. Otherwise, our definitions are the same as before.

```
(define (heron-method-tolerance a tolerance g)
  (if (close-enough? a tolerance g)
      g
      (heron-method-tolerance a tolerance (heron-next-guess a g))))


(define (find-sqrt-approx a tolerance)
  (heron-method-tolerance a tolerance 1))
```

Note that the value passed in as *tolerance* does not change with each recursive call. We are making the problem smaller by making each successive guess closer to the required answer.

Here are some example interactions with *find-sqrt-approx*:

> (*exact->inexact* (*square* (*find-sqrt-approx* 2 0.01)))
2.0069444444444446
> (*exact->inexact* (*square* (*find-sqrt-approx* 2 0.0000001)))
2.000000000004511

Excursion 4.1: Square Roots. Scheme provides a built-in *sqrt* procedure. How accurate is it? Can you produce more accurate square roots than the built-in *sqrt* procedure? (Why doesn't the built-in procedure do better?)

4.4 Evaluating Recursive Applications

Evaluating an application of a recursive procedure follows the evaluation rules just like any other expression evaluation. It may be confusing, however, to see that this works because of the apparent circularity of the procedure definition. Here, we show in detail the evaluation steps for evaluating (*factorial* 2). The evaluation and application rules refer to the rules summary in Section 3.8. We first show the complete evaluation following the substitution model evaluation rules in full gory detail, and later consider a subset containing the most revealing steps. Stepping through even a fairly simple evaluation using the evaluation rules is quite tedious, and not something humans should do very often (that's what we have computers for!) but instructive to do once to understand exactly how an expression is evaluated.

The evaluation rule for an application expression does not specify the order in which the subexpressions are evaluated. A Scheme interpreter is free to evaluate them in any order. Here, we choose to evaluate the subexpressions in the order that is most readable. As long as none of the expressions have side effects, the value produced by an evaluation does not depend on the order in which the subexpressions are evaluated.

In the evaluation steps, we use `typewriter` font for uninterpreted Scheme expressions and sans-serif font to show values. So, `2` represents the Scheme expression that evaluates to the number 2.

1. `(factorial 2)`
   Evaluation Rule 3(a): Application subexpressions
2. `(factorial 2)`
   Evaluation Rule 2: Name
3. `((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 2)`
   Evaluation Rule 4: Lambda
4. ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 2)

Evaluation Rule 1: Primitive

5. `((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 2)`

Evaluation Rule 3(b): Application, Application Rule 2

6. `(if (= 2 0) 1 (* 2 (factorial (- 2 1))))`

Evaluation Rule 5(a): If predicate

7. `(if (= 2 0) 1 (* 2 (factorial (- 2 1))))`

Evaluation Rule 3(a): Application subexpressions

8. `(if (= 2 0) 1 (* 2 (factorial (- 2 1))))`

Evaluation Rule 1: Primitive

9. `(if (= 2 0) 1 (* 2 (factorial (- 2 1))))`

Evaluation Rule 3(b): Application, Application Rule 1

10. `(if false 1 (* 2 (factorial (- 2 1))))`

Evaluation Rule 5(b): If alternate

11. `(* 2 (factorial (- 2 1)))`

Evaluation Rule 3(a): Application subexpressions

12. `(* 2 (factorial (- 2 1)))`

Evaluation Rule 1: Primitive

13. `(* 2 (factorial (- 2 1)))`

Evaluation Rule 3(a): Application subexpressions

14. `(* 2 (factorial (- 2 1)))`

Evaluation Rule 3(a): Application subexpressions

15. `(* 2 (factorial (- 2 1)))`

Evaluation Rule 1: Primitive

16. `(* 2 (factorial (- 2 1)))`

Evaluation Rule 3(b): Application, Application Rule 1

17. `(* 2 (factorial 1))`

Continuing Evaluation Rule 3(a); Evaluation Rule 2: Name

18. `(* 2 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 1))`

Evaluation Rule 4: Lambda

19. `(* 2 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 1))`

Evaluation Rule 3(b): Application, Application Rule 2

20. `(* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))`

Evaluation Rule 5(a): If predicate

21. `(* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))`

Evaluation Rule 3(a): Application subexpressions

22. `(* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))`

Evaluation Rule 1: Primitives

23. `(* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))`

Evaluation Rule 3(b): Application Rule 1

24. `(* 2 (if false 1 (* 1 (factorial (- 1 1)))))`

Evaluation Rule 5(b): If alternate

25. `(* 2 (* 1 (factorial (- 1 1))))`

Evaluation Rule 3(a): Application

26. `(* 2 (* 1 (factorial (- 1 1))))`
    Evaluation Rule 1: Primitives
27. `(* 2 (* 1 (factorial (- 1 1))))`
    Evaluation Rule 3(a): Application
28. `(* 2 (* 1 (factorial (- 1 1))))`
    Evaluation Rule 3(a): Application
29. `(* 2 (* 1 (factorial (- 1 1))))`
    Evaluation Rule 1: Primitives
30. `(* 2 (* 1 (factorial (- 1 1))))`
    Evaluation Rule 3(b): Application, Application Rule 1
31. `(* 2 (* 1 (factorial 0)))`
    Evaluation Rule 2: Name
32. `(* 2 (* 1 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 0)))`
    Evaluation Rule 4, Lambda
33. `(* 2 (* 1 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 0)))`
    Evaluation Rule 3(b), Application Rule 2
34. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`
    Evaluation Rule 5(a): If predicate
35. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`
    Evaluation Rule 3(a): Application subexpressions
36. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`
    Evaluation Rule 1: Primitives
37. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`
    Evaluation Rule 3(b): Application, Application Rule 1
38. `(* 2 (* 1 (if true 1 (* 0 (factorial (- 0 1))))))`
    Evaluation Rule 5(b): If consequent
39. `(* 2 (* 1 1))`
    Evaluation Rule 1: Primitives
40. `(* 2 (* 1 1))`
    Evaluation Rule 3(b): Application, Application Rule 1
41. `(* 2 1)`
    Evaluation Rule 3(b): Application, Application Rule 1
42. **2**
    Evaluation finished, no unevaluated expressions remain.

The key to evaluating applications of recursive procedures is the evaluation rule for the if special form. If the if-expression were evaluated like a regular application all subexpressions would be evaluated, and the alternative expression with the recursive call would never finish evaluating! Since the evaluation rule for if requires that the predicate expression is evaluated and the alternative expression is *not* evaluated when the predicate expression is true, the circularity in the definition ends when the predicate expression

evaluates to true. This is the base case of the recursion, when (= *n* 0) evaluates to true. This occurs when (*factorial* 0) is evaluated, and instead of producing another recursive call it evaluates to the value of the consequent expression, 1.

Exercise 4.15. These exercises test your understanding of the (*factorial* 2) evaluation. Try to answer them yourself before reading the remainder of this section.

a. In step 5, the second part of the application evaluation rule, Rule 3(b), is used. In which step does this evaluation rule complete?

b. In step 11, the first part of the application evaluation rule, Rule 3(a), is used. In which step is the following use of Rule 3(b) started?

c. In step 25, the first part of the application evaluation rule, Rule 3(a), is used. In which step is the following use of Rule 3(b) started?

d. How many times is Evaluation Rule 5 (If) used?

e. [⋆] In order to evaluate (*factorial* 3) how many times would Evaluation Rule 2 be used to evaluate the name *factorial*?

Exercise 4.16. For which input values *n* will an evaluation of (*factorial n*) eventually reach a value? For values where the evaluation is guaranteed to finish, make a convincing argument why it must finish. For values where the evaluation does not finish, explain why.

4.4.1 The Evaluation Stack

We can see the structure of the evaluation process better if we focus just on the most revealing stems. Here are the evaluation steps that involve applications of the *factorial* procedure extracted from the full evaluation and indented to line up the steps from each application evaluation:

```
1.  (factorial 2)
17.     (* 2 (factorial 1))
31.         (* 2 (* 1 (factorial 0)))
40.         (* 2 (* 1 1))
41.     (* 2 1)
42. 2
```

In Step 1, we start to evaluate (*factorial* 2). The final result of this evaluation is found in Step 42. To evaluate (*factorial* 2), we follow the evaluation rules,

eventually reaching the body expression of the if-expression in the factorial definition. This is Step 17: (∗ 2 (*factorial* 1)). To evaluate this expression, we need to evaluate the (*factorial* 1) subexpression.

We can think of each of these application evaluations as a stack, similarly to how we processed RTN subnetworks in Section 2.4.1. A stack has the property that the first item pushed on the stack will be the last item removed—all the items pushed on top of this one must be removed before this item can be removed. For application evaluations, the elements on the stack are expressions to evaluate. To finish evaluating the first expression, all of its component subexpressions must be evaluated. Hence, the first application evaluation started will be the last one to finish.

At Step 17, the first evaluation is in progress, but to complete it we need the value resulting from the second evaluation. The second evaluation results in the body expression, (∗ 1 (*factorial* 0)), shown for Step 31. At this point, the evaluation of (*factorial* 2) is stuck in Evaluation Rule 3, waiting for the value of (*factorial* 1) subexpression. The evaluation of the (*factorial* 1) application leads to the (*factorial* 0) subexpression, which must be evaluated before the (*factorial* 1) evaluation can complete. In Step 40, the (*factorial* 0) subexpression evaluation has completed and produced the value 1. Now, the (*factorial* 1) evaluation can complete, producing 1 as shown in Step 41. Once the (*factorial* 1) evaluation completes, all the subexpressions needed to evaluate the expression in Step 17 are now evaluated, and the evaluation completes in Step 42.

4.5 Developing Complex Programs

To develop and use more complex procedures it will be useful to learn a few techniques that will help you understand what is going on when your procedures are evaluated (and fix problems when things are not working as intended). It is very rare for a first attempt to create a program to be completely correct, even for an expert programmer. Instead, it is best to build programs incrementally, by building and testing small components one at a time.

debugging

The process of fixing broken programs is known as *debugging*. The name comes from the early days of computing when real bugs in the machine could cause problems. The key to debugging effectively is to be systematic and thoughtful. It is a good idea to take notes to keep track of what you have learned and what you have tried. Thoughtless debugging can be very frustrating, and is unlikely to lead to a correct program.

Moth from Grace Hopper's notebook, 1947

A good strategy for debugging is to:

1. First, make sure you understand the intended behavior of your procedure. Think of a few representative inputs, and what the expected output should be.
2. Then, do experiments to observe the actual behavior of your procedure. Does it work correctly for some inputs but not others? What is the relationship between the actual outputs and the desired outputs?
3. Next, make changes to your procedure and retest it. If you are not sure what to do, make changes in small steps and carefully observe the impact of each change.

As your programs get more complex, you'll want to follow this strategy but at the level of sub-components. For example, you can try debugging at the level of one expression before trying the whole procedure. If you break your program down into several procedures, you can test and debug each procedure independently. The smaller the unit you are considering at one time, the easier it will be to understand what the problem is and how to fix it.

DrScheme provides many useful and powerful features to aid debugging, although the most useful tool there is for debugging is using your brain to think carefully about what your program should be doing and how its observed behavior differs from the desired behavior. Next, we describe two simple techniques that are helpful for observing program behavior.

4.5.1 Printing

One useful procedure built-in to DrScheme is the *display* procedure. It takes one input, and produces no output. Instead of producing an output, it prints out the value of the input (it will appear in purple in the Interactions window). We can use *display* to observe what a procedure is doing before it produces an output. For example, if we add a (*display n*) expression at the beginning of our *factorial* procedure we can see all the intermediate calls. To make each printed value appear on a separate line, we use the *newline* procedure. The *newline* procedure takes not inputs and produces no output, but prints out a new line.

```
(define (factorial n)
  (display "Enter factorial: ") (display n) (newline)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Now, evaluating (*factorial* 3) produces:

```
Enter factorial: 3
Enter factorial: 2
Enter factorial: 1
Enter factorial: 0
6
```

For printing out lots of strings and values, the built-in *printf* procedure is more useful. The *printf* procedure takes one or more inputs. The first input is a string (a sequence of characters enclosed in double quotes). The string can include special `~a` markers that print out values of objects inside the string. Each `~a` marker is matched with a corresponding input, and the value of that input is printed in place of the `~a` in the string. Another special marker, `~n`, prints out a new line inside the string. Using *printf*, we can define our *factorial* procedure with printing as:

```
(define (factorial n)
  (printf "Enter factorial: ~a~n" n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

side effects

Note that *display*, *printf*, and *newline* do not produce output values. Instead, they are applied to produce *side effects*. A side effect is something that changes the state of a computation. In this case, the side effect is printing in the Interactions window. Side effects make reasoning about what programs do much more complicated since the order in which events happen now matters. We will mostly avoid using procedures with side effects until Chapter 11, but printing procedures are so useful that we introduce them here.

4.5.2 Tracing

DrScheme provides a more automated way to observe applications of procedures. We can use tracing to observe the beginning of a procedure evaluation (including the procedure inputs), and when the evaluation complete (including the procedure outputs). To use tracing, it is necessary to first load the tracing library by evaluating this expression:

```
(require (lib "trace.ss"))
```

This defines the *trace* procedure that takes one input, a compound procedure (trace does not work for primitive procedures). After evaluating (*trace proc*), every time an application of *proc* is evaluated the tracing will print out the procedure name and its inputs at the beginning of the application evaluation, and the value of the output at the end of the application evaluation. If there are other applications before the first application finishes evaluating, these will be printed too, but indented so it is possible to match up the beginning and end of each application evaluation. For example (the trace outputs are shown in `typewriter` font),

```
> (trace factorial)
> (factorial 3)
|(factorial 3)
| (factorial 2)
| |(factorial 1)
| | (factorial 0)
| | 1
| |1
| 2
|6
6
```

From the trace, one can see that (*factorial* 3) is evaluated first; within its evaluation, (*factorial* 2), then (*factorial* 1), and (*factorial* 0) are evaluated. The outputs of each of these applications is lined up vertically below the application entry trace.

Excursion 4.2: Recipes for π. The value π is the defined as the ratio between the circumference of a circle and its diameter. One way to calculate the value of π is the Gregory-Leibniz series (which was actually discovered by the Indian mathematician Mādhava in the 14^{th} century):

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$$

This summation converges to π. The more terms that are included, the closer the computed value will be to the actual value of π.

a. [⋆] Define a procedure *compute-pi* that takes as input n, the number of terms to include and outputs an approximation of π computed using the first n terms of the Gregory-Leibniz series. For example, (*compute-pi* 1) should evaluate to 4 and (*compute-pi* 2) should evaluate to 2 2/3 ($= \frac{4}{1} - \frac{4}{3}$). For higher terms, use the built-in procedure *exact->inexact* to see the decimal value. For example (*exact->inexact* (*compute-pi* 10))

should evaluate to 3.0418396189294024 and (*exact->inexact* (*compute-pi* 10000)) evaluates (after a long wait!) to 3.1414926535900434.

The Gregory-Leibniz series is fairly simple, but it takes an awful long time to converge to a good approximation for π — only the first digit is correct after 10 terms, and after summing 10000 terms only the first four digits are correct.

Mādhava discovered another series for computing the value of π that converges much more quickly:

$$\pi = \sqrt{12} * (1 - \frac{1}{3*3} + \frac{1}{5*3^2} - \frac{1}{7*3^3} + \frac{1}{9*3^4} - \ldots)$$

Mādhava computed the first 21 terms of this series, finding an approximation of π that is correct for the first 12 digits: 3.14159265359.

b. [⋆⋆] Define a procedure *cherry-pi* that takes as input n, the number of terms to include and outputs an approximation of π computed using the first n terms of the faster Madhava series. (Continue reading for hints.)

To define *cherry-pi*, you will first want to define two helper functions: *cherry-pi-filling*, that takes one input, n, and computes the sum of the first n terms in the series without the $\sqrt{12}$ factor, and *cherry-pi-term* that takes one input n and computes the value of the n^{th} term in the series (without alternating the adding and subtracting). For example, (*cherry-pi-term* 1) should evaluate to 1 and (*cherry-pi-term* 2) should evaluate to 1/9. Then, you can define *cherry-pi* as:

(**define** (*cherry-pi terms*) (∗ (*sqrt* 12) (*cherry-pi-helper terms*)))

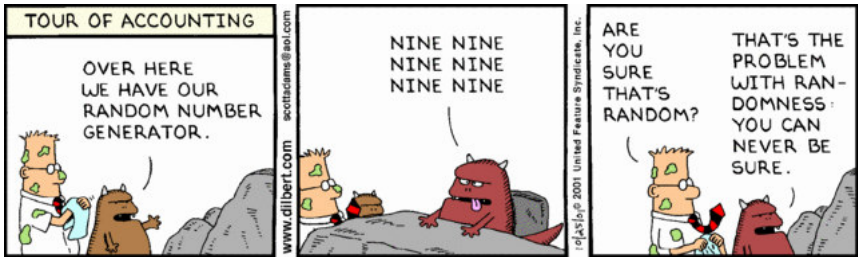
Here, we use the built-in *sqrt* procedure that takes one input and produces as output an approximation of its square root. The accuracy of the *sqrt* procedure[5] limits the number of digits of π that can be correctly computed using this method (see Example 4 for ways to compute a more accurate approximation for the square root of 12). You should be able to get a few more correct digits than Mādhava was able to get without a computer 600 years ago, but to get more digits would need a more accurate *sqrt* procedure or another method for computing π.

To compute the powers $(3^1, 3^2, 3^3, \ldots)$, you can use the built-in *expt* procedure (which takes two inputs, a and b, and produces a^b as its output), but

[5] To test its accuracy, try evaluating (*square* (*sqrt* 12)).

you should also be able to figure out a way to define your own procedure to compute a^b for any integer inputs a and b.

c. [⋆⋆⋆] Find a procedure for computing enough digits of π to find the *Feynman point* where there are six consecutive 9 digits. This point is name for Richard Feynman, who quipped that he wanted to memorize π to that point so he could recite it as "... nine, nine, nine, nine, nine, nine, and so on".



Producing good random numbers is an important and interesting problem, as is the question of determining whether a given sequence of numbers is random. We consider it in Chapter 19.

Excursion 4.3: Recursive Definitions and Games. Many games can be analyzed by thinking recursively. For this excursion, we consider how to develop a winning strategy for some two-player games. In all the games, we assume player 1 moves first, and the two players take turns until the game ends. The game ends when the player who's turn it is cannot move; the other player wins.

A strategy is a *winning strategy* if it provides a way to always select a move that wins the game, regardless of what the other player does. A good approach for thinking about these games is to work backwards from the winning position (which corresponds to the base case in a recursive definition). If the game reaches a winning position for player 1, then player 1 is guaranteed to win. Moving back one move, if the game reaches a position where it is player 2's move, but all possible moves lead to a winning position for player 1, then player 1 is guaranteed to win. Continuing backwards, if the game reaches a position where it is player 1's move, and there is a move that leads to a position where all possible moves for player 2 lead to a winning position for player 1, then player 1 is guaranteed to win.

The first game we will consider is called *Nim*. Variants on Nim have been played widely over many centuries, and no one is quite sure where the name comes from.

We'll start with a simple variation on the game that was called *Thai 21* when it was used as an Immunity Challenge on *Survivor*. In this version of Nim,

the game starts with a pile of 21 stones. One each turn, a player removes one, two, or three stones. The player who removes the last stone wins, since the other player cannot make a valid move on the following turn.

a. [⋆] What should the player who moves first do to ensure she can always win the game? (Hint: start with the base case, and work backwards. Think about a game starting with 5 stones first, before trying 21.)

b. [⋆] Suppose instead of being able to take 1–3 stones with each turn, you can take 1–n stones where n is some number greater than or equal to 1. For what values of n should the first player always win (when the game starts with 21 stones)?

c. [⋆] Define a procedure that plays a winning game of *Thai 21*. Your procedure should take one input, representing the number of stones left, and produce as output a number indicating the number of sticks to take. If there is no winning strategy from the given input number of sticks, your procedure should output `0`.

d. [⋆] Generalize your procedure to support the version of the game where the maximum number of stones that can be removed is a parameter. The new procedure should take two inputs, the first is the number of stones left, and the second is the maximum number of stones that can be removed on each turn, and produce as output the number of stones to remove.

In the standard Nim, instead of just having one heap of stones, the game starts with three heaps. At each turn, a player removes any number of stones from any one heap (but may not remove stones from different heaps). We can describe the state of a 3-heap game of Nim using three numbers, representing the number of stones in each heap. For example, the *Thai 21* game starts with the state `(21 0 0)` (one heap with 21 stones, and two empty heaps). (With the standard Nim rules, this would not be an interesting game since the first player can simply win by removing all 21 stones from the first heap.) (If you get stuck, you'll find many resources about Nim on the Internet; but, you'll get a lot more out of this if you try and solve it yourself.)

e. What should the first player do to win if the starting state is `(2 1 0)`?

f. Which player should win if the starting state is `(2 2 1)`?

g. Which player should win if the starting state is `(2 2 2)`?

h. [⋆] Which player should win if the starting state is `(5 6 7)`?

i. [⋆⋆] Describe a strategy for always winning a game of Nim starting from any position.

The final game we consider is the "Corner the Queen" game invented by Rufus Isaacs (and described by Martin Gardner [Gar97]). The game is played using a single Queen on a arbitrarily large chessboard as shown in Figure 4.5.

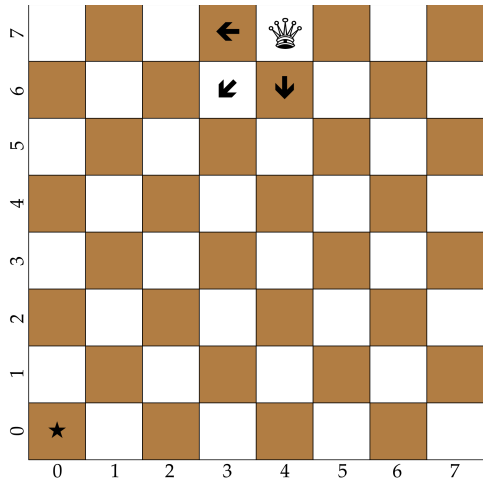


Figure 4.5. Cornering the Queen.

On each turn, a played moves the Queen one or more squares in either the left, down, or diagonally down-left direction (unlike a standard chess Queen, in this game the queen may not move right, up or up-right). As with the other games, the last player to make a legal move wins. For this game, once the Queen reaches the bottom left square marked with the $\star$, there are no moves possible. Hence, the player who moves the Queen onto the $\star$ wins the game. We name the squares using the numbers on the sides of the chessboard with the column number first. So, the Queen in the picture is on square (4 7).

j. Identify all the starting squares for which the first played to move can win right away. (Your answer should generalize to any size square chessboard.)

k. Suppose the Queen is on square (2 1) and it is your move. Explain why there is no way you can avoid losing the game.

l. If the Queen is on square (5 6) and it is your move, how should you move to guarantee you will win the game?

m. [$\star$] Given the shown starting position (with the Queen at (4 7), would you rather be the first or second player?

n. [$\star$] Describe a strategy for winning the game (when possible). Explain from which starting positions it is not possible to win (assuming the other player always makes the right move).

o. [⋆] Define a variant of Nim that is essentially the same as the "Corner the Queen" game. (The game is known as Wythoff's Nim.)

4.6 Summary

By breaking problems down into simpler problems we can develop solutions to complex problems. Many problems can be solved by combining instances of the same problem on simpler inputs. When we define a procedure to solve a problem this way, it needs to have a predicate expression to determine when the base case has been reached, a consequent expression that provides the value for the base case, and an alternate expression that defines the solution to the given input as an expression using a solution to a slightly smaller input.

Our general problem solving strategy is:

1. Be optimistic! Assume you can solve it.
2. Think of the simplest version of the problem, something you can already solve. This is the base case.
3. Consider how you would solve a big version of the problem by using the result for a slightly smaller version of the problem. This is the recursive case.
4. Combine the base case and the recursive case to solve the problem.

For problems involving numbers, the base case will often be when the input value is zero (but not always, as we saw in the *find-maximum* value, where the base case is reached when the difference between two of the input values is zero). The way the problem size is reduced is by subtracting some value (usually 1) from one of the inputs.

In the next chapter, we introduce more complex data structures. For problems involving complex data, the same strategy will often work, but we will use other ways to identify a base case and to shrink the problem size.

5 Data

From a bit to a few hundred megabytes, from a microsecond to half an hour of computing confronts us with the completely baffling ratio of 10^9*! .... By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history.*

Edsger Dijkstra

For all the programs so far, we have been limited to simple data such as numbers and Booleans. We call this *scalar* data since it has no structure. As we saw in Chapter 1, we can represent all discrete data using just (enormously large) whole numbers. For example, we could represent the text of a book using only one (very large!) number, and manipulate the characters in the book by changing the value of that number. But, it would be very difficult to design and understand computations if we only use numbers to represent data.

Instead, we need more complex data structures that allow us to model the structured data our problem concerns with structures our programs can more directly manipulate. We want to represent data in ways that allow us to think about the problem we are trying to solve, rather than the low-level details of how data is represented and manipulated.

In this chapter, we develop techniques for building complex data structures and for defining procedures that manipulate structured data, and introduce data abstraction as a tool for managing program complexity.

5.1 Types

All data in a program has an associated type. Internally, all data is stored just as a sequence of bits, so the type of the data is important to understand what it means. We have seen several different types of data already: Numbers, Booleans, and Procedures (we use initial capital letters to signify a datatype).

A datatype defines a set (often infinite) of possible values. The Boolean

datatype contains the two Boolean values, true and false. The Number type includes the infinite set of all whole numbers (it also includes negative numbers and rational numbers). We think of the set of possible Numbers as infinite, even though on any particular computer there is some limit to the amount of memory available, and hence, some largest number that can be represented. On any real computer, the number of possible values of any data type is always finite. But, we can imagine a computer large enough to represent any given number.

The type of data determines what can be done with it. For example, a Number can be used as one of the inputs to the primitive procedures $+$, $*$, and $=$. A Boolean can be used as the first sub-expression of an if-expression and as the input to the *not* procedure (note that the way *not* is defined, it can also take a Number as its input, but for all Number value inputs the output is false), but cannot be used as the input to $+$, $*$, or $=$.[1]

A Procedure can be the first sub-expression in an application expression, and can be passed as an input to the *fcompose* procedure (defined in the previous chapter). There are infinitely many different types of Procedures, since the type of a Procedure depends on its input and output types. For example, recall *bigger* procedure from Chapter 3:

(**define** (*bigger a b*) (**if** ($>$ *a b*) *a b*))

It takes two Numbers as input and produces a Number as output. We denote this type as:

$$\text{Number} \times \text{Number} \rightarrow \text{Number}$$

The inputs to the procedure are shown on the left side of the arrow. The type of each input is shown in order, separated by the $\times$ symbol.[2] The output type is given on the right side of the arrow.

From its definition, we know the *bigger* procedure takes two inputs. But how do we know the inputs must be Numbers and the output is a Number?

The body of the *bigger* procedure is an if-expression with the predicate expression ($>$ *a b*). This applies the $>$ primitive procedure to the two inputs.

[1] The primitive procedure *equal?* is a more general comparison procedure that can take as inputs any two values, so could be used to compare Boolean values. For example, (*equal?* false false) evaluates to true and (*equal?* true 3) is a valid expression that evaluates to false.

[2] The notation using $\times$ to separate input types makes sense if you think about the number of different inputs to a procedure. For example, consider a procedure that takes two Boolean values as inputs, so its type is Boolean $\times$ Boolean $\rightarrow$ Value. Each Boolean input can be one of two possible values. If we combined both inputs into one input, there would be 2×2 different values needed to represent all possible inputs.

The type of the $>$ procedure is Number $\times$ Number $\rightarrow$ Boolean. So, for the predicate expression to be valid, its inputs must both be Numbers. This means the input values to *bigger* must both be Numbers. We know the output of the *bigger* procedure will be a Number by analyzing the consequence and alternate sub-expressions: each evaluates to one of the input values, which must be a Number.

Starting with the primitive Boolean, Number, and Procedure types, we can build arbitrarily complex datatypes. In this chapter, we will introduce mechanisms for building complex datatypes by combining the primitive datatypes.

Exercise 5.1. Describe the type of each of these expressions.

a. 17

b. (**lambda** (*a*) ($>$ *a* 0))

c. ((**lambda** (*a*) ($>$ *a* 0)) 3)

d. (**lambda** (*a*) (**lambda** (*b*) ($>$ *a* *b*)))

e. (**lambda** (*a*) *a*)

Exercise 5.2. For each part, define or identify a procedure that has the given type.

a. Number $\times$ Number $\rightarrow$ Boolean

b. Number $\rightarrow$ Number

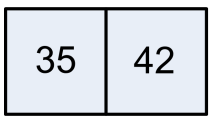
c. (Number $\rightarrow$ Number) $\times$ (Number $\rightarrow$ Number) $\rightarrow$ (Number $\rightarrow$ Number)

d. Number $\rightarrow$ (Number $\rightarrow$ (Number $\rightarrow$ Number))

5.2 Pairs

The simplest structured data construct is a *Pair*. A Pair packages two values together. We draw a Pair as two boxes, each containing a value. We call each box of a Pair a *cell*. Here is a Pair where the first cell has the value 35 and the second cell has the value 42:

Pair



Scheme provides built-in procedures for constructing a Pair, and for extracting each cell from a Pair:

- *cons*: Value × Value → Pair — evaluates to a Pair where the first cell is the first input value and the second cell is the second input value. The Value inputs can be of any type.
- *car*: Pair → Value — evaluates to the first cell of the input, which must be a Pair.
- *cdr*: Pair → Value — evaluates to the second cell of input, which must be a Pair.

These rather unfortunate names come from the original LISP implementation on the IBM 704. The name *cons* is short for "construct". The name *car* is short for "*C*ontents of the *A*ddress part of the *R*egister" and the name *cdr* (pronounced "could-er") is short for "*C*ontents of the *D*ecrement part of the *R*egister". The designers of the original LISP implementation picked the names because of how pairs could be implemented on the IBM 704 using a single register to store both parts of a pair, but it is a mistake to name things after details of their implementation (see Section 5.6). Unfortunately, the names stuck and continue to be used in many LISP-derived languages, including Scheme.

We can construct the Pair shown in the previous diagram by evaluating (*cons* 35 42). DrScheme will display a Pair by printing the value of each cell separated by a dot: (35 . 42). The interactions below show some example uses of *cons*, *car*, and *cdr*.

```
> (define mypair (cons 35 42))
> mypair
(35 . 42)
> (car mypair)
35
> (cdr mypair)
42
```

The values in the cells of a Pair can be any values, including other Pairs!

For example,

```
(define doublepair (cons (cons 1 2) (cons 3 4)))
```

defines a Pair where each cell of the Pair is itself a Pair.

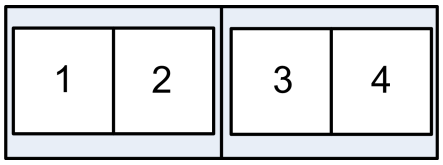
We can use the *car* and *cdr* procedures to access components of our nested *doublepair* structure: (*car doublepair*) evaluates to the Pair (1 . 2), and (*cdr doublepair*) evaluates to the Pair (3 . 4).

We can compose multiple *car* and *cdr* applications to extract components from the nested pairs:

```
> (cdr (car doublepair))
2
> (car (cdr doublepair))
3
> ((fcompose cdr cdr) doublepair)
4
> (car (car (car doublepair)))
car: expects argument of type <pair>; given 1
```

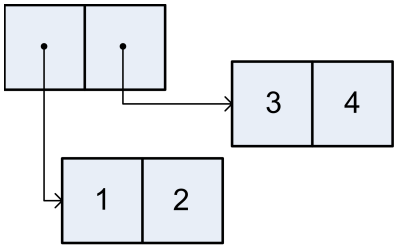
The last expression produces an error when it is evaluated since *car* is applied to the scalar value 1. The *car* and *cdr* procedures can only be applied to an input that is a Pair. Hence, an error results when we attempt to apply *car* to a scalar value. This is an important property of data: the type of data (e.g., a Pair) defines how it can be used (e.g., passed as the input to *car* and *cdr*). Every procedure expects a certain type of inputs, and typically produces an error when it is applied to values of the wrong type.

We can draw the value of *doublepair* by nesting Pairs within cells:

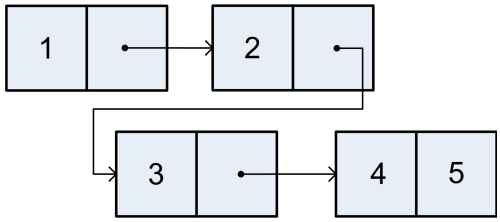


Drawing Pairs within Pairs within Pairs can get quite difficult, however. For example, try drawing (*cons* 1 (*cons* 2 (*cons* 3 (*cons* 4 5)))) this way.

Instead, we us arrows to point to the contents of cells that are not simple values. This is the structure of *doublepair* shown using arrows:



Using arrows to point to cell contents allows us to draw arbitrarily complicated data structures, keeping the cells reasonable sizes. For example, here is one way to draw (*cons* 1 (*cons* 2 (*cons* 3 (*cons* 4 5)))):



Exercise 5.3. Suppose the following definition has been executed:

```
(define tpair
  (cons (cons (cons 1 2) (cons 3 4))
        5))
```

Draw the structure defined by *tpair*, and give the value of each of the following expressions.

a. (*cdr tpair*)

b. (*car* (*car* (*car tpair*)))

c. (*cdr* (*cdr* (*car tpair*)))

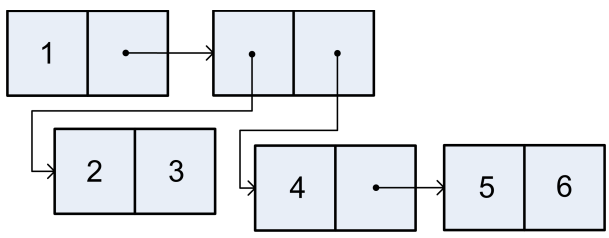
d. (*car* (*cdr* (*cdr tpair*)))

Exercise 5.4. Suppose the following definition has been executed:

```
(define fstruct (cons 1 (cons 2 (cons 3 4))))
```

Write expressions that extract each of the four elements from *fstruct*.

Exercise 5.5. What expression procedures the structure shown below:



5.2.1 Making Pairs

Although Scheme provides the built-in procedures *cons*, *car*, and *cdr* for creating Pairs and accessing their cells, there is nothing magic about these procedures. We could define procedures with the same behavior ourselves using only the subset of Scheme introduced in Chapter 3. For example, here is one way to define the pair procedures (we prepend an *s* to the names to avoid confusion with the built-in procedures):

```
(define (scons a b) (lambda (w) (if w a b)))
(define (scar pair) (pair true))
(define (scdr pair) (pair false))
```

The *scons* procedure takes the two parts of the Pair as inputs, and produces as output a procedure. The output procedure takes one input, a selector that determines which of the two cells of the Pair to output. If the selector is true, the value of the if-expression is the value of the first cell; if false, it is the value of the second cell. The *scar* and *scdr* procedures apply a procedure constructed by *scons* to either true (to select the first cell in *scar*) or false (to select the second cell in *scdr*).

Exercise 5.6. Convince yourself the definitions of *scons*, *scar*, and *scdr* above work as expected by following the evaluation rules to evaluate

```
(scar (scons 1 2))
```

Exercise 5.7. Suppose we define *tcons* as:

```
(define (tcons a b) (lambda (w) (if w b a)))
```

Show the corresponding definitions of *tcar* and *tcdr* needed to provide the correct pair selection behavior for a pair created using *tcons*.

5.2.2 Triples to Octuples

Pairs are useful for representing data that is composed of two parts such as a calendar da (composed of a number and month), or a playing card (composed of a rank and suit). But, what if we want to represent data composed of more than two parts such as a date (composed of a number, month, and year) or a poker hand consisting of five playing cards? For more complex data structures, we want to construct data structures that have more than two components.

A *triple* has three components. Here is one way to define a triple datatype and its accessors:

```
(define (make-triple a b c)
  (lambda (w) (if (= w 0) a (if (= w 1) b c))))

(define (triple-first t)  (t 0))
(define (triple-second t) (t 1))
(define (triple-third t)  (t 2))
```

Since a triple has three components we need three different selector values; we use 0, 1, and 2.

A simpler way to make a triple would be to combine two Pairs. We do this by making a Pair whose second cell is itself a Pair:

```
(define (make-triple a b c) (cons a (cons b c)))
```

Then, we can use the *car* and *cdr* procedures to define procedures for selecting elements of the triple:

```
(define (triple-first t)  (car t))
(define (triple-second t) (car (cdr t)))
(define (triple-third t)  (cdr (cdr t)))
```

Similarly, we can define a *quadruple* as a Pair whose second cell is a triple:

```
(define (make-quad a b c d) (cons a (make-triple b c d)))
(define (quad-first q)  (car q))
(define (quad-second q) (triple-first (cdr q))
(define (quad-third q)  (triple-second (cdr q))
(define (quad-fourth q) (triple-third (cdr q))
```

We could continue in this manner defining increasingly large tuples:

- A *triple* is a Pair whose second cell is a Pair.
- A *quadruple* is a Pair whose second cell is a *triple*.
- A *quintuple* is a Pair whose second cell is a *quadruple*.
- A *sextuple* is a Pair whose second cell is a *quintuple*.
- A *septuple* is a Pair whose second cell is a *sextuple*.
- ...
- A $(+\ n\ 1)$-*uple* is a Pair whose second cell is an *n-uple*.

Hence, building from the simple Pair, we can construct tuples containing any number of components.

Exercise 5.8. Define a procedure that constructs a quintuple and procedures for selecting the five elements of a quintuple.

Exercise 5.9. Another way of thinking of a triple is as a Pair where the first cell is a Pair and the second cell is a scalar. Provide definitions of *make-triple*, *triple-first*, *triple-second*, and *triple-third* for this alternate triple definition.

5.3 Lists

In the previous section, we saw how to construct arbitrarily large tuples building from Pairs. This way of managing data is not very satisfying, however. It requires defining different procedures for constructing and accessing elements of every length tuple. For many applications, we want to be able to manage data of any length such as all the items in a web store, or all the bids on a given item. Since the number of components in these objects can change, it would be very painful to need to define a new tuple type every time an item is added. Hence, we need a data type that can hold any number of items, and a way to define procedures that work on any length tuple.

What we want is a way to construct and manipulate tuples works for tuples containing *any* number of elements. This definition of an *any-uple* almost provides what we need:

> An *any-uple* is a Pair whose second cell is an *any-uple*.

This seems to allow an *any-uple* to contain any number of elements. The problem is we have no stopping point. With only the definition above, there is no way to construct an *any-uple* without already having one.

The situation is similar to defining *MoreDigits* as zero or more digits in Chapter 2, defining *MoreExpressions* in the Scheme grammar in Chapter 3 as zero or more *Expression*s, and recursive composition in Chapter 4.

Recall the grammar rules for *MoreExpressions*:

$$\begin{array}{l} \textit{MoreExpressions} ::\Rightarrow\ \textit{Expression MoreExpressions} \\ \textit{MoreExpressions} ::\Rightarrow\ \epsilon \end{array}$$

The rule for constructing an *any-uple* above corresponds to the first *MoreExpression* replacement rule. To allow an *any-uple* to be constructed, we also need a construction rule similar to the second rule, where *MoreExpression* can be replaced with nothing. Since it is hard to type and read nothing in a program, Scheme has a name for this value: *null*. *null*

DrScheme will print out the value of *null* as (). It is also known as the *empty list*, since it represents the List containing no elements. The built-in procedure *null?* takes one input parameter and evaluates to true if and only if the value of that parameter is null.

List Using null, we can now define a *List*:

> A *List* is either (1) null or (2) a Pair whose second cell is a *List*.

Symbolically, we can define a List as:

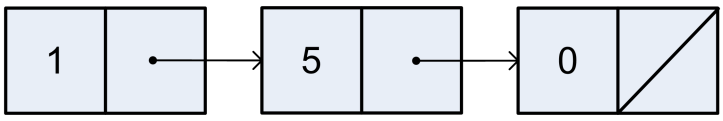
$$\begin{aligned} \textit{List} &::\Rightarrow \ \texttt{null} \\ \textit{List} &::\Rightarrow \ (\textit{cons Value List}) \end{aligned}$$

These two rules define a List as a data structure that can contain any number of elements. Starting from null, we can create Lists of any length. For example:

- *null* evaluates to a List containing no elements.
- (*cons* 1 *null*) evaluates to containing one element.
- (*cons* 1 (*cons* 5 *null*)) evaluates to a List containing two elements.
- (*cons* 1 (*cons* 5 (*cons* 0 *null*))) evaluates to a List containing three elements.

Scheme provides a convenient procedure, *list*, for constructing a List. The *list* procedure takes zero or more inputs, and evaluates to a List containing those inputs in order. The following expressions are equivalent to the corresponding expressions above: (*list*), (*list* 1), (*list* 1 5), and (*list* 1 5 0).

Lists are just a collection of Pairs, so we can draw a List using the same box and arrow notation we used to draw structures created with Pairs. Here is the structure resulting from (*list* 1 5 0):



There are three Pairs in the List, the second cell of each Pair is a List. For the third Pair, the second cell is the List *null*, which we draw as a slash through the final cell in the diagram.

Table 5.1 summarizes some of the built-in procedures for manipulating Pairs and Lists.

Procedure	Type	Output
cons	Value × Value → Pair	a Pair consisting of the two inputs
car	Pair → Value	the first cell of the input Pair
cdr	Pair → Value	the second cell of the input Pair
list	zero or more Values → List	a List containing the inputs
null?	Value → Boolean	true if the input is null, otherwise false
pair?	Value → Boolean	true if the input is a Pair, otherwise false
list?	Value → Boolean	true if the input is a List, otherwise false

Table 5.1. Selected Built-In Scheme Procedures for Lists and Pairs.

Exercise 5.10. For each of the following expressions, explain whether or not the expression evaluates to a List. You can check your answers with a Scheme interpreter by using the *list?* procedure that takes one input and evaluates to true if the value of that input is a List, and false otherwise.

a. *null*

b. (*cons* 1 2)

c. (*cons* 1 *null*)

d. (*cons null null*)

e. (*cons* (*cons* (*cons* 1 2) 3) *null*)

f. (*cons* 1 (*cons* 2 (*cons* 3 (*cons* 4 *null*))))

g. (*cdr* (*cons* 1 (*cons* 2 (*cons null null*))))

h. (*cons* (*list* 1 2 3) 4)

5.4 List Procedures

Since the List data structure is defined recursively, it is natural to define recursive procedures to examine and manipulate lists. Whereas most recursive procedures on inputs that are Numbers usually used zero as the base case, for lists the most common base case is the empty list. With numbers, we make progress by subtracting one to produce the input for the recursive application; with lists, we make progress by using *cdr* to reduce the length of the input List by one element for each recursive application. This

means we often break problems involving Lists into figuring out what to do with the first element of the List and the result of applying the recursive procedure to the rest of the List.

Next we consider procedures that examine lists by walking through their elements and outputing a scalar value. Section 5.4.2 generalizes these procedures. In Section 5.4.3, we explore procedures that produce lists as their output.

5.4.1 Procedures that Examine Lists

Here we explore procedures take a single List as input and produce a scalar value that depends on the elements of the List as output. All of the examples in this section involve base cases where the List is empty, and recursive cases that apply the recursive procedure to the *cdr* of the input List.

Example 5.1: Length. How many elements are in a given List?[3]

Our standard recursive problem solving technique is to "Think of the simplest version of the problem, something you can already solve." For this procedure, the simplest version of the problem is when the input is the empty list (*null*). We already know the length of the empty list is 0. So, the base case test is (*null? p*) and the output for the base case is 0.

For the recursive case, we need to consider the structure of all lists other than *null*. Recall our definition of a List: either *null* or (*cons Value List*). The base case deals with the *null* list, so the recursive case must deal with a List that is a Pair of an element an a List. The length of this List is one more than the length of the List that is the *cdr* of the Pair.

```
(define (list-length p)
    (if (null? p)
        0
        (+ 1 (list-length (cdr p)))))
```

[3]Scheme provides a built-in procedure *length* that takes a List as its input and outputs the number of elements in the List. Here, we will define our own *list-length* procedure that does this (without using the built-in *length* procedure). As with many other examples and exercises in this chapter, it is instructive to define our own versions of some of the built-in list procedures.

Here are a few example applications of our *list-length* procedure:

```
> (list-length null)
0
> (list-length (cons 0 null))
1
> (list-length (list 1 2 3 4))
4
> (list-length (cons 1 2))
cdr: expects argument of type <pair>; given 2
```

The last evaluation produces and error, since we are applying *list-length* to an input that is not a List.

Example 5.2: List Sums and Products. First, we define a procedure that takes a List of numbers as input and produces as output the sum of the numbers in the input List. As usual, the base case is when the input is *null*: the sum of an empty list is 0. For the recursive case, we need to add the value of the first number in the List, to the sum of the rest of the numbers in the List.

```
(define (list-sum p)
  (if (null? p)
      0
      (+ (car p) (list-sum (cdr p)))))
```

We can define *list-product* similarly, except here we multiply the numbers in the List. The base case result cannot be 0, though, since then the final result would always be 0 since any number multiplied by zero is zero. We follow the mathematical convention that the product of the empty list is 1.

```
(define (list-product p)
  (if (null? p)
      1
      (* (car p) (list-product (cdr p)))))
```

Exercise 5.11. [⋆] Define a procedure *is-list?* that takes one input and produces true if the input is a List, and false otherwise. (Scheme provides a built-in *list?* procedure with this behavior, but you should not use it in your definition.) Hint: recall the definition of a List in Section 5.3.

Exercise 5.12. [⋆] Define a procedure *list-max* that takes a List of non-negative numbers as its input and produces as its result the value of the greatest element in the List (or 0 if there are no elements in the input List). For example, (*list-max* (*list* 1 5 0)) should evaluate to 5.

5.4.2 Generic Accumulators

The *list-length*, *list-sum*, and *list-product* procedures all have very similar structures. They all have a structure where the base case is when the input is the empty list, and the recursive case involves doing something with the first element of the List and recursively calling the procedure with the rest of the List:

```
(define (Recursive-Procedure p)
  (if (null? p)
      Base-Case-Result
      (Accumulator-Function (car p) (Recursive-Procedure (cdr p)))))
```

We can define a generic accumulator procedure for lists by making the base case result and accumulator function inputs:

```
(define (list-accumulate f base p)
  (if (null? p)
      base
      (f (car p) (list-accumulate f base (cdr p)))))
```

Then, we can define the *list-sum* and *list-product* procedures using *list-accumulate*:

```
(define (list-sum p) (list-accumulate + 0 p))
(define (list-product p) (list-accumulate * 1 p))
```

Defining the *list-length* procedure is a bit more complicated.

In our previous definition, the recursive case in the *list-length* procedure is (+ 1 (*list-length* (*cdr p*))). Unlike the *list-sum* example, the recursive case for *list-length* does not use the value of the first element of the List. The way *list-accumulate* is defined, we need a procedure that takes two inputs—the first input is the first element of the List; the second input is the result of applying *list-accumulate* to the rest of the List.

We should follow our usual strategy: be optimistic! Being optimistic as in recursive definitions, the value of the second input should be the length of the rest of the List. Hence, we need to pass in a procedure that takes two inputs, ignores the first input, and outputs one more than the value of the second input. We can make this procedure using (**lambda** (*a b*) (+ 1 *b*)). In the definition, we give the parameters more meaningful names:

```
(define (list-length p)
  (list-accumulate (lambda (el length-rest) (+ 1 length-rest)) 0 p))
```

Exercise 5.13. Define the *list-max* procedure (from Exercise 12) using *list-accumulate*.

Exercise 5.14. [⋆] Define the *list?* procedure (from Exercise 11) using *list-accumulate*.

Example 5.3: Accessing List Elements. The built-in *car* procedure provides a way to get the first element of a list, but what if we want to get the third element? We can do this by taking the *cdr* twice to eliminate the first two elements, and then using *car* to get the third: (*car* (*cdr* (*cdr p*))) evaluates to the third element of the List *p*.

We want a more general procedure that can access any selected list element. Hence, the procedure needs two inputs: a List, and an index Number that identifies the element. If we start counting from 1, then the base case is when the index is 1 and the output should be the first element of the List: (**if** (= *n* 1) (*car p*) ...).

For the recursive case, we make progress by eliminating the first element of the list. But, we also need to adjust the index. Since we have removed the first element of the list, the index should be reduced by one. For example, instead of wanting the third element of the original list, we now want the second element of the *cdr* of the original list.

```
(define (list-get-element p n)
  (if (= n 1)
      (car p)
      (list-get-element (cdr p) (- n 1))))
```

What happens if we apply *list-get-element* with an index that is larger than the size of the input List (for example, (*list-get-element* (*list* 1 2) 3))?

The first recursive call will be (*list-get-element* (*list* 2) 2). The second recursive call will be (*list-get-element* (*list*) 1). At this point, *n* is 1, so the base case is reached and (*car p*) is evaluated. But, *p* is the empty list (which is not a Pair), so an error results.

A better version of *list-get-element* would provide a meaningful error message when the requested element is out of range. We can do this by adding an if-expression that tests if the input List is *null*:

```
(define (list-get-element p n)
  (if (null? p)
      (error "Index out of range")
      (if (= n 1)
          (car p)
          (list-get-element (cdr p) (- n 1)))))
```

The built-in procedure *error* takes a String as input. The String datatype is a sequence of characters; we can create a String by surrounding characters with double quotes, as in the example. The *error* procedure terminates program execution with a message that displays the input value.

defensive programming

Checking explicitly for invalid inputs is known as *defensive programming*. As your programs get more complex, programming defensively will help you avoid tricky to debug errors and will make it easier to understand and remember what your code is doing.

Exercise 5.15. [⋆] Define a procedure *list-last-element* that takes as input a List and outputs the last element of the input List. It is an error to apply *list-last-element* to the empty list.

Exercise 5.16. [⋆] The error message produced by our *list-get-element* is not very helpful. A more helpful message would print out the original input list and index. This is not possible to add with the provided implementation, since by the time the error is reached several elements of the list may have been eliminated and the index reduced through the recursive calls. Define a version of *list-get-element* that behaves identically for normal inputs, but produces more helpful error messages when the input index is out of range. A built-in procedure that will be useful for this is the *format* procedure. It is similar to the *printf* procedure described in Section 4.5.1, but instead of printing it returns a String. For example, (*format* "The list ~a is different from the number ~a." (*list* 1 5 0) 150) evaluates to the String "The list (1 5 0) is different from the number 150.".

Exercise 5.17. [⋆] Define a procedure *list-ordered?* that takes two inputs, a test procedure and a List. It evaluates to true if all the elements of the List are ordered according to the test procedure. For example, (*list-ordered?* $<$ (*list* 1 2 3)) should evaluate to true, and (*list-ordered?* $>$ (*list* 4 3 3 2)) should evaluate to false.

5.4.3 Procedures that Construct Lists

The procedures in this section take values (including lists) as input, and produce a new List as output. As before, the empty list is typically the base case. Since we are producing a List as output, the result for the base case is also usually null. The recursive case will use *cons* to construct a List combining the first element with the result of the recursive application on the rest of the List.

Example 5.4: Mapping. One common task for manipulating a List is to produce a new List that is the result of applying the same procedure to

every element in the input List.

For the base case, applying a procedure to every element of the empty list produces the empty list. For the recursive case, we use *cons* to construct a List consisting of the procedure applied to the first element of the List and the result of applying the procedure to every element in the rest of the List.

For example, here is a procedure that constructs a List that contains the square of every element of the input List:

```
(define (list-square p)
  (if (null? p)
      null
      (cons (square (car p))
            (list-square (cdr p)))))
```

We can generalize this by making the procedure which is mapping each List element an input. The procedure *list-map* takes a procedure as its first input and a List as its second input and evaluates to the List containing the values resulting from applying the procedure to each element of the input List.[4]

```
(define (list-map f p)
  (if (null? p)
      null
      (cons (f (car p))
            (list-map f (cdr p)))))
```

Then, *square-all* could be defined as:

```
(define (square-all p) (list-map square p))
```

Exercise 5.18. Define a procedure *list-increment* that takes as input a List of numbers, and produces as output the input List with each value incremented by one. For example, (*list-increment* 1 5 0) should evaluate to the List (2 6 1).

Exercise 5.19. Use *list-map* and *list-sum* to define *list-length*:

```
(define (list-length p) (list-sum (list-map _____ p)))
```

Example 5.5: Filtering. Consider defining a procedure that takes as input a List of numbers, and evaluates to a List of all the non-negative numbers

[4]Scheme provides a built-in *map* procedure. It behaves like this one when passed a procedure and a single List as inputs, but can also work on more than one List input at a time.

in the input. For example, (*list-filter-negative* (*list* 1 −3 −4 5 −2 0)) should evaluate to the List (1 5 0).

First, consider the base case when the input List is empty. If we filter the negative numbers from the empty list, the result is an empty list. So, for the base case, the result should be null.

In the recursive case, we need to determine if the first element should be included in the output List or not. If it should be included, we construct a new List consisting of the first element followed by the result of filtering the remaining elements in the List. If it should not be included, we do not include the first element, and the result is the result of filtering the remaining elements in the List.

```
(define (list-filter-negative p)
  (if (null? p)
      null
      (if (>= (car p) 0)
          (cons (car p) (list-filter-negative (cdr p)))
          (list-filter-negative (cdr p)))))
```

Similarly to *list-map*, we can generalize our filter by making the test procedure as an input, so we can use any predicate to determine which elements to include in the output List.[5]

```
(define (list-filter test p)
  (if (null? p)
      null
      (if (test (car p))
          (cons (car p) (list-filter test (cdr p)))
          (list-filter test (cdr p)))))
```

Using the *list-filter* procedure, we can define *list-filter-negative* as:

```
(define (list-filter-negative p) (list-filter (lambda (x) (>= x 0)) p))
```

We could also define the *list-filter* procedure using the *list-accumulate* procedure from Section 5.4.1:

```
(define (list-filter test p)
  (list-accumulate
    (lambda (el rest) (if (test el) (cons el rest) rest))
    null
    p))
```

[5]Scheme provides a built-in function *filter* that behaves like our *list-filter* procedure.

Exercise 5.20. Define a procedure *list-filter-even* that takes as input a List of numbers and produces as output a List consisting of all the even elements of the input List.

Exercise 5.21. [⋆] Define a procedure *list-remove* that takes two inputs: a test procedure and a List. As output, it produces a List that is a copy of the input List with all of the elements for which the test procedure evaluates to true removed. For example, (*list-remove* (**lambda** (*x*) (= *x* 0)) (*list* 0 1 2 3)) should evaluates to the List (1 2 3).

Exercise 5.22. [⋆⋆] Define a procedure *list-unique-elements* that takes as input a List and produces as output a List containing the unique elements of the input List. The output List should contain the elements in the same order as the input List, keeping the first appearance of each duplicate value.

Example 5.6: Append. The *list-append* procedure takes as input two lists and produces as output a List consisting of the elements of the first List followed by the elements of the second List.[6] For the base case, when the first List is empty, the result of appending the lists should just be the second List. When the first List is non-empty, we can produce the result by *cons*-ing the first element of the first List with the result of appending the rest of the first List and the second List.

```
(define (list-append p q)
  (if (null? p)
      q
      (cons (car p) (list-append (cdr p) q))))
```

Example 5.7: Reverse. The *list-reverse* procedure takes a List as input and produces as output a List containing the elements of the input List in reverse order.[7] For example, (*list-reverse* (*list* 1 2 3)) should evaluate to the List (3 2 1). As usual, we consider the base case where the input List is null first. The reverse of the empty list is the empty list. To reverse a non-empty List, we should put the first element of the List at the end of the result of reversing the rest of the List. The tricky part is putting the first element at the end, since *cons* only puts elements at the beginning of a List. We can use the *list-append* procedure defined in the previous example to put a List at the end of another List. To make this work, we need to turn the element at the front of the List into a List containing just that element. We do this using (*list* (*car p*)).

[6] There is a built-in procedure *append* that does this. The built-in *append* takes any number of Lists as inputs, and appends them all into one List.

[7] The built-in procedure *reverse* does this.

```
(define (list-reverse p)
  (if (null? p)
      null
      (list-append (list-reverse (cdr p)) (list (car p)))))
```

Exercise 5.23. [⋆] Define the *list-reverse* procedure using *list-accumulate*.

Example 5.8: Intsto. For our final example, we consider the problem of constructing a List containing the whole numbers between 1 and the input parameter value. For example, (*intsto* 5) should evaluate to the List (1 2 3 4 5).

Here, we combine some of the ideas from the previous chapter on creating recursive definitions for problems involving numbers, and from this chapter on lists. Since the input parameter is not a List, the base case is not the usual base case when the input is *null*. Instead, we use the input value 0 as the base case. The result for input 0 is the empty list. For higher values, the output is the result of putting the input value at the end of the List of numbers up to the input value minus one.

A first attempt that doesn't quite work is:

```
(define (revintsto n)
  (if (= n 0)
      null
      (cons n (revintsto (- n 1)))))
```

The problem with this solution is that it is *cons*-ing the higher number to the front of the result, instead of at the end. Hence, it produces the List of numbers in descending order: (*revintsto* 5) evaluates to the List (5 4 3 2 1).

One solution is to reverse the result by composing *list-reverse* with *revintsto*:

```
(define (intsto n) (list-reverse (revintsto n)))
```

Equivalently, we can use the *fcompose* procedure from Section 4.2:

```
(define intsto (fcompose list-reverse revintsto))
```

Alternatively, we could use *list-append* to put the high number directly at the end of the List. Since the second operand to *list-append* must be a List, we use (*list n*) to make a singleton List containing the value as we did for *list-reverse*.

```
(define (intsto n)
  (if (= n 0)
      null
      (list-append (intsto (- n 1)) (list n))))
```

Although all of these procedures are functionally equivalent (for all valid inputs, each function produces exactly the same output), the amount of computing work (and hence the time they take to execute) varies substantially across the implementations. In Chapter 8, we introduce techniques for understanding the cost of evaluating a procedure and we consider the costs of the different *intsto* implementations.

Exercise 5.24. Define the *factorial* procedure (from Example 1) using *intsto*.

5.5 Lists of Lists

The elements of a List can be any datatype, including, of course, other Lists. In defining procedures that operate on Lists of Lists, we often use more than one recursive call when we need to go inside the inner Lists.

Example 5.9: Summing Nested Lists. Consider the problem of summing all the numbers in a List of Lists. For example, (*nested-list-sum* (*list* (*list* 1 2 3) (*list* 4 5 6))) should evaluate to 21. We can define *nested-list-sum* using *list-sum* on each List.

```
(define (nested-list-sum p)
  (if (null? p)
      0
      (+ (list-sum (car p))
         (nested-list-sum (cdr p)))))
```

This works when we know the input is a List of Lists. But, what if the input can contain arbitrarily deeply nested Lists?

To handle this, we need to recursively sum the inner Lists. Each element in our deep List is either a List or a Number. If it is a List, we should add the value of the sum of all elements in the List to the result for the rest of the List. If it is a Number, we should just add the value of the Number to the result for the rest of the List. So, our procedure involves two recursive calls: one for the first element in the List when it is a List, and the other for the rest of the List.

```
(define (deep-list-sum p)
  (if (null? p)
      0
      (+ (if (list? (car p))
             (deep-list-sum (car p))
             (car p))
         (deep-list-sum (cdr p)))))
```

Example 5.10: Flattening Lists. Another way to compute the deep list sum would be to first flatten the List, and then use the *list-sum* procedure.

Flattening a nested list takes a List of Lists and evaluates to a List containing the elements of the inner Lists. We can define *list-flatten* by using *list-append* to append all the inner Lists together.

```
(define (list-flatten p)
  (if (null? p)
      null
      (list-append (car p) (list-flatten (cdr p)))))
```

This flattens a List of Lists into a single List. If we want to completely flatten a deeply nested List, we need to use multiple recursive calls as we did with *deep-list-sum*.

```
(define (deep-list-flatten p)
  (if (null? p)
      null
      (list-append (if (list? (car p))
                       (deep-list-flatten (car p))
                       (list (car p)))
                   (deep-list-flatten (cdr p)))))
```

Now we can define *deep-list-sum* as

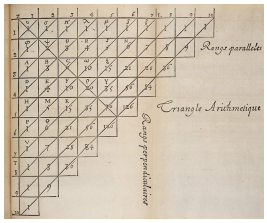
```
(define deep-list-sum (fcompose deep-list-flatten list-sum))
```

Exercise 5.25. Define a procedure *deep-list-map* that behaves similarly to *list-map* but on deeply nested lists. It should take two parameters, a mapping procedure, and a List (that may contain deeply nested Lists as elements), and output a List with the same structure as the input List with each value mapped using the mapping procedure.

Exercise 5.26. Define a procedure *deep-list-filter* that behaves similarly to *list-filter* but on deeply nested lists.

Excursion 5.1: Pascal's Triangle. This triangle is known as Pascal's Triangle (named for Blaise Pascal, although known to many others before him):

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...
```



Pascal's Triangle

Each number in the triangle is the sum of the two numbers immediately above and to the left and right of it. The numbers in Pascal's Triangle are the coefficients in a binomial expansion. The numbers of the n^{th} row (where the rows are numbered starting from 0) are the coefficients of the binomial expansion of $(x + y)^n$. For example, $(x + y)^2 = x^2 + 2xy + y^2$, so the coefficients are 1 2 1, matching the third row in the triangle; from the fifth row, $(x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$. The values in the triangle also match the number of ways to choose k elements from a set of size n (see Exercise 5) — the k^{th} number on the n^{th} row of the triangle gives the number of ways to choose k elements from a set of size n. For example, the third number on the fifth (n = 4) row is 6, so there are 6 ways to choose 3 items from a set of size 4.

The goal of this excursion is to define a procedure, *pascals-triangle* to produce Pascal's Triangle. The input to your procedure should be the number of rows; the output should be a list, where each element of the list is a list of the numbers on that row of Pascal's Triangle. For example, (*pascals-triangle* 0) should produce () (the empty list), (*pascals-triangle* 0) should produce ((1)) (a list containing one element which is a list containing the number 1), and (*pascals-triangle* 4) should produce ((1) (1 1) (1 2 1) (1 3 3 1) (1 4 6 4 1)).

Very ambitious readers will attempt to define *pascals-triangle* themselves; the sub-parts below provide some hints for one way to define it.

a. First, define a procedure *expand-row* that expands one row in the triangle. It takes a List of numbers as input, and as output produces a List with one more element than the input list. The first number in the output List should be the first number in the input List; the last number in the output List should be the last number in the input List. Every other number in the output List is the sum of two numbers in the input List. The n^{th} number in the output List is the sum of the $n - 1^{th}$ and n^{th} numbers in the input List. For example, (*expand-row* (*list* 1)) should evaluate to the List (1 1); (*expand-row* (*list* 1 1)) should evaluate to the List (1 2 1); and (*expand-row* (*list* 1 4 6 4 1)) should evaluate to the List (1 5 10 10 5 1). This is trickier than the recursive list procedures we have seen so

far since the base case is not the empty list (it is okay if your *expand-row* procedure produces an error when the input is an empty list). It also needs to deal with the first element specially. So, to define *expand-row*, it will be helpful to divide it into two procedures, one that deals with the first element of the list, and one that produces the rest of the list:

```
(define (expand-row p)
  (cons (car p) (expand-row-rest p)))
```

b. Next, define a procedure *pascals-triangle-row* that takes one input, n, and outputs the n^{th} row of Pascal's Triangle. For example, (*pascals-triangle-row* 0) should evaluate to the List (1) and (*pascals-triangle-row* 4) should produce (1 4 6 4 1).

c. Finally, define *pascals-triangle* with the behavior described above. (If you have a hard time getting the rows of the triangle to appear in the right order, look at the *intsto* example.)

5.6 Data Abstraction

The mechanisms we have for constructing and deconstructing complex data structures are valuable because they enable us to think about programs closer to the level of the problem we are solving than the low level of how data is stored and manipulated in the computer. Our goal is to hide unnecessary details about how data is represented so we can focus on the important aspects of what the data means and what we need to do with it to solve our problem. The technique of hiding how data is represented from how it is used is known as *data abstraction*.

data abstraction

The datatypes we have seen so far are not very abstract. We have datatypes for representing Pairs, triples, and Lists, but we want datatypes for representing objects closer to the level of the problem we want to solve. A good data abstraction is abstract enough to be used without worrying about details like which cell of the Pair contains which data and how to access the different elements of a List. Instead, we want to define procedures with meaningful names that extract or manipulate the relevant parts of our data.

The rest of this section is an extended example that illustrates how we can solve problems by first identifying the objects we need to model to model the problem, and then implementing data abstractions that represent those objects. Once the appropriate data abstractions are designed and implemented, the solution to the problem often follows readily. This example also uses many of the list procedures defined earlier in this chapter.

Pegboard Puzzle

Example 5.11: Pegboard Puzzle. For this example, we develop a program to solve the infamous pegboard puzzle, often found tormenting unsuspecting diners at pancake restaurants.The standard puzzle is a one-player game played on a triangular-shaped board with fifteen holes, initially with a peg in each hole. The game begins by removing one of the pegs. Then, the goal is to remove all but one of the pegs by jumping pegs over one another. A peg may jump over an adjacent peg only when there is a free hole on the other side of the peg. The jumped peg is removed. The game ends when there are no possible moves. If there is only one peg remaining, the player wins (according to the Cracker Barrel version of the game, "Leave only one—you?re genius"). If more than one peg remains, the player loses ("Leave four or more'n you're just plain 'eg-no-ra-moose'.").

Our goal is to develop a program that provides a winning solution to the pegboard game, regardless of the starting position (which hole is initially empty). We will use a *brute force* approach: try all possible moves until we find one that works. Brute force solutions work well when problems are fairly small. Because they have to try all possibilities, however, they are often too slow for solving large problems (even on the most powerful computers).[8]

brute force

The first thing to think about to solve a complex problem is what datatypes we need. We want datatypes that represent the things we need to model in our problem solution. For the pegboard game, one thing we need to model is the board. The important thing about a datatype is what you can do with it. So, to design our board datatype we need to think about what we want to do with a board. In the physical pegboard game, the board holds the pegs. The important property we need to observe about the board is which holes on the board contain pegs. For this, we need a way of identifying board positions.

We can identify the board positions using row and column numbers:

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

So, we will make a Position datatype to represent a position on the board. A position has a row and a column, so we could just use a Pair to represent a position. This would work, but we prefer to have a more abstract datatype

[8] As we will see in Chapter 18, the generalized pegboard puzzle is an example of a class of problems known as *NP-Complete*. This means it is not known whether or not any solution exists that is substantially better than the brute force solution, but it would be extraordinarily surprising (and have momentous impact) to find one.

so we can think about a position's row and column, rather than thinking that a position is a Pair and using the *car* and *cdr* procedures to extract the row and column from the position.

Our Position datatype should provide at least these operations:

- *make-position*: Number × Number → Position — create a Position representing the row and column given by the inputs
- *position-get-row*: Position → Number — gets the row number of the input Position
- *position-get-column*: Position → Number — gets the column number of the input Position

Since the Position needs to keep track of two numbers, a natural way to implement the Position datatype is to use a Pair:

```
(define (make-position row col) (cons row col))
(define (position-get-row posn) (car posn))
(define (position-get-column posn) (cdr posn))
```

tagged list

A more defensive way to implement the Position datatype is to use a *tagged list*. We can use a symbol identify the type of data, and the remaining elements in the list as the data. Symbols are a quote (') followed by a sequence of characters. The important operation we can do with a Symbol, is test whether it is an exact match for another symbol using the *eq?* procedure. So, we first define the Tagged-List datatype (we use the *list-get-element* procedure from Example 3, and the built-in *format* procedure introduced in Exercise 16):

```
(define (make-taggedlist tag p) (cons tag p))
(define (taggedlist-get-tag p) (car p))

(define (taggedlist-get-element tag p n)
  (if (eq? (taggedlist-get-tag p) tag)
      (list-get-element (cdr p) n)
      (error (format "Bad list tag: ~a (expected ~a)"
                     (taggedlist-get-tag p) tag))))
```

This is an example of defensive programming. Using our tagged lists, if we accidentally attempt to use a value that is not a Position as a position, we will get an easy to understand error message instead of a hard-to-debug error (or worse, an unnoticed incorrect result).

Using the tagged list, we define the Position datatype as:

```
(define (make-position row col) (make-taggedlist 'Position (list row col)))
```

These procedures provide ways to extract the row and column from a Position, and to determine if two Positions are the same place:

```
(define (position-get-row posn)
  (taggedlist-get-element 'Position posn 1))

(define (position-get-column posn)
  (taggedlist-get-element 'Position posn 2))
```

Here are some example interactions with our Position datatype:

```
> (define pos (make-position 2 1))
> pos
(Position 2 1)
> (get-position-row pos)
2
> (get-position-row (list 1 2))
Bad list tag: 1 (expected Position)
```

The last evaluation produces an error, since we are attempting to apply a procedure that expects a Position as its input to a value that is not a Position.

Now we have a way of identifying positions, we can make progress on the board datatype. It should provide at least these two procedures:

- *make-board*: Number → Board — create a board full of pegs with the input number of rows. (In the physical board, the board is always the same size with 5 rows, but we define our Board datatype to support any number of rows.)
- *board-contains-peg?*: Position → Boolean — evaluates to true if and only if the hold at the input Position contains a peg.

We also need to be able to change the state of the board to reflect moves in the game. This means the board data type should have operations for removing and adding pegs to the board. Nothing we have seen so far, however, provides a means for changing the state of an existing object.[9] So,

[9]We will introduce mechanisms for changing state in Chapter 11. Allowing state to change leads to lots of complexities including breaking our substitution model of evaluation.

instead of thinking of these operations as changing the state of the board, what they really do is create a new board that is different from the input board by the one new peg. Hence, these procedures will take a Board and Position as inputs, and produce as output a Board.

- *board-add-peg*: Board × Position → Board — the output is a Board containing all the pegs of the input Board and one additional peg at the input Position. If the input Board already has a peg at the input Position, produces an error.
- *board-remove-peg*: Board × Position → Board — the output is a Board containing all the pegs of the input Board except for the peg at the input Position. If the input Board does not have a peg at the input Position, produces an error.

There are lots of different ways we could represent the Board. One possibility is to keep a List of the Positions of the pegs on the board. Another possibility is to keep a List of the Positions of the empty holes on the board. Yet another possibility is to keep a List of Lists, where each List corresponds to one row on the board. The elements in each of the Lists are Booleans representing whether or not there is a peg at that position. The good thing about data abstraction is we could pick any of these representations and change it to a different representation later (for example, if we needed a more efficient board implementation). As long as the procedures for implementing the Board are updated the work with the new representation, all of the other code should continue to work correctly without any changes.

We choose the third option, to represent a Board using a List of Lists, where each element of the inner lists is a Boolean indicating whether or not the corresponding position contains a peg. So, our *make-board* procedure should evaluate to a List of Lists, where each element of the List contains the row number of elements and all the inner elements are true (the initial board is completely full of pegs). First, we define a procedure *make-list-of-constants* that takes two inputs, a Number, *n*, and a Value, *val*. The output is a List of length *n* where each element has the value *val*.

```
(define (make-list-of-constants n val)
  (if (= n 0)
      null
      (cons val (make-list-of-constants (- n 1) val))))
```

To make the initial board, we can use *make-list-of-constants* to make each row of the board. As usual, a recursive problem solving strategy works well: the simplest board is a board with zero rows (which should be represented as

the empty list); for each larger board, we need to add a row with the right number of elements.

The tricky part is getting the rows to be in order. This is similar to the problem we faced with *intsto*, and a similar solution using *append-list* works here.

```
(define (make-board rows)
  (if (= rows 0)
      null
      (list-append (make-board (- rows 1))
                   (list (make-list-of-constants rows true)))))
```

For example, (*make-board* 5) evaluates to the List of Lists:

```
((true)
 (true true)
 (true true true)
 (true true true true)
 (true true true true true))
```

The *board-rows* procedure takes a Board as input and outputs the number of rows on the board.

```
(define (board-rows board) (length board))
```

Now that we have defined our board datatype, we can implement the procedures for observing and manipulating the board. The *board-contains-peg?* procedure takes a Board and a Position as input, and should output a Boolean indicating whether or not that Position contains a peg. To implement *board-contains-peg?* we need to find the appropriate row in our board representation, and then find the element in its list corresponding to the Position's column. The *list-get-element* procedure (from Example 3) does exactly what we need. Since our board is represented as a List of Lists, we need to use it twice: first to get the row, and then to select the column within that row:

```
(define (board-contains-peg? board pos)
  (list-get-element (list-get-element board (position-get-row pos))
                    (position-get-column pos)))
```

Defining the procedures for adding and removing pegs from the board is a bit more complicated. We need to make a board with every row identical to the input board, except the row where the peg is added or removed. For that row, we need to replace the corresponding value. First we implement

the *board-add-peg* procedure. After implementing this, we will see that removing is very similar, so both adding and removing can be done with a more general procedure that takes an extra input.

To implement *board-add-peg*, we first consider the subproblem of adding a peg to a row. The procedure *row-add-peg* takes as input a List representing a row on the board and a Number indicating the column where the peg should be added. We can define *row-add-peg* recursively: the base case is when the new peg belongs at the beginning of the row (the column number is 1); in the recursive case, we copy the first element in the List, and add the peg to the rest of the list.

```
(define (row-add-peg pegs col)
  (if (= col 1)
      (cons true (cdr pegs))
      (cons (car pegs) (row-add-peg (cdr pegs) (- col 1)))))
```

To add the peg to the board, we can use *row-add-peg* to add the peg to the appropriate row, and keep all the other rows the same. We divide the problem into a procedure that checks the add is valid (the selected position does not already contain a peg), and a helper procedure that produces the new board.

```
(define (board-add-peg-helper board row col)
  (if (= row 1) ; this row
      (cons (row-add-peg (car board) col)
            (cdr board))
      (cons (car board)
            (board-add-peg-helper (cdr board) (- row 1) col))))


(define (board-add-peg board pos)
  (if (board-contains-peg? board pos)
      (error (format "Board already contains peg at position: ~a" pos))
      (board-add-peg-helper board (position-get-row pos)
                                  (position-get-column pos))))
```

To define *board-replace-peg* we would need to do essentially the same thing, but instead of making the value at the selected position true to represent a peg, we need to make it false to indicate an empty hole. One might be tempted to "cut-and-paste" to define *board-remove-peg* using the adding procedures already defined, but this makes the code nearly twice as long as it needs to be. It is much better to instead write a more general procedure that can be used to define both *board-add-peg* and *board-remove-peg*. We do this by defining the *board-replace-peg* procedure that takes an extra parameter (true for adding, false for replacing).

```
(define (row-replace-peg pegs col val)
  (if (= col 1)
      (cons val (cdr pegs))
      (cons (car pegs) (row-replace-peg (cdr pegs) (- col 1) val))))

(define (board-replace-peg board row col val)
  (if (= row 1)
      (cons (row-replace-peg (car board) col val)
            (cdr board))
      (cons (car board)
            (board-replace-peg (cdr board) (- row 1) col val))))
```

Then, both *board-add-peg* and *board-remove-peg* can be defined simply using *board-remove-peg*.

```
(define (board-add-peg board pos)
  (if (board-contains-peg? board pos)
      (error (format "Board already contains peg at position: ~a" pos))
      (board-replace-peg board (position-get-row pos)
                               (position-get-column pos) true)))

(define (board-remove-peg board pos)
  (if (not (board-contains-peg? board pos))
      (error (format "Board does not contain peg at position: ~a" pos))
      (board-replace-peg board (position-get-row pos)
                               (position-get-column pos) false)))
```

Now we have datatypes for representing a Position and the Board.

The next datatype we want is a way to represent a move. A move involves three positions: the position where the jumping peg starts, the position of the peg that is jumped and removed, and the landing position. One possibility would be to represent a move as a list of the three positions. A better option is to observe that once any two of the positions are known, the third position is determined. For example, if we know the starting position and the landing position, we know the jumped peg is at the position between them. So, we could represent a jump using a List containing the starting and landing positions and find the jumped position as needed from those.

Another possibility is to represent a jump by storing the starting Position and the direction. This is also enough to determine the jumpee and landing positions. This approach seems most natural, and avoids the difficulty of calculating jumpee positions. To do it, we first design a Direction datatype for representing the possible move directions. Directions have two components: the change in the row (we use 1 for down and -1 for up), and the

change in the column (1 for right and -1 for left). We can move directly up, down, left, and right. Because of the triangular shape of the board, only two of the diagonal directions make sense: up-left and down-right (up-right does not make sense since moving up normally is actually moving up and right on the triangular board).

We implement the Direction datatype using a similar tagged list datatype to how we defined Position.

```
(define (make-direction down right)
    (make-taggedlist 'Direction (list down right)))

(define (direction-get-vertical dir)
    (taggedlist-get-element 'Direction dir 1))

(define (direction-get-horizontal dir)
    (taggedlist-get-element 'Direction dir 2))
```

We also define names representing the six possible move directions, and a list of all move directions. This will be useful when we consider all possible moves.

```
(define direction-up (make-direction -1 0))
(define direction-down (make-direction 1 0))
(define direction-left (make-direction 0 -1))
(define direction-right (make-direction 0 1))
(define direction-up-left (make-direction -1 -1))
(define direction-down-right (make-direction 1 1))

(define all-directions
        (list direction-up direction-down direction-left direction-right
        direction-up-left direction-down-right))
```

Now, we can define the Move datatype using the starting position and the jump direction.

```
(define (make-move start direction)
  (make-taggedlist 'Move (list start direction)))

(define (move-get-start move)
  (taggedlist-get-element 'Move move 1))

(define (move-get-direction move)
  (taggedlist-get-element 'Move move 2))
```

We also need to define procedures for getting the jumpee and landing positions of a move. The jumpee position is the result of moving one step in the move direction from the starting position. So, it will be useful to define a procedure that takes a Position and a Direction as input, and outputs a Position that is one step in the input Direction from the input Position.

```
(define (direction-step pos dir)
  (make-position
    (+ (position-get-row pos) (direction-get-vertical dir))
    (+ (position-get-column pos) (direction-get-horizontal dir))))
```

Using *direction-step* we can implement procedure to get the middle and landing positions.

```
(define (move-get-jumpee move)
  (direction-step (move-get-start move) (move-get-direction move)))


(define (move-get-landing move)
  (direction-step (move-get-jumpee move) (move-get-direction move)))
```

With our Board, Move, and Position datatypes, we can now implement a procedure that models making a move on a board. To make a move we remove the jumped peg, and move the peg at the move starting position to the landing position. We can consider moving the peg as removing the peg from the starting position and adding a peg to the landing position.

```
(define (execute-move board move)
  (board-add-peg
   (board-remove-peg (board-remove-peg board (move-get-start move))
                     (move-get-jumpee move))
   (move-get-landing move)))
```

Now that we have datatypes for modeling positions, moves, and the board, and a way to make moves on the board, we are ready to develop the solution. At each step, we need to try all valid moves on the board to see if any move leads to a winning position (that is, a position with only one peg remaining). So, we need a procedure that takes a Board as its input and outputs a List of all valid moves on the board. We break this down into the problem of producing a list of all conceivable moves (all moves in all directions from all starting positions on the board), filtering that list for moves that stay on the board, and then filtering the resulting list for moves that are legal (start at a position containing a peg, jump over a position containing a peg, and land in a position that is an empty hole).

First, we generate all conceivable moves by creating moves starting from each position on the board and moving in all possible move directions. We break this down further: the first problem is to produce a List of all positions on the board. We can generate a list of all row numbers using the *intsto* procedure (from Example 8): (*intsto* (*board-rows board*)). To get a list of all the positions, we need to produce a list of the positions for each row. We can do this by mapping each row to the corresponding list:

```
(list-map
  (lambda (row)
    (list-map
      (lambda (col) (make-position row col))
      (intsto row)))
  (intsto (board-rows board)))
```

This almost does what we need, except instead of producing one List containing all the positions, it produces a List of Lists for the positions in each row. The *list-flatten* procedure (from Example 10) produces a flat list containing all the positions.

```
(define (all-positions board)
  (list-flatten
   (list-map
     (lambda (row)
       (list-map
         (lambda (col) (make-position row col))
         (intsto row)))
     (intsto (board-rows board)))))
```

Now, for each Position, we need to consider all possible moves. Again, *list-map* is useful. For each Position we create a list of moves where each move is that Position as the starting position and the Direction is one of the possible move directions. This produces a List of Lists, so we use *list-flatten* to flatten the output of the *list-map* application into a single List of Moves.

```
(define (all-conceivable-moves board)
  (list-flatten
    (list-map
      (lambda (pos)
        (list-map
          (lambda (dir) (make-move pos dir))
          all-directions))
      (all-positions board))))
```

The List of Moves produced by *all-conceivable-moves* includes moves that fly

off the board. We use the *list-filter* procedure to remove those moves, to get the list of possible moves. For the filter procedure, we need a procedure that determines if a Position is on the board. A position is valid if its row number is between 1 and the number of rows on the board, and its column numbers is between 1 and the row number.

```
(define (valid-position? board pos)
  (and
   (>= (position-get-row pos) 1)
   (>= (position-get-column pos) 1)
   (<= (position-get-row pos) (board-rows board))
   (<= (position-get-column pos) (position-get-row pos))))
```

Hence, we can define *all-possible-moves* as:

```
(define (all-possible-moves board)
  (list-filter
    (lambda (move)
      (valid-position? board (move-get-landing move)))
    (all-conceivable-moves board)))
```

Finally, we need to filter out the moves that are not legal moves. A legal move must start at a position that contains a peg, jump over a position that contains a peg, and land in an empty hole. We can use *list-filter* to keep only the legal moves similarly to how we kept only the moves that stay on the board.

```
(define (all-legal-moves board)
  (list-filter
    (lambda (move)
      (and (board-contains-peg? board (move-get-start move))
           (board-contains-peg? board (move-get-jumpee move))
           (not (board-contains-peg? board
                                     (move-get-landing move)))))
   (all-possible-moves board)))
```

To find a solution, we will need a way to know when we have found a winning board. This is a board with only one peg. We implement a more general procedure to count the number of pegs on a board first. Our board representation used true to represent a peg. To count the pegs, we first map the Boolean values used to represent pegs to 1 if there is a peg and 0 if there is no peg. Then, we can use *sum-list* to count the number of pegs. Since the Board is a List of Lists, we first need to use *list-flatten* to put all the pegs in a single List.

```
(define (board-number-of-pegs board)
  (list-sum
   (list-map
    (lambda (peg) (if peg 1 0))
    (list-flatten board))))
```

Then, we can determine if a board represents a winning position by checking if it contains one peg.

```
(define (is-winning-board? board)
  (= (board-number-of-pegs board) 1))
```

Note that our *board-number-of-pegs* procedure depends on the way we choose to represent the Board data type (this is why we give it a name that begins with *board-*). Unlike the other procedures we have written for solving the game, this procedure would need to be reconsidered if we change the way the Board type is represented.

Our goal is to find a sequence of moves that leads to a winning position, starting from the current board. If there is a winning sequence of moves, we can find it by trying all possible moves on the current board. Each of these moves leads to a new board. If the original board has a winning sequence of moves, at least one of the new boards has a winning sequence of moves. Hence, we can solve the puzzle by recursively trying all moves until finding a winning position.

```
(define (solve-pegboard board)
  (if (is-winning-board? board)
      null ; no moves needed to reach winning position
      (try-moves board (all-legal-moves board))))
```

The *solve-pegboard* procedure evaluates to a List of Moves, if there is a sequence of moves that wins the game starting from the input Board. This could be null, in the case where the input board is already a winning board. If there is no sequence of moves to win from the input board, it outputs false.

It remains to define the *try-moves* procedure. It takes a Board and a List of Moves as inputs, and outputs either false if there is no sequence of moves that wins from the input Board starting with one of the Moves in the input List, or a List of Moves that wins.

The base case is when there are no moves to try. When the input list is *null* it means there are no moves to try. We will use the output false to mean this attempt did not lead to a winning board. Otherwise, we should try the

first move. If it leads to a winning position, we should evaluate to the List of Moves that starts with the first move, and is followed by the rest of the moves needed to solve the board resulting from taking the first move (that is, the result of *solve-pegboard* applied to the Board resulting from taking the first move). If the first move doesn't lead to a winning board, we should try the rest of the moves (by calling *try-moves* recursively).

```
(define (try-moves board moves)
  (if (null? moves)
      false ; didn't find a winner
      (if (solve-pegboard (execute-move board (car moves)))
          (cons (car moves)
                (solve-pegboard (execute-move board (car moves))))
          (try-moves board (cdr moves)))))
```

Evaluating (*solve-pegboard* (*make-board* 5)) produces false since there is no way to win starting from a completely full board. Evaluating (*solve-pegboard* (*board-remove-peg* (*make-board* 5) (*make-position* 1 1))) takes about three minutes to produce a result:

```
((Move (Position 3 1) (Direction −1 0))
 (Move (Position 3 3) (Direction 0 −1))
 (Move (Position 1 1) (Direction 1 1))
 (Move (Position 4 1) (Direction −1 0))
 (Move (Position 4 4) (Direction −1 −1))
 (Move (Position 5 2) (Direction −1 0))
 (Move (Position 5 3) (Direction −1 0))
 (Move (Position 2 1) (Direction 1 1))
 (Move (Position 2 2) (Direction 1 1))
 (Move (Position 5 5) (Direction −1 −1))
 (Move (Position 3 3) (Direction 1 0))
 (Move (Position 5 4) (Direction 0 −1))
 (Move (Position 5 1) (Direction 0 1)))
```

This is a sequence of moves for winning the game starting from a 5-row board with the top peg removed.

Exercise 5.27. [⋆] Change the implementation to use a different Board representation, such as keeping a list of the Positions of each hole on the board. Only the procedures with names starting with *board-* should need to change when the Board representation is changed. Compare your implementation to the one described here. Which representation is better?

Exercise 5.28. [⋆⋆] The described implementation is very inefficient. It does lots of redundant computation. For example, *all-possible-moves* evaluates to the same value every time it is applied to a board with the same number of rows. It is wasteful to recompute this over and over again to solve a given board. See how much faster you can make the pegboard solver. Can you make it fast enough to solve the 5-row board in less than half the original time? Can you make it fast enough to solve a 6-row board?

Exercise 5.29. [⋆] The standard pegboard puzzle uses a triangular board, but there is no reason the board has to be a triangle. Define a more general pegboard puzzle solver that works for a board of any shape.

5.7 Summary

Building from the simple pair, we can create complex data structures including lists. A List is either *null*, or a Pair whose second cell is a List. Since the List data structure is itself defined recursively, it is not surprising that many procedures that involve lists can be defined recursively.

We can specialize our general problem solving strategy from Chapter 3 for procedures involving lists:

1. Be *very* optimistic! Since lists themselves are recursive data structures, most problems involving lists can be solved with recursive procedures.
2. Think of the simplest version of the problem, something you can already solve. This is the base case. For lists, this is usually the empty list.
3. Consider how you would solve a big version of the problem by using the result for a slightly smaller version of the problem. This is the recursive case. For lists, the smaller version of the problem is the rest (*cdr*) of the List.
4. Combine the base case and the recursive case to solve the problem.

As we develop programs to solve more complex problems, it is increasingly important to find ways to manage complexity. We need to break problems down into smaller parts that can be solved separately. Data abstraction hides the details of how data is represented from what you can do with it. Solutions to complex problems can be developed by thinking about what objects need to be modeled, and designing data abstractions the implement those models. Most of the work in solving the problem is defining the right

datatypes; once we have the datatypes we need to model the problem well, we are usually well along the path to a solution.

One limitation of lists is that the only structure they provide is a linear sequence of elements; for some problems, it will be convenient to have richer structures such as trees (structures where each element can have more than one successor) and graphs (structures where elements can be connected in arbitrary ways). All of these structures can be built using just the Pair structure introduced in this chapter. In later chapters, we will explore these more complex data structures. Most of the concepts needed to construct and manipulate them follow directly from our simple List structures. Another problem with lists is that because of their sequential structure, many tasks cannot be efficiently performed on lists that could be performed efficiently on other data structures. In the following chapters, we consider the resources required to execute a procedure.