

I Introduction	3
3.1 Getting Started	8
3.2 Naming Values	16
3.3 From Repeated Expressions to Functions	21
3.4 Conditionals and Booleans	29
4.1 Introduction to Tabular Data	41
4.2 Processing Tables	51
5.1 From Tables to Lists	65
5.2 Processing Lists	74
5.3 Recursive Data	95
6.1 Introduction to Structured Data	102
6.2 Collections of Structured Data	108
7.1 Trees	114
8.1 Functions as Data	121
8.2 Queues from Lists	128
8.3 Examples, Testing, and Program Checking	134
9.1 From Pyret to Python	138
9.2 Dictionaries	147
9.3 Arrays	152
10.1 Introduction to Pandas	155
10.2 Reshaping Tables	166
11.1 State, Change, and Testing	167
11.2 Understanding Equality	179
11.3 Arrays and Lists in Memory	185
11.4 Cyclic Data	188
12.1 Modifying Variables	193
12.2 Mutable Lists	202
13 Predicting Growth	205
14 Halloween Analysis	216
15.1 Sharing and Equality	219

15.2 The Size of a DAG	228
16.1 Introducing Graphs	234
16.2 Basic Graph Traversals	242
16.3 Graphs With Weighted Edges	246
16.4 Shortest (or Lightest) Paths	247
16.5 Moravian Spanning Trees	249
17.1 Representing Sets as Lists	254
17.2 Making Sets Grow on Trees	259
17.3 Union-Find	266
17.4 Hashes, Sets, and Key-Values	268
17.5 Equality, Ordering, and Hashing	275
17.6 Sets as a Case Study	279
18 State and Equality	280
19 Recursion and Cycles from Mutation	285
20 Detecting Cycles	288
21 Avoiding Recomputation by Remembering Answers	292
22 Partial Domains	306
23 Staging	314
24 Factoring Numbers	318
25 Deconstructing Loops	320
26 Interactive Games as Reactive Systems	326
27 Appendices	338

I Introduction

1.1 What This Book is About This book is an introduction to computer science. It will teach you to program, and do so in ways that are of practical value and importance. However, it will also go beyond programming to computer science, a rich, deep, fascinating, and beautiful intellectual discipline. You will learn many useful things that you can apply right away, but we will also show you some of what lies beneath and beyond.

Most of all, we want to give you ways of thinking about solving problems using computation. Some of these ways are technical methods, such as working from data and examples to construct solutions to problems. Others are scientific methods, such as ways of making sure that programs are reliable and do what they claim. Finally, some are social, thinking about the impacts that programs have on people.

1.2 The Values That Drive This Book Our perspective is guided by our decades of experience as software developers, researchers, and educators. This has instilled in us the following beliefs:

- Software is not written only to be run. It must also be written to be read and maintained by others. Often, that “other” person is you, six months later, who has forgotten what they did and why.
- Programmers are responsible for their software meeting its desired goals and being reliable. This is reflected in a variety of disciplines inside computer science, such as testing and verification.
- Programs ought to be amenable to prediction. We need to know, as much as possible, before a program runs, how it will behave. This behavior includes not only technical characteristics such as running time, space, power, and so on, but also social impacts, benefits, and harms. Programmers have been notoriously poor at thinking about the latter.

1.3 Our Perspective on Data These concerns intersect with our belief about how computer science has evolved as a discipline. It is a truism that we live in a world awash with data, but what consequence does that have?

At a computational level, data have had a profound effect. Traditionally, the only way to make a program better was to improve the program directly, which often meant making it more complicated and impacting the values we discuss above. But there are classes of programs for which there is another method: simply give the same program more or better data, and the program can improve. These data-driven programs lie at the heart of many innovations we see around us.

In addition to this technical effect, data can have a profound pedagogic impact, too. Most introductory programming is plagued by artificial data that have no real meaning, interest, or consequence (and often, artificial problems to accompany them). With real data, learners can personalize their education, focusing on problems they find meaningful, enriching, or just plain fun—asking and answering questions they find worthwhile. Indeed, from this perspective, programs interrogate data: that is, programs are tools for answering questions. In turn, the emphasis on real data and real questions enables us to discuss the social impacts of computing.

These phenomena have given rise to whole new areas of study, typically called data science. However, typical data science curricula also have many limitations. They pay little attention to what we know about the difficulties of learning to program. They have little emphasis on software reliability. And they fail to recognize that their data are often quite limited in their structure. These limitations, where data science typically ends, are where computer science begins. In particular, the structure of data serve as a point of departure for thinking about and achieving some of the values above—performance, reliability, and predictability—using the many tools of computer science.

1.4 What Makes This Book Unique First, we propose a new perspective on structuring computing curricula, which we call data centricity. For more about this, read our essay. We view a data-centric curriculum as

data centric = data science + data structures

in that order: we begin with ideas from data science, before shifting to classical ideas from data structures and the rest of computer science. This book lays out this vision concretely and in detail.

Second, computing education talks a great deal about notional machines—abstractions of program behavior meant to help students understand how programs work—but few curricula actually use one. We take notional machines seriously, developing a sequence of them and weaving them through the curriculum. This ties to our belief that programs are not only objects that run, but also objects that we reason about.

Third, we weave content on socially-responsible computing into the text. Unlike other efforts that focus on exposing students to ethics or the pitfalls of technology in general, we aim to show students how the constructs and concepts that they are turning into code right now can lead to adverse impacts unless used with care. In keeping with our focus on testing and concrete examples, we introduce several topics by getting students to think about assumptions at the level of concrete data. This material is called out explicitly throughout the book.

Finally, this book is deeply informed by recent and ongoing research results. Our choices of material, order of presentation, programming methods, and more are driven by what we know from the research literature. In many cases, we ourselves are the ones doing the research, so the curriculum and research live in a symbiotic relationship. You can find our papers (some with each other, others not) on our respective pages.

1.5 Who This Book is For This book is written primarily for students who are in the early stages of computing education at the tertiary level (college or university). However, many—especially the earlier—parts of it are also suitable for secondary education (in the USA, for instance, roughly grades 6–12, or ages 12–18). Indeed, we see a natural continuum between secondary and tertiary education, and think this book can serve as a useful bridge between the two.

1.6 The Structure of This Book Unlike some other textbooks, this one does not follow a top-down narrative. Rather it has the flow of a conversation, with backtracking. We will often build up programs incrementally, just as a pair of programmers would. We will include mistakes, not because we don't know better, but because this is the best way for you to learn. Including mistakes makes it impossible for you to read passively: you must instead engage with the material, because you can never be sure of the veracity of what you're reading.

At the end, you'll always get to the right answer. However, this non-linear path is more frustrating in the short term (you will often be tempted to say, “Just tell me the answer, already!”), and it makes the book a poor reference guide (you can't open up to a random page and be sure what it says is correct). However, that feeling of frustration is the sensation of learning. We don't know of a way around it.

We use visual formatting to highlight some of these points. Thus, in several places you will encounter this:

Exercise

This is an exercise. Do try it.

This is a traditional textbook exercise. It's something you need to do on your own. If you're using this book as part of a course, this may very well have been assigned as homework. In contrast, you will also find exercise-like questions that look like this:

Do Now!

There's an activity here! Do you see it?

When you get to one of these, stop. Read, think, and formulate an answer before you proceed. You must do this because this is actually an exercise, but the answer is already in the book—most often in the text immediately following (i.e., in the part you're reading right now)—or is something you can determine for yourself by running a program. If you just read on, you'll see the answer without having thought about it (or not see it at all, if the instructions are to run a program), so you will get to neither (a) test your knowledge, nor (b) improve your intuitions. In other words, these are additional, explicit attempts to encourage active learning. Ultimately, however, we can only encourage it; it's up to you to practice it.

Specific strategies for program design and development get highlighted in boxes that look like this:

Strategy: How to ...

here's a summary of how to do something.

Finally, we also call out content on socially-responsible computing with visually distinctive regions like this:

Responsible Computing: Did you consider ...

Here are social pitfalls from using material naively.

1.7 Organization of the Material Because this book covers what would be considered multiple semesters worth of material at the tertiary level in the USA, we have divided it into seven main booklets. Later booklets depend on some earlier ones, but the earlier ones can be treated as a stand-alone book that arrives at a satisfying ending for a student or course that does not proceed further.

1. Introduction to Programming: An introduction to programming for beginners that teaches programming and rudimentary data analysis. It introduces core programming concepts through composing images and processing tables, before covering lists and trees. The notional machine throughout this section is based on substitution.

Dependencies: None!

2. From Pyret to Python: Students learn to transfer their knowledge from Pyret to Python, highlighting similarities and differences between the languages and their traditional programming styles (“paradigms”). Students are also introduced to Pandas, as a real-world table-processing system.

Dependencies: Introduction to Programming.

3. Programming With State: Students learn the subtleties of state and aliasing. Much of the coverage is in both Python and Pyret. This contrast lets students understand how multiple languages can approach the same topic; the ways in which the underlying ideas are actually the same; but also some key differences that provide insight. The notional machine grows to cover state and aliasing by separating the naming environment (here called the directory) from a heap of structured data values.

Dependencies: Introduction to Programming is essential. From Pyret to Python is helpful to follow the Python portions of this booklet, but a student can do the Pyret parts without having seen Python.

4. Algorithm Analysis: Students are introduced to multiple techniques for analyzing algorithms.

Dependencies: Introduction to Programming. There is no dependency on state.

5. Data Structures with Analysis: Students are introduced to more advanced data structures through a lens of algorithm analysis, which motivates their revision and variation.

Dependencies: Introduction to Programming and Algorithm Analysis are essential everywhere. Programming With State is necessary for some material.

6. Advanced Topics: Students cover a grab-bag of interesting computer science topics in program design and algorithmic programming. Relative to the other material, this content is either more subtle, more advanced, or less essential in a mainstream course.

Dependencies: All the material depends on Introduction to Programming. Some material depends on Algorithm Analysis and/or Programming With State.

7. Interactive Programs: Students can write interactive programs with relatively few dependencies!

Dependencies: Introduction to Programming.

This decomposition into booklets allows flexibility in offering several different kinds of courses at very different levels of sophistication. For instance, we already offer two very different courses by remixing this material, which others could follow:

- An introductory course can use Introduction to Programming, From Pyret to Python, and Programming With State to cover the data-centric view of computer science and leaving students with basic skills in Python. This corresponds to CSCI 0111 at Brown University.
- A more advanced course can start with Introduction to Programming, then do Algorithm Analysis (perhaps in increments), followed by Data Structures with Analysis, before returning to Programming With State, while interspersing content from Advanced Topics. this corresponds to CSCI 0190 at Brown University.

The course pages archive all prior instances of the courses, which include all the assignments and related materials. Readers are welcome to use these in their own courses.

Many of these courses will have entering students who have programmed with state before (in Python, Java, Scratch, or other languages). In our experience, most of these students have been given either vastly incomplete, or outright misleading, explanations of and metaphors for state (e.g., “a variable is a box”). Thus, they have a poor understanding of it beyond the absolute basics, especially when they get to important topics like aliasing. As a result, many of these students have found it both novel and insightful to properly understand how state really works through our notional machine. For that reason, we recommend going through that material slowly and carefully.

We of course invite readers to create their own mashups of the chapters within the sections. We would love to hear about others’ designs.

1.8 Our Programming Language Choice If we wanted to get rich, we’d have written this book entirely in Python. As of this writing, Python is enjoying its instructional-use heyday (just like Java before it, C++ before that, C before that, Pascal earlier, and so on). And there are, indeed, many attractive aspects of Python, not least its presence next to bullet points on job listings. However, we’ve been repeatedly frustrated by Python as an entrypoint into learning programming.

As a result, this book features two programming languages. It starts with a language, called Pyret, that we designed to address our needs and frustrations. It has been expressly designed for the style of programming in this book, so the two can grow in harmony. It draws on Python, but also on many other excellent programming languages. Beginning programmers can therefore rest in the knowledge they are being cared for, while programmers with past acquaintance of the language menagerie, from serpents to dromedaries, should find Pyret familiar and comfortable.

Then, recognizing the value of Python both as a standard language of communication and for its extensive libraries, the Programming with State (in Both Pyret and Python) part of this book explicitly covers Python. Rather than starting from scratch in Python, we present a systematic and gradual transition to it from the earlier material. We believe this will make you learn general programming better than if you had seen only one programming language. However, we believe this will help you understand Python better, too: just like you learn to appreciate your own language, country, or culture better once you’ve stepped outside and been exposed to other ones.

Programming Tools: The programming environment for Pyret is called CPO (an abbreviation of code.pyret.org). It runs entirely in the browser and uses Google Drive for authentication and file storage. We do not recommend any particular Python environment. Any Python editor that allows you to use pytest and load external data files should work fine.

1.9 Sending Us Feedback, Errors, and Comments As you work through the book, you may spot typos, notice points where we could have been clearer, or have a suggestion for a future release. You can pass these along to us by filing an issue on our public GitHub site. Thanks in advance!

1.10 Staying Up-To-Date You can subscribe to our very-low-volume mailing list, dcic-notifications.

2 Acknowledgments

This book has benefited from the attention of many.

Special thanks to the students at Brown University, who have been drafted into acting as a crucible for every iteration of this book. They have supported it with unusual grace, creating a welcoming and rewarding environment for pedagogic effort. Thanks also to our academic homes—Brown, Northeastern, and UC San Diego—for comfort and encouragement.

The following people have helpfully provided information on typos and other infelicities:

Abhabongse Janthong, Alex Hayworth, Alex Kleiman, Athyuttam Eleti, Benjamin S. Shapiro, Cheng Xie, Daniel Cai, Danil Braun, Dave Lee, David Cooper, Doug Kearns, Ebube Chuba, Evelyn Mitchell, frodokomodo (on github), Graeme McCutcheon, gregshubert (on github), Harrison Pincket, Igor Moreno Santos, Iuliu Balibanu, Jason Bennett, Jeremy Siek, jiad (from Discord), Jonathan Zhou, John (Spike) Hughes, Jon Sailor, Jonathan Zhou, Josh Paley, Kelechi Ukadike, Kendrick Cole, Kishore Vancheeshwaran,

Marc Smith, Mehmet Fatih Köksal, Michael Morehouse, Noah Tye, Rafał Gwoździński, rawalplawit (on github), Raymond Plante, Ricardo Vela, Samuel Ainsworth, Samuel Kortchmar, timotree (on github).

The following have done the same, but in much greater quantity or depth:

Dorai Sitaram, John Palmer, Kartik Singhal, Kenichi Asai, Lev Litichevskiy.

Even amongst the problem-spotters, one is hors catégorie:

Sorawee Porncharoenwase.

This book is completely dependent on Pyret, and thus on the many people who have created and sustained it.

We thank Matthew Butterick for his help with book styling (though the ultimate style is ours, so don't blame him!).

Many, many years ago, Alejandro Schäffer introduced SK to the idea of nature as a fat-fingered typist. Alejandro's fingerprints are over many parts of this book, even if he wouldn't necessarily approve of what has come of his patient instruction.

We are deeply inspired by the work and ideas of Matthias Felleisen, Matthew Flatt, and Robby Findler. Matthias, in particular, inspired our ideas on program design. Even where we disagree, he continues to engage with and challenge our ideas in ways that force us to grow and improve. Our work is better than it would be in incalculable ways due to his influence.

The chapter on Interactive Games as Reactive Systems is translated from How to Design Worlds, and owes thanks to all the people acknowledged there.

This book is written in Scribble, the authoring tool of choice for the discerning programmer.

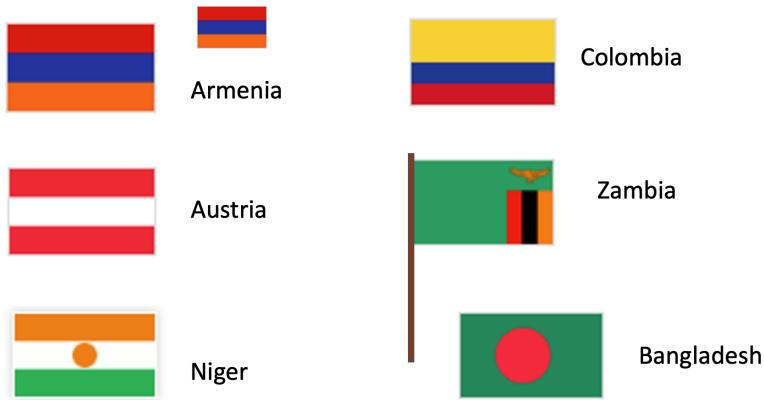
We thank cloudconvert for their free conversion tools.

contents ← prev up next →

3.1 Getting Started

3.1 Getting Started

3.1.1 Motivating Example: Flags Imagine that you are starting a graphic design company, and want to be able to create images of flags of different sizes and configurations for your customers. The following diagram shows a sample of the images that your software will have to help you create:



Before we try to write code to create these different images, you should step back, look at this collection of images, and try to identify features of the images that might help us decide what to do. To help with this, we're going to answer a pair of specific questions to help us make sense of the images:

- What do you notice about the flags?
- What do you wonder about the flags or a program that might produce them?

Do Now!

Actually write down your answers. Noticing features of data and information is an essential skill in computing.

Some things you might have noticed:

- Some flags have similar structure, just with different colors
- Some flags come in different sizes
- Some flags have poles
- Most of these look pretty simple, but some real flags have complicated figures on them

...and so on.

Some things you might have wondered:

- Do I need to be able to draw these images by hand?
- Will we be able to generate different sized flags from the same code?
- What if we have a non-rectangular flag?

...and so on.

The features that we noticed suggest some things we'll need to be able to do to write programs to generate flags:

- We might want to compute the heights of the stripes from the overall flag dimensions (we'll write programs using numbers)
- We need a way to describe colors to our program (we'll learn strings)
- We need a way to create images based on simple shapes of different colors (we'll create and combine expressions)

Let's get started!

3.1.2 Numbers

Start simple: compute the sum of 3 and 5.

To do this computation with a computer, we need to write down the computation and ask the computer to run or evaluate the computation so that we get a number back. A software or web-application in which you write and run programs is called a programming environment. In the first part of this course, we will use a language called Pyret.

Go to the on-line editor (which we'll henceforth refer to as "CPO"). For now, we will work only in the right-hand side (the interactions pane).

The `>>>` is called the "prompt"—that's where we tell CPO to run a program. Let's tell it to add 3 and 5. Here's what we write:

```
3 + 5
```

Press the Return key, and the result of the computation will appear on the line below the prompt, as shown below:

```
3 + 5
```

```
8
```

Not surprisingly, we can do other arithmetic computations

```
2 * 6
```

```
12
```

(Note: `*` is how we write the multiplication sign.)

What if we try `3 + 4 * 5`?

Do Now!

Try it! See what Pyret says.

Pyret gave you an error message. What it says is that Pyret isn't sure whether we mean

```
(3 + 4) * 5
```

or

```
3 + (4 * 5)
```

so it asks us to include parentheses to make that explicit. Every programming language has a set of rules about how you have to write down programs. Pyret's rules require parentheses to avoid ambiguity.

```
(3 + 4) * 5
```

```
35
```

```
3 + (4 * 5)
```

```
23
```

Another Pyret rule requires spaces around the arithmetic operators. See what happens if you forget the spaces:

```
3+4
```

Pyret will show a different error message that highlights the part of the code that isn't formatted properly, along with an explanation of the issue that Pyret has detected. To fix the error, you can press the up-arrow key within the right pane and edit the previous computation to add the spaces.

Do Now!

Try doing it right now, and confirm that you succeeded!

What if we want to get beyond basic arithmetic operators? Let's say we want the minimum of two numbers. We'd write this as

```
num-min(2, 8)
```

Why `num-`? It's because "minimum" is a concept that makes sense on data other than numbers; Pyret calls the `min` operator `num-min` to avoid ambiguity.

3.1.3 Expressions Note that when we run `num-min`, we get a number in return (as we did for `+`, `*`, ...). This means we should be able to use the result of `num-min` in other computations where a number is expected:

```
5 * num-min(2, 8)
```

10

```
(1 + 5) * num-min(2, 8)
```

12

Hopefully you are starting to see a pattern. We can build up more complicated computations from smaller ones, using operations to combine the results from the smaller computations. We will use the term expression to refer a computation written in a format that Pyret can understand and evaluate to an answer.

Exercise

In CPO, try to write the expressions for each of the following computations:

- subtract 3 from 7, then multiply the result by 4
- subtract 3 from the multiplication of 7 and 4
- the sum of 3 and 5, divided by 2
- the max of 5 - 10 and -20
- 2 divided by the sum of 3 and 5

Do Now!

What if you get a fraction as a response?

If you're not sure how to get a fraction, there are two ways: you can either write an expression that produces a fractional answer, or you can type one in directly (e.g., `1/3`).

Either way, you can click on the result in the interactions pane to change how the number is presented. Try it!

3.1.4 Terminology Look at an interaction like

```
(3 + 4) * (5 + 1)
```

42

There are actually several kinds of information in this interaction, and we should give them names:

- Expression: a computation written in the formal notation of a programming language

Examples here include `4`, `5 + 1`, and `(3 + 4) * (5 + 1)`

- Value: an expression that can't be computed further (it is its own result)

So far, the only values we've seen are numbers.

- Program: a sequence of expressions that you want to run

3.1.5 Strings What if we wanted to write a program that used information other than numbers, such as someone’s name? For names and other text-like data, we use what are called strings. Here are some examples:

```
"Kathi"
"Go Bears!"
"CSCI0111"
"Carberry, Josiah"
```

What do we notice? Strings can contain spaces, punctuation, and numbers. We use them to capture textual data. For our flags example, we’ll use strings to name colors: “red”, “blue”, etc.

Note that strings are case-sensitive, meaning that capitalization matters (we’ll see where it matters shortly).

3.1.6 Images We have seen two kinds of data: numbers and strings. For flags, we’ll also need images. Images are different from both numbers and strings (you can’t describe an entire image with a single number—well, not unless you get much farther into computer science but let’s not get ahead of ourselves).

Pyret has built-in support for images. When you start up Pyret, you’ll see a grayed-out line that says “use context essentials2021” (or something similar). This line configures Pyret with some basic functionality beyond basic numbers and strings.

Do Now!

Press the “Run” button (to activate the features in essentials), then write each of these Pyret expressions at the interactions prompt to see what they produce:

- `circle(30, "solid", "red")`
- `circle(30, "outline", "blue")`
- `rectangle(20, 10, "solid", "purple")`

Each of these expressions names the shape to draw, then configures the shape in the parentheses that follow. The configuration information consists of the shape dimensions (the radius for circles, the width and height for rectangles, both measured in screen pixels), a string indicating whether to make a solid shape or just an outline, then a string with the color to use in drawing the shape.

Which shapes and colors does Pyret know about? Hold this question for just a moment. We’ll show you how to look up information like this in the documentation shortly.

3.1.6.1 Combining Images Earlier, we saw that we could use operations like `+` and `*` to combine numbers through expressions. Any time you get a new kind of datum in programming, you should ask what operations the language gives you for working with that data. In the case of images in Pyret, the collection includes the ability to:

- rotate them
- scale them
- flip them
- put two of them side by side
- place one on top of the other
- and more ...

Let’s see how to use some of these.

Exercise

Type the following expressions into Pyret:

```
rotate(45, rectangle(20, 30, "solid", "red"))
```

What does the `45` represent? Try some different numbers in place of the `45` to confirm or refine your hypothesis.

```
overlay(circle(25, "solid", "yellow"),
        rectangle(50, 50, "solid", "blue"))
```

Can you describe in prose what `overlay` does?

```
above(circle(25, "solid", "red"),
      rectangle(30, 50, "solid", "blue"))
```

What kind of value do you get from using the `rotate` or `above` operations? (hint: your answer should be one of number, string, or image)

These examples let us think a bit deeper about expressions. We have simple values like numbers and strings. We have operations or functions that combine values, like `+` or `rotate` (“functions” is the term more commonly used in computing, whereas your math classes likely used “operations”). Every function produces a value, which can be used as input to another function. We build up expressions by using values and the outputs of functions as inputs to other functions.

For example, we used `above` to create an image out of two smaller images. We could take that image and rotate it using the following expression.

```
rotate(45,
       above(circle(25, "solid", "red"),
             rectangle(30, 50, "solid", "blue"))))
```

This idea of using the output of one function as input to another is known as composition. Most interesting programs arise from composing results from one computation with another. Getting comfortable with composing expressions is an essential first step in learning to program.

Exercise

Try to create the following images:

- a blue triangle (you pick the size). As with `circle`, there is a `triangle` function that takes a side length, fill style, and color and produces an image of an equilateral triangle.
- a blue triangle inside a yellow rectangle
- a triangle oriented at an angle
- a bullseye with 3 nested circles aligned in their centers (e.g., the Target logo)
- whatever you want—play around and have fun!

The bullseye might be a bit challenging. The `overlay` function only takes two images, so you’ll need to think about how to use composition to layer three circles.

3.1.6.2 Making a Flag We’re ready to make our first flag! Let’s start with the flag of Armenia, which has three horizontal stripes: red on top, blue in the middle, and orange on the bottom.

Exercise

Use the functions we have learned so far to create an image of the Armenian flag. You pick the dimensions (we recommend a width between 100 and 300).

Make a list of the questions and ideas that occur to you along the way.

3.1.7 Stepping Back: Types, Errors, and Documentation Now that you have an idea of how to create a flag image, let’s go back and look a bit more carefully at two concepts that you’ve already encountered: types and error messages.

3.1.7.1 Types and Contracts Now that we are composing functions to build more complicated expressions out of smaller ones, we will have to keep track of which combinations make sense. Consider the following sample of Pyret code:

```
8 * circle(25, "solid", "red")
```

What value would you expect this to produce? Multiplication is meant to work on numbers, but this code asks Pyret to multiply a number and an image. Does this even make sense?

This code does not make sense, and indeed Pyret will produce an error message if we try to run it.

Do Now!

Try to run that code, then look at the error message. Write down the information that the error message is giving you about what went wrong (we'll come back to your list shortly).

The bottom of the error message says:

The `*` operator expects to be given two Numbers

Notice the word “Numbers”. Pyret is telling you what kind of information works with the `*` operation. In programming, values are organized into types (e.g., number, string, image). These types are used in turn to describe what kind of inputs and results (a.k.a., outputs) a function works with. For example, `*` expects to be given two numbers, from which it will return a number. The last expression we tried violated that expectation, so Pyret produced an error message.

Talking about “violating expectations” sounds almost legal, doesn’t it? It does, and the term contract refers to the required types of inputs and promised types of outputs when using a specific function. Here are several examples of Pyret contracts (written in the notation you will see in the documentation):

```
* :: (x1 :: Number, x2 :: Number) -> Number
```

```
circle :: (radius :: Number,
           mode :: String,
           color :: String) -> Image
```

```
rotate :: (degrees :: Number,
            img :: Image) -> Image
```

```
overlay :: (upper-img :: Image,
            lower-img :: Image) -> Image
```

Do Now!

Look at the notation pattern across these contracts. Can you label the various parts and what information they appear to be giving you?

Let’s look closely at the `overlay` contract to make sure you understand how to read it. It gives us several pieces of information:

- There is a function called `overlay`
- It takes two inputs (the parts within the parentheses), both of which have the type `Image`
- The first input is the image that will appear on top
- The second input is the image that will appear on the bottom
- The output from calling the function (which follows `->`) will have type `Image`

In general, we read the double-colon (`::`) as “has the type”. We read the arrow (`->`) as “returns”.

Whenever you compose smaller expressions into more complex expressions, the types produced by the smaller expressions have to match the types required by the function you are using to compose them. In the case of our erroneous `*` expression, the contract for `*` expects two numbers as inputs, but we gave an image for the second input. This resulted in an error message when we tried to run the expression.

A contract also summarizes how many inputs a function expects. Look at the contract for the `circle` function. It expects three inputs: a number (for the radius), a string (for the style), and a string (for the color). What if we forgot the style string, and only provided the radius and color, as in:

```
circle(100, "purple")
```

The error here is not about the type of the inputs, but rather about the number of inputs provided.

Exercise

Run some expressions in Pyret that use an incorrect type for some input to a function. Run others where you provide the wrong number of inputs to a function.

What text is common to the incorrect-type errors? What text is common to the wrong numbers of inputs?

Take note of these so you can recognize them if they arise while you are programming.

3.1.7.2 Format and Notation Errors We've just seen two different kinds of mistakes that we might make while programming: providing the wrong type of inputs and providing the wrong number of inputs to a function. You've likely also run into one additional kind of error, such as when you make a mistake with the punctuation of programming. For example, you might have typed an example such as these:

- `3+7`
- `circle(50 "solid" "red")`
- `circle(50, "solid, "red")`
- `circle(50, "solid," "red")`
- `circle 50, "solid," "red")`

Do Now!

Make sure you can spot the error in each of these! Evaluate these in Pyret if necessary.

You already know various punctuation rules for writing prose. Code also has punctuation rules, and programming tools are strict about following them. While you can leave out a comma and still turn in an essay, a programming environment won't be able to evaluate your expressions if they have punctuation errors.

Do Now!

Make a list of the punctuation rules for Pyret code that you believe you've encountered so far.

Here's our list:

- Spaces are required around arithmetic operators.
- Parentheses are required to indicate order of operations.
- When we use a function, we put a pair of parentheses around the inputs, and we separate the inputs with commas.
- If we use a double-quotation mark to start a string, we need another double-quotation mark to close that string.

In programming, we use the term **syntax** to refer to the rules of writing proper expressions (we explicitly didn't say "rules of punctuation" because the rules go beyond what you think of as punctuation, but that's a fair place to start). Making mistakes in your syntax is common at first. In time, you'll internalize the rules. For now, don't get discouraged if you get errors about syntax from Pyret. It's all part of the learning process.

3.1.7.3 Finding Other Functions: Documentation At this point, you may be wondering what else you can do with images. We mentioned scaling images. What other shapes might we make? Is there a list somewhere of everything we can do with images?

Every programming language comes with documentation, which is where you find out the various operations and functions that are available, and your options for configuring their parameters. Documentation can be overwhelming

for novice programmers, because it contains a lot of detail that you don't even know that you need just yet. Let's take a look at how you can use the documentation as a beginner.

Open the Pyret Image Documentation. Focus on the sidebar on the left. At the top, you'll see a list of all the different topics covered in the documentation. Scroll down until you see "rectangle" in the sidebar: surrounding that, you'll see the various function names you can use to create different shapes. Scroll down a bit further, and you'll see a list of functions for composing and manipulating images.

If you click on a shape or function name, you'll bring up details on using that function in the area on the right. You'll see the contract in a shaded box, a description of what the function does (under the box), and then a concrete example or two of what you type to use the function. You could copy and paste any of the examples into Pyret to see how they work (changing the inputs, for example).

For now, everything you need documentation wise is in the section on images. We'll go further into Pyret and the documentation as we go.

contents ← prev up next →

3.2 Naming Values

3.2 Naming Values

3.2.1 The Definitions Pane So far, we have only used the interactions pane on the right half of the CPO screen. As we have seen, this pane acts like a calculator: you type an expression at the prompt and CPO produces the result of evaluating that expression.

The left pane is called the definitions pane. This is where you can put code that you want to save to a file. It has another use, too: it can help you organize your code as your expressions get larger.

3.2.2 Naming Values The expressions that create images involve a bit of typing. It would be nice to have shorthands so we can “name” images and refer to them by their names. This is what the definitions pane is for: you can put expressions and programs in the definitions pane, then use the “Run” button in CPO to make the definitions available in the interactions pane.

Do Now!

Put the following in the definitions pane:

```
red-circ = circle(30, "solid", "red")
```

Hit run, then enter `red-circ` in the interactions pane. You should see the red circle.

More generally, if you write code in the form:

```
NAME = EXPRESSION
```

Pyret will associate the value of `EXPRESSION` with `NAME`. Anytime you write the (shorthand) `NAME`, Pyret will automatically (behind the scenes) replace it with the value of `EXPRESSION`. For example, if you write `x = 5 + 4` at the prompt, then write `x`, CPO will give you the value `9` (not the original `5 + 4` expression).

What if you enter a name at the prompt that you haven’t associated with a value?

Do Now!

Try typing `puppy` at the interactions pane prompt (`>>>`). Are there any terms in the error message that are unfamiliar to you?

CPO (and indeed many programming tools) use the phrase “unbound identifier” when an expression contains a name that has not been associated with (or bound to) a value.

3.2.2.1 Names Versus Strings At this point, we have seen words being used in two ways in programming: (1) as data within strings and (2) as names for values (also called identifiers). These are two very different uses, so it is worth reviewing them.

- Syntactically (another way of saying “in terms of how we write it”), we distinguish strings and names by the presence of double quotation marks. Note the difference between `puppy` and `"puppy"`.
- Strings can contain spaces, but names cannot. For example, `"hot pink"` is a valid piece of data, but `hot pink` is not a single name. When you want to combine multiple words into a name (like we did above with `red-circ`), use a hyphen to separate the words while still having a single name (as a sequence of characters). Different programming languages allow different separators; for Pyret, we’ll use hyphens.

- Entering a word as a name versus as a string at the interactions prompt changes the computation that you are asking Pyret to perform. If you enter `puppy` (the name, without double quotes), you are asking Pyret to lookup the value that you previously stored under that name. If you enter `"puppy"` (the string, with double quotes) you are simply writing down a piece of data (akin to typing a number like `3`): Pyret returns the value you entered as the result of the computation.
- If you enter a name that you have not previously associated with a value, Pyret will give you an “unbound identifier” error message. In contrast, since strings are just data, you won’t get an error for writing a previously-unused string (there are some special cases of strings, such as when you want to put a quotation mark inside them, but we’ll set that aside for now).

Novice programmers frequently confuse names and strings at first. For now, remember that the names you associate with values using `=` cannot contain quotation marks, while word- or text-based data must be wrapped in double quotes.

3.2.2.2 Expressions versus Statements Definitions and expressions are two useful aspects of programs, each with their own role. Definitions tell Pyret to associate names with values. Expressions tell Pyret to perform a computation and return the result.

Exercise

Enter each of the following at the interactions prompt:

- `5 + 8`
- `x = 14 + 16`
- `triangle(20, "solid", "purple")`
- `blue-circ = circle(x, "solid", "blue")`

The first and third are expressions, while the second and fourth are definitions. What do you observe about the results of entering expressions versus the results of entering definitions?

Hopefully, you notice that Pyret doesn’t seem to return anything from the definitions, but it does display a value from the expressions. In programming, we distinguish expressions, which yield values, from statements, which don’t yield values but instead give some other kind of instruction to the language. So far, definitions are the only kinds of statements we’ve seen.

Exercise

Assuming you still have the `blue-circ` definition from above in your interactions pane, enter `blue-circ` at the prompt (you can re-enter that definition if it is no longer there).

Based on what Pyret does in response, is `blue-circ` an expression or a definition?

Since `blue-circ` yielded a result, we infer that a name by itself is also an expression. This exercise highlights the difference between making a definition and using a defined name. One produces a value while the other does not. But surely something must happen, somewhere, when you run a definition. Otherwise, how could you use that name later?

3.2.3 The Program Directory Programming tools do work behind the scenes as they run programs. Given the program `2 + 3`, for example, a calculation takes place to produce `5`, which in turn displays in the interactions pane.

When you write a definition, Pyret makes an entry in an internal directory in which it associates names with values. You can’t see the directory, but Pyret uses it to manage the values that you’ve associated with names. If you write:

```
width = 30
```

Pyret makes a new directory entry for `width` and records that `width` has value `30`. If you then write

```
height = width * 3
```

Pyret evaluates the expression on the right side (`width * 3`), then stores the resulting value (here, `90`) alongside `height` in the directory.

How does Pyret evaluate `(width * 3)`? Since `width` is a word (not a string), Pyret looks up its value in the directory. Pyret substitutes that value for the name in the expression, resulting in `30 * 3`, which then evaluates to 90. After running these two expressions, the directory looks like:

Directory

- `width`

→

30

- `height`

→

90

Note that the entry for `height` in the directory has the result of `width * 3`, not the expression. This will become important as we use named values to prevent us from doing the same computation more than once.

The program directory is an essential part of how programs evaluate. If you are trying to track how your program is working, it sometimes helps to track the directory contents on a sheet of paper (since you can't view Pyret's directory).

Exercise

Imagine that you have the following code in the definitions pane when you press the Run button:

```
name = "Matthias"  
"name"
```

What appears in the interactions pane? How does each of these lines interact with the program directory?

Exercise

What happens if you enter a subsequent definition for the same name, such as `width = 50`? How does Pyret respond? What if you then ask to see the value associated with this same name at the prompt? What does this tell you about the directory?

When you try to give a new value to a name that is already in the directory, Pyret will respond that the new definition "shadows a previous declaration" of that same name. This is Pyret's way of warning you that the name is already in the directory. If you ask for the value associated with the name again, you'll see that it still has the original value. Pyret doesn't let you change the value associated with an existing name with the `name = value` notation. While there is a notation that will let you reassign values, we won't work with this concept until Modifying Variables.

3.2.3.1 Understanding the Run Button Now that we've learned about the program directory, let's us discuss what happens when you press the Run button. Let's assume the following contents are in the definitions pane:

```
width = 30  
height = width * 3  
blue-rect = rectangle(width, height, "solid", "blue")
```

When you press Run, Pyret first clears out the program directory. It then processes your file line by line, starting at the top. If you have an `include` statement, Pyret adds the definitions from the included library to the directory. After processing all of the lines for this program, the directory will look like:

Directory

- `circle`

→ <the circle operation>

- `rectangle`
→ <the rectangle operation>
- ...
- `width`
→
30
- `height`
→
90
- `blue-rect`
→ <the actual rectangle image>

If you now type at the interactions prompt, any use of an identifier (a sequence of characters not enclosed in quotation marks) results in Pyret consulting the directory.

If you now type

```
beside(blue-rect, rectangle(20, 20, "solid", "purple"))
```

Pyret will look up the image associated with `blue-rect`.

Do Now!

Is the purple rectangle in the directory? What about the image with the two rectangles?

Neither of these shapes is in the directory. Why? We didn't ask Pyret to store them there with a name. What would be different if we instead wrote the following (at the interactions prompt)?

```
two-rects = beside(blue-rect, rectangle(20, 20, "solid", "purple"))
```

Now, the two-shape image would be in the directory, associated with the name `two-rects`. The purple rectangle by itself, however, still would not be stored in the directory. We could, however, reference the two-shape image by name, as shown below:

The screenshot shows the Pyret interface with a code editor and an interactions pane. The code editor contains the following Python-like code:

```

1 use context essentials2021
2
3 width = 30
4 height = width * 3
5 blue-rect =
6   rectangle(width, height, "solid", "blue")

```

The interactions pane shows the result of running this code:

```

>>> two-rects = beside(blue-rect, rectangle(20, 20, "solid", "purple"))

>>> two-rects

```

Below the code editor, there is a preview area showing a blue rectangle next to a smaller purple rectangle. A cursor is visible in the interactions pane.

Do Now!

Imagine that we now hit the Run button again, then typed `two-rects` at the interactions prompt. How would Pyret respond and why?

3.2.4 Using Names to Streamline Building Images The ability to name values can make it easier to build up complex expressions. Let's put a rotated purple triangle inside a green square:

```
overlay(rotate(45, triangle(30, "solid", "purple")),
        rectangle(60, 60, "solid", "green"))
```

However, this can get quite difficult to read and understand. Instead, we can name the individual shapes before building the overall image:

```
purple-tri = triangle(30, "solid", "purple")
green-sqr = rectangle(60, 60, "solid", "green")

overlay(rotate(45, purple-tri),
        green-sqr)
```

In this version, the `overlay` expression is quicker to read because we gave descriptive names to the initial shapes.

Go one step further: let's add another purple-triangle on top of the existing image:

```
purple-tri = triangle(30, "solid", "purple")
green-sqr = rectangle(60, 60, "solid", "green")
```

```
above(purple-tri,
      overlay(rotate(45, purple-tri),
              green-sqr))
```

Here, we see a new benefit to leveraging names: we can use `purple-tri` twice in the same expression without having to write out the longer `triangle` expression more than once.

Exercise

Assume that your definitions pane contained only this most recent code example (including the `purple-tri` and `green-sqr` definitions). How many separate images would appear in the interactions pane if you pressed Run? Do you see the purple triangle and green square on their own, or only combined? Why or why not?

Exercise

Re-write your expression of the Armenian flag (from Making a Flag), this time giving intermediate names to each of the stripes.

In practice, programmers don't name every individual image or expression result when creating more complex expressions. They name ones that will get used more than once, or ones that have particular significance for understanding their program. We'll have more to say about naming as our programs get more complicated.

contents ← prev up next →

3.3 From Repeated Expressions to Functions

3.3 From Repeated Expressions to Functions

3.3.1 Example: Similar Flags Consider the following two expressions to draw the flags of Armenia and Austria (respectively). These two countries have the same flag, just with different colors. The `frame` operator draws a small black frame around the image.

```
# Lines starting with # are comments for human readers.  
# Pyret ignores everything on a line after #.  
  
# armenia  
frame(  
    above(rectangle(120, 30, "solid", "red"),  
        above(rectangle(120, 30, "solid", "blue"),  
            rectangle(120, 30, "solid", "orange"))))  
  
# austria  
frame(  
    above(rectangle(120, 30, "solid", "red"),  
        above(rectangle(120, 30, "solid", "white"),  
            rectangle(120, 30, "solid", "red"))))
```

Rather than write this program twice, it would be nice to write the common expression only once, then just change the colors to generate each flag. Concretely, we'd like to have a custom operator such as `three-stripe-flag` that we could use as follows:

```
# armenia  
three-stripe-flag("red", "blue", "orange")  
  
# austria  
three-stripe-flag("red", "white", "red")
```

In this program, we provide `three-stripe-flag` only with the information that customizes the image creation to a specific flag. The operation itself would take care of creating and aligning the rectangles. We want to end up with the same images for the Armenian and Austrian flags as we would have gotten with our original program. Such an operator doesn't exist in Pyret: it is specific only to our application of creating flag images. To make this program work, then, we need the ability to add our own operators (henceforth called functions) to Pyret.

3.3.2 Defining Functions In programming, a function takes one or more (configuration) parameters and uses them to produce a result.

Strategy: Creating Functions From Expressions

If we have multiple concrete expressions that are identical except for a couple of specific data values, we create a function with the common code as follows:

- Write down at least two expressions showing the desired computation (in this case, the expressions that produce the Armenian and Austrian flags).
- Identify which parts are fixed (i.e., the creation of rectangles with dimensions 120 and 30, the use of `above` to stack the rectangles) and which are changing (i.e., the stripe colors).

- For each changing part, give it a name (say `top`, `middle`, and `bottom`), which will be the parameter that stands for that part.
- Rewrite the examples to be in terms of these parameters. For example:

```
frame(
    above(rectangle(120, 30, "solid", top),
        above(rectangle(120, 30, "solid", middle),
            rectangle(120, 30, "solid", bottom))))
```

- Name the function something suggestive: e.g., `three-stripe-flag`.
- Write the syntax for functions around the expression:

```
fun <function name>(<parameters>):
    <the expression goes here>
end
```

where the expression is called the body of the function. (Programmers often use angle brackets to say “replace with something appropriate”; the brackets themselves aren’t part of the notation.)

Here’s the end product:

```
fun three-stripe-flag(top, middle, bot):
    frame(
        above(rectangle(120, 30, "solid", top),
            above(rectangle(120, 30, "solid", middle),
                rectangle(120, 30, "solid", bot))))
end
```

While this looks like a lot of work now, it won’t once you get used to it. We will go through the same steps over and over, and eventually they’ll become so intuitive that you won’t need to start from multiple similar expressions.

Do Now!

Why does the function body have only one expression, when before we had a separate one for each flag?

We have only one expression because the whole point was to get rid of all the changing parts and replace them with parameters.

With this function in hand, we can write the following two expressions to generate our original flag images:

```
three-stripe-flag("red", "blue", "orange")
three-stripe-flag("red", "white", "red")
```

When we provide values for the parameters of a function to get a result, we say that we are calling the function. We use the term call for expressions of this form.

If we want to name the resulting images, we can do so as follows:

```
armenia = three-stripe-flag("red", "blue", "orange")
austria = three-stripe-flag("red", "white", "red")
```

(Side note: Pyret only allows one value per name in the directory. If your file already had definitions for the names `armenia` or `austria`, Pyret will give you an error at this point. You can use a different name (like `austria2`) or comment out the original definition using `#.`)

3.3.2.1 How Functions Evaluate

So far, we have learned three rules for how Pyret processes your program:

- If you write an expression, Pyret evaluates it to produce its value.
- If you write a statement that defines a name, Pyret evaluates the expression (right side of `=`), then makes an entry in the directory to associate the name with the value.

- If you write an expression that uses a name from the directory, Pyret substitutes the name with the corresponding value.

Now that we can define our own functions, we have to consider two more cases: what does Pyret do when you define a function (using `fun`), and what does Pyret do when you call a function (with values for the parameters)?

- When Pyret encounters a function definition in your file, it makes an entry in the directory to associate the name of the function with its code. The body of the function does not get evaluated at this time.
- When Pyret encounters a function call while evaluating an expression, it replaces the call with the body of the function, but with the parameter values substituted for the parameter names in the body. Pyret then continues to evaluate the body with the substituted values.

As an example of the function-call rule, if you evaluate

```
three-stripe-flag("red", "blue", "orange")
```

Pyret starts from the function body

```
frame(
  above(rectangle(120, 30, "solid", top),
    above(rectangle(120, 30, "solid", middle),
      rectangle(120, 30, "solid", bot))))
```

substitutes the parameter values

```
frame(
  above(rectangle(120, 30, "solid", "red"),
    above(rectangle(120, 30, "solid", "blue"),
      rectangle(120, 30, "solid", "orange"))))
```

then evaluates the expression, producing the flag image.

Note that the second expression (with the substituted values) is the same expression we started from for the Armenian flag. Substitution restores that expression, while still allowing the programmer to write the shorthand in terms of `three-stripe-flag`.

3.3.2.2 Type Annotations

What if we made a mistake, and tried to call the function as follows:

```
three-stripe-flag(50, "blue", "red")
```

Do Now!

What do you think Pyret will produce for this expression?

The first parameter to `three-stripe-flag` is supposed to be the color of the top stripe. The value 50 is not a string (much less a string naming a color). Pyret will substitute 50 for `top` in the first call to `rectangle`, yielding the following:

```
frame(
  above(rectangle(120, 30, "solid", 50),
    above(rectangle(120, 30, "solid", "blue"),
      rectangle(120, 30, "solid", "red"))))
```

When Pyret tries to evaluate the `rectangle` expression to create the top stripe, it generates an error that refers to that call to `rectangle`.

If someone else were using your function, this error might not make sense: they didn't write an expression about rectangles. Wouldn't it be better to have Pyret report that there was a problem in the use of `three-stripe-flag` itself?

As the author of `three-stripe-flag`, you can make that happen by annotating the parameters with information about the expected type of value for each parameter. Here's the function definition again, this time requiring the three parameters to be strings:

```

fun three-stripe-flag(top :: String,
    mid :: String,
    bot :: String):
frame(
    above(rectangle(120, 30, "solid", top),
        above(rectangle(120, 30, "solid", mid),
            rectangle(120, 30, "solid", bot))))
end

```

Notice that the notation here is similar to what we saw in contracts within the documentation: the parameter name is followed by a double-colon (::) and a type name (so far, one of `Number`, `String`, or `Image`). Putting each parameter on its own line is not required, but it sometimes helps with readability.

Run your file with this new definition and try the erroneous call again. You should get a different error message that is just in terms of `three-stripe-flag`.

It is also common practice to add a type annotation that captures the type of the function's output. That annotation goes after the list of parameters:

```

fun three-stripe-flag(top :: String,
    mid :: String,
    bot :: String) -> Image:
frame(
    above(rectangle(120, 30, "solid", top),
        above(rectangle(120, 30, "solid", mid),
            rectangle(120, 30, "solid", bot))))
end

```

Note that all of these type annotations are optional. Pyret will run your program whether or not you include them. You can put type annotations on some parameters and not others; you can include the output type but not any of the parameter types. Different programming languages have different rules about types.

We will think of types as playing two roles: giving Pyret information that it can use to focus error messages more accurately, and guiding human readers of programs as to the proper use of user-defined functions.

3.3.2.3 Documentation Imagine that you opened your program file from this chapter a couple of months from now. Would you remember what computation `three-stripe-flag` does? The name is certainly suggestive, but it misses details such as that the stripes are stacked vertically (rather than horizontally) and that the stripes are equal height. Function names aren't designed to carry this much information.

Programmers also annotate a function with a docstring, a short, human-language description of what the function does. Here's what the Pyret docstring might look like for `three-stripe-flag`:

```

fun three-stripe-flag(top :: String,
    middle :: String,
    bot :: String) -> Image:
doc: "produce image of flag with three equal-height horizontal stripes"
frame(
    above(rectangle(120, 30, "solid", top),
        above(rectangle(120, 30, "solid", middle),
            rectangle(120, 30, "solid", bot))))
end

```

While docstrings are also optional from Pyret's perspective, you should always provide one when you write a function. They are extremely helpful to anyone who has to read your program, whether that is a co-worker, grader...or yourself, a couple of weeks from now.

3.3.3 Functions Practice: Moon Weight Suppose we're responsible for outfitting a team of astronauts for lunar exploration. We have to determine how much each of them will weigh on the Moon's surface. On the Moon, objects weigh only one-sixth their weight on earth. Here are the expressions for several astronauts (whose weights are expressed in pounds):

```
100 * 1/6
150 * 1/6
90 * 1/6
```

As with our examples of the Armenian and Austrian flags, we are writing the same expression multiple times. This is another situation in which we should create a function that takes the changing data as a parameter but captures the fixed computation only once.

In the case of the flags, we noticed we had written essentially the same expression more than once. Here, we have a computation that we expect to do multiple times (once for each astronaut). It's boring to write the same expression over and over again. Besides, if we copy or re-type an expression multiple times, sooner or later we're bound to make a transcription error. This is an instance of the DRY principle, where DRY means "don't repeat yourself".

Let's remind ourselves of the steps for creating a function:

- Write down some examples of the desired calculation. We did that above.
- Identify which parts are fixed (above, `* 1/6`) and which are changing (above, `100, 150, 90...`).
- For each changing part, give it a name (say `earth-weight`), which will be the parameter that stands for it.
- Rewrite the examples to be in terms of this parameter:

```
earth-weight * 1/6
```

This will be the body, i.e., the expression inside the function.

- Come up with a suggestive name for the function: e.g., `moon-weight`.
- Write the syntax for functions around the body expression:

```
fun moon-weight(earth-weight):
    earth-weight * 1/6
end
```

- Remember to include the types of the parameter and output, as well as the documentation string. This yields the final function:

```
fun moon-weight(earth-weight :: Number) -> Number:
    doc: "Compute weight on moon from weight on earth"
    earth-weight * 1/6
end
```

3.3.4 Documenting Functions with Examples In each of the functions above, we've started with some examples of what we wanted to compute, generalized from there to a generic formula, turned this into a function, and then used the function in place of the original expressions.

Now that we're done, what use are the initial examples? It seems tempting to toss them away. However, there's an important rule about software that you should learn: Software Evolves. Over time, any program that has any use will change and grow, and as a result may end up producing different values than it did initially. Sometimes these are intended, but sometimes these are a result of mistakes (including such silly but inevitable mistakes like accidentally adding or deleting text while typing). Therefore, it's always useful to keep those examples around for future reference, so you can immediately be alerted if the function deviates from the examples it was supposed to generalize.

Pyret makes this easy to do. Every function can be accompanied by a `where` clause that records the examples. For instance, our `moon-weight` function can be modified to read:

```
fun moon-weight(earth-weight :: Number) -> Number:
    doc: "Compute weight on moon from weight on earth"
    earth-weight * 1/6
where:
    moon-weight(100) is 100 * 1/6
    moon-weight(150) is 150 * 1/6
    moon-weight(90) is 90 * 1/6
```

```
end
```

When written this way, Pyret will actually check the answers every time you run the program, and notify you if you have changed the function to be inconsistent with these examples.

Do Now!

Check this! Change the formula—for instance, replace the body of the function with

```
earth-weight * 1/3
```

—and see what happens. Pay attention to the output from CPO: you should get used to recognizing this kind of output.

Do Now!

Now, fix the function body, and instead change one of the answers—e.g., write

```
moon-weight(90) is 90 * 1/3
```

—and see what happens. Contrast the output in this case with the output above.

Of course, it's pretty unlikely you will make a mistake with a function this simple (except through a typo). After all, the examples are so similar to the function's own body. Later, however, we will see that the examples can be much simpler than the body, and there is a real chance for things to get inconsistent. At that point, the examples become invaluable in making sure we haven't made a mistake in our program. In fact, this is so valuable in professional software development that good programmers always write down large collections of examples—called tests—to make sure their programs are behaving as they expect.

For our purposes, we are writing examples as part of the process of making sure we understand the problem. It's always a good idea to make sure you understand the question before you start writing code to solve a problem. Examples are a nice intermediate point: you can sketch out the relevant computation on concrete values first, then worry about turning it into a function. If you can't write the examples, chances are you won't be able to write the function either. Examples break down the programming process into smaller, manageable steps.

3.3.5 Functions Practice: Cost of pens Let's create one more function, this time for a more complicated example. Imagine that you are trying to compute the total cost of an order of pens with slogans (or messages) printed on them. Each pen costs 25 cents plus an additional 2 cents per character in the message (we'll count spaces between words as characters).

Following our steps to create a function once again, let's start by writing two concrete expressions that do this computation.

```
# ordering 3 pens that say "wow"
3 * (0.25 + (string-length("wow") * 0.02))

# ordering 10 pens that say "smile"
10 * (0.25 + (string-length("smile") * 0.02))
```

These examples introduce a new built-in function called `string-length`. It takes a string as input and produces the number of characters (including spaces and punctuation) in the string. These examples also show an example of working with numbers other than integers. Pyret requires a number before the decimal point, so if the “whole number” part is zero, you need to write 0 before the decimal. Also observe that Pyret uses a decimal point; it doesn't support conventions such as “0,02”.

The second step to writing a function was to identify which information differs across our two examples. In this case, we have two: the number of pens and the message to put on the pens. This means our function will have two parameters rather than just one.

```
fun pen-cost(num-pens :: Number, message :: String) -> Number:
    num-pens * (0.25 + (string-length(message) * 0.02))
end
```

Of course, as things get too long, it may be helpful to use multiple lines:

```

fun pen-cost(num-pens :: Number, message :: String)
    -> Number:
    num-pens * (0.25 + (string-length(message) * 0.02))
end

```

If you want to write a multi-line docstring, you need to use ````` rather than `"` to begin and end it, like so:

```

fun pen-cost(num-pens :: Number, message :: String)
    -> Number:
    doc: ```total cost for pens, each 25 cents
        plus 2 cents per message character```
    num-pens * (0.25 + (string-length(message) * 0.02))
end

```

We should also document the examples that we used when creating the function:

```

fun pen-cost(num-pens :: Number, message :: String)
    -> Number:
    doc: ```total cost for pens, each 25 cents
        plus 2 cents per message character```
    num-pens * (0.25 + (string-length(message) * 0.02))
where:
    pen-cost(3, "wow")
        is 3 * (0.25 + (string-length("wow") * 0.02))
    pen-cost(10, "smile")
        is 10 * (0.25 + (string-length("smile") * 0.02))
end

```

When writing `where` examples, we also want to include special yet valid cases that the function might have to handle, such as an empty message.

```
pen-cost(5, "") is 5 * 0.25
```

Note that our empty-message example has a simpler expression on the right side of `is`. The expression for what the function returns doesn't have to match the body expression; it simply has to evaluate to the same value as you expect the example to produce. Sometimes, we'll find it easier to just write the expected value directly. For the case of someone ordering no pens, for example, we'd include:

```
pen-cost(0, "bears") is 0
```

The point of the examples is to document how a function behaves on a variety of inputs. What goes to the right of the `is` should summarize the computation or the answer in some meaningful way. Most important? Do not write the function, run it to determine the answer, then put that answer on the right side of the `is`! Why not? Because the examples are meant to give some redundancy to the design process, so that you catch errors you might have made. If your function body is incorrect, and you use the function to generate the example, you won't get the benefit of using the example to check for errors.

We'll keep returning to this idea of writing good examples. Don't worry if you still have questions for now. Also, for the time being, we won't worry about nonsensical situations like negative numbers of pens. We'll get to those after we've learned additional coding techniques that will help us handle such situations properly.

Do Now!

We could have combined our two special cases into one example, such as

```
pen-cost(0, "") is 0
```

Does doing this seem like a good idea? Why or why not?

3.3.6 Recap: Defining Functions This chapter has introduced the idea of a function. Functions play a key role in programming: they let us configure computations with different concrete values at different times. The first time we compute the cost of pens, we might be asking about 10 pens that say `"Welcome"`. The next time, we might be

asking about 100 pens that say "Go Bears!". The core computation is the same in both cases, so we want to write it out once, configuring it with different concrete values each time we use it.

We've covered several specific ideas about functions:

- We showed the `fun` notation for writing functions. You learned that a function has a name (that we can use to refer to it), one or more parameters (names for the values we want to configure), as well as a body, which is the computation that we want to perform once we have concrete values for the parameters.
- We showed that we should include examples with our functions, to illustrate what the function computes on various specific values. Examples go in a `where` block within the function.
- We showed that we can use a function by providing concrete values to configure its parameters. To do this, we write the name of the function we want to use, followed by a pair of parenthesis around comma-separated values for the parameters. For example, writing the following expression (at the interactions prompt) will compute the cost of a specific order of pens:

```
pen-cost(10, "Welcome")
```

- We discussed that if we define a function in the definitions pane then press Run, Pyret will make an entry in the directory with the name of the function. If we later use the function, Pyret will look up the code that goes with that name, substitute the concrete values we provided for the parameters, and return the result of evaluating the resulting expression. Pyret will NOT produce anything in the interactions pane for a function definition (other than a report about whether the examples hold).

There's much more to learn about functions, including different reasons for creating them. We'll get to those in due course.

contents ← prev up next →

3.4 Conditionals and Booleans

3.4 Conditionals and Booleans

3.4.1 Motivating Example: Shipping Costs In Functions Practice: Cost of pens, we wrote a program (`pen-cost`) to compute the cost of ordering pens. Continuing the example, we now want to account for shipping costs. We'll determine shipping charges based on the cost of the order.

Specifically, we will write a function `add-shipping` to compute the total cost of an order including shipping. Assume an order valued at \$10 or less ships for \$4, while an order valued above \$10 ships for \$8. As usual, we will start by writing examples of the `add-shipping` computation.

Do Now!

Use the `is` notation from `where` blocks to write several examples of `add-shipping`. How are you choosing which inputs to use in your examples? Are you picking random inputs? Being strategic in some way? If so, what's your strategy?

Here is a proposed collection of examples for `add-shipping`.

```
add-shipping(10) is 10 + 4
add-shipping(3.95) is 3.95 + 4
add-shipping(20) is 20 + 8
add-shipping(10.01) is 10.01 + 8
```

Do Now!

What do you notice about our examples? What strategies do you observe across our choices?

Our proposed examples feature several strategic decisions:

- Including 10, which is at the boundary of charges based on the text
- Including 10.01, which is just over the boundary
- Including both natural and real (decimal) numbers
- Including examples that should result in each shipping charge mentioned in the problem (4 and 8)

So far, we have used a simple rule for creating a function body from examples: locate the parts that are changing, replace them with names, then make the names the parameters to the function.

Do Now!

What is changing across our `add-shipping` examples? Do you notice anything different about these changes compared to the examples for our previous functions?

Two things are new in this set of examples:

- The values of 4 and 8 differ across the examples, but they each occur in multiple examples.
- The values of 4 and 8 appear only in the computed answers—not as an input. Which one we use seems to depend on the input value.

These two observations suggest that something new is going on with `add-shipping`. In particular, we have clusters of examples that share a fixed value (the shipping charge), but different clusters (a) use different values and (b) have a pattern to their inputs (whether the input value is less than or equal to 10). This calls for being able to ask questions about inputs within our programs.

3.4.2 Conditionals: Computations with Decisions To ask a question about our inputs, we use a new kind of expression called an if expression. Here's the full definition of `add-shipping`:

```
fun add-shipping(order-amt :: Number) -> Number:  
  doc: "add shipping costs to order total"  
  if order-amt <= 10:  
    order-amt + 4  
  else:  
    order-amt + 8  
  end  
where:  
  add-shipping(10) is 10 + 4  
  add-shipping(3.95) is 3.95 + 4  
  add-shipping(20) is 20 + 8  
  add-shipping(10.01) is 10.01 + 8  
end
```

In an `if` expression, we ask a question that can produce an answer that is true or false (here `order-amt <= 10`, which we'll explain below in Booleans), provide one expression for when the answer to the question is true (`order-amt + 4`), and another for when the result is false (`order-amt + 8`). The `else` in the program marks the answer in the false case; we call this the `else` clause. We also need `end` to tell Pyret we're done with the question and answers.

3.4.3 Booleans Every expression in Pyret evaluates in a value. So far, we have seen three types of values: `Number`, `String`, and `Image`. What type of value does a question like `order-amt <= 10` produce? We can use the interactions prompt to experiment and find out.

Do Now!

Enter each of the following expressions at the interactions prompt. What type of value did you get? Do the values fit the types we have seen so far?

```
3.95 <= 10  
20 <= 10
```

The values `true` and `false` belong to a new type in Pyret, called `Boolean`. Named for George Boole. While there are an infinitely many values of type `Number`, there are only two of type `Boolean`: `true` and `false`.

Exercise

What would happen if we entered `order-amt <= 10` at the interactions prompt to explore booleans? Why does that happen?

3.4.3.1 Other Boolean Operations There are many other built-in operations that return `Boolean` values. Comparing values for equality is a common one: There is much more we can and should say about equality, which we will do later [Re-Examining Equality].

```
1 == 1  
true  
1 == 2  
false  
"cat" == "dog"  
false  
"cat" == "CAT"  
false
```

In general, `==` checks whether two values are equal. Note this is different from the single `=` used to associate names with values in the directory.

The last example is the most interesting: it illustrates that strings are case-sensitive, meaning individual letters must match in their case for strings to be considered equal. This will become relevant when we get to tables later.

Sometimes, we also want to compare strings to determine their alphabetical order. Here are several examples:

```
"a" < "b"  
true  
"a" >= "c"  
false  
"that" < "this"  
true  
"alpha" < "beta"  
true
```

which is the alphabetical order we're used to; but others need some explaining:

```
"a" >= "C"  
true  
"a" >= "A"  
true
```

These use a convention laid down a long time ago in a system called ASCII. Things get far more complicated with non-ASCII letters: e.g., Pyret thinks "Ł" is > than "Z", but in Polish, this should be `false`. Worse, the ordering depends on location (e.g., Denmark/Norway vs. Finland/Sweden).

Do Now!

Can you compare `true` and `false`? Try comparing them for equality (`==`), then for inequality (such as `<`).

In general, you can compare any two values for equality (well, almost, we'll come back to this later); for instance:

```
"a" == 1  
false
```

If you want to compare values of a specific kind, you can use more specific operators:

```
num-equal(1, 1)  
true  
num-equal(1, 2)  
false  
string-equal("a", "a")  
true  
string-equal("a", "b")  
false
```

Why use these operators instead of the more generic `==`?

Do Now!

Try

```
num-equal("a", 1)  
string-equal("a", 1)
```

Therefore, it's wise to use the type-specific operators where you're expecting the two arguments to be of the same type. Then, Pyret will signal an error if you go wrong, instead of blindly returning an answer (`false`) which lets your program continue to compute a nonsensical value.

There are even more Boolean-producing operators, such as:

```
wm = "will.i.am"  
string-contains(wm, "will")  
true
```

Note the capital W.

```
string-contains(wm, "Will")  
false
```

In fact, just about every kind of data will have some Boolean-valued operators to enable comparisons.

3.4.3.2 Combining Booleans Often, we want to base decisions on more than one Boolean value. For instance, you are allowed to vote if you're a citizen of a country and you are above a certain age. You're allowed to board a bus if you have a ticket or the bus is having a free-ride day. We can even combine conditions: you're allowed to drive if you are above a certain age and have good eyesight and—either pass a test or have a temporary license. Also, you're allowed to drive if you are not inebriated.

Corresponding to these forms of combinations, Pyret offers three main operations: `and`, `or`, and `not`. Here are some examples of their use:

```
(1 < 2) and (2 < 3)  
true  
(1 < 2) and (3 < 2)  
false  
(1 < 2) or (2 < 3)  
true  
(3 < 2) or (1 < 2)  
true  
not(1 < 2)  
false
```

Exercise

Explain why numbers and strings are not good ways to express the answer to a true/false question.

3.4.4 Asking Multiple Questions Shipping costs are rising, so we want to modify the `add-shipping` program to include a third shipping level: orders between \$10 and \$30 ship for \$8, but orders over \$30 ship for \$12. This calls for two modifications to our program:

- We have to be able to ask another question to distinguish situations in which the shipping charge is 8 from those in which the shipping charge is 12.
- The question for when the shipping charge is 8 will need to check whether the input is between two values.

We'll handle these in order.

The current body of `add-shipping` asks one question: `order-amt <= 10`. We need to add another one for `order-amt <= 30`, using a charge of 12 if that question fails. Where do we put that additional question?

An expanded version of the if-expression, using `else if`, allows you to ask multiple questions:

```

fun add-shipping(order-amt :: Number) -> Number:
  doc: "add shipping costs to order total"
  if order-amt <= 10:
    order-amt + 4
  else if order-amt <= 30:
    order-amt + 8
  else:
    order-amt + 12
  end
where:
...
end

```

At this point, you should also add `where` examples that use the 12 charge.

How does Pyret determine which answer to return? It evaluates each question expression in order, starting from the one that follows `if`. It continues through the questions, returning the value of the answer of the first question that returns true. Here's a summary of the `if`-expression syntax and how it evaluates.

```

if QUESTION1:
  <result in case first question true>
else if QUESTION2:
  <result in case QUESTION1 false and QUESTION2 true>
else:
  <result in case both QUESTIONS false>
end

```

A program can have multiple `else if` cases, thus accommodating an arbitrary number of questions within a program.

Do Now!

The problem description for `add-shipping` said that orders between 10 and 30 should incur an 8 charge. How does the above code capture “between”?

This is currently entirely implicit. It depends on us understanding the way an `if` evaluates. The first question is `order-amt <= 10`, so if we continue to the second question, it means `order-amt > 10`. In this context, the second question asks whether `order-amt <= 30`. That's how we're capturing “between”-ness.

Do Now!

How might you modify the above code to build the “between 10 and 30” requirement explicitly into the question for the 8 case?

Remember the `and` operator on booleans? We can use that to capture “between” relationships, as follows:

`(order-amt > 10) and (order-amt <= 30)`

Do Now!

Why are there parentheses around the two comparisons? If you replace `order-amt` with a concrete value (such as 20) and leave off the parenthesis, what happens when you evaluate this expression in the interactions pane?

Here is what `add-shipping` look like with the `and` included:

```

fun add-shipping(order-amt :: Number) -> Number:
  doc: "add shipping costs to order total"
  if order-amt <= 10:
    order-amt + 4
  else if (order-amt > 10) and (order-amt <= 30):
    order-amt + 8
  else:
    order-amt + 12
  end

```

where:

```
add-shipping(10) is 10 + 4
add-shipping(3.95) is 3.95 + 4
add-shipping(20) is 20 + 8
add-shipping(10.01) is 10.01 + 8
add-shipping(30) is 30 + 8
add-shipping(32) is 32 + 12
end
```

Both versions of `add-shipping` support the same examples. Are both correct? Yes. And while the first part of the second question (`order-amt > 10`) is redundant, it can be helpful to include such conditions for three reasons:

1. They signal to future readers (including ourselves!) the condition covering a case.
2. They ensure that if we make a mistake in writing an earlier question, we won't silently get surprising output.
3. They guard against future modifications, where someone might modify an earlier question without realizing the impact it's having on a later one.

Exercise

An online-advertising firm needs to determine whether to show an ad for a skateboarding park to website users. Write a function `show-ad` that takes the age and haircolor of an individual user and returns `true` if the user is between the ages of 9 and 18 and has either pink or purple hair.

Try writing this two ways: once with `if` expressions and once using just boolean operations.

Responsible Computing: Harms from Reducing People to Simple Data

Assumptions about users get encoded in even the simplest functions. The advertising exercise shows an example in which a decision gets made on the basis of two pieces of information about a person: age and haircolor. While some people might stereotypically associate skateboarders with being young and having colored hair, many skateboarders do not fit these criteria and many people who fit these criteria don't skateboard.

While real programs to match ads to users are more sophisticated than this simple function, even the most sophisticated advertising programs boil down to tracking features or information about individuals and comparing it to information about the content of ads. A real ad system would differ in tracking dozens (or more) of features and using more advanced programming ideas than simple conditionals to determine the suitability of an ad (we'll discuss some of these later in the book). This example also extends to situations far more serious than ads: who gets hired, granted a bank loan, or sent to or released from jail are other examples of real systems that depend on comparing data about individuals with criteria maintained by a program.

From a social responsibility perspective, the questions here are what data about individuals should be used to represent them for processing by programs and what stereotypes might those data encode. In some cases, individuals can be represented by data without harm (a university housing office, for example, stores student ID numbers and which room a student is living in). But in other cases, data about individuals get interpreted in order to predict something about them. Decisions based on those predictions can be inaccurate and hence harmful.

3.4.5 Evaluating by Reducing Expressions In How Functions Evaluate, we talked about how Pyret reduces expressions and function calls to values. Let's revisit this process, this time expanding to consider if-expressions. Suppose we want to compute the wages of a worker. The worker is paid \$10 for every hour up to the first 40 hours, and is paid \$15 for every extra hour. Let's say `hours` contains the number of hours they work, and suppose it's 45:

```
hours = 45
```

Suppose the formula for computing the wage is

```
if hours <= 40:
    hours * 10
```

```

else if hours > 40:
    (40 * 10) + ((hours - 40) * 15)
end

```

Let's now see how this results in an answer, using a step-by-step process that should match what you've seen in algebra classes (the steps are described in the margin notes to the right): The first step is to substitute the `hours` with 45.

```

if 45 <= 40:
    45 * 10
else if 45 > 40:
    (40 * 10) + ((45 - 40) * 15)
end

```

Next, the conditional part of the `if` expression is evaluated, which in this case is `false`.

```

=> if false:
    45 * 10
else if 45 > 40:
    (40 * 10) + ((45 - 40) * 15)
end

```

Since the condition is `false`, the next branch is tried.

```

=> if 45 > 40:
    (40 * 10) + ((45 - 40) * 15)
end

```

Pyret evaluates the question in the conditional, which in this case produces `true`.

```

=> if true:
    (40 * 10) + ((45 - 40) * 15)
end

```

Since the condition is `true`, the expression reduces to the body of that branch. After that, it's just arithmetic.

```

=> (40 * 10) + ((45 - 40) * 15)
=> 400 + (5 * 15)
=> 475

```

This style of reduction is the best way to think about the evaluation of Pyret expressions. The whole expression takes steps that simplify it, proceeding by simple rules. You can use this style yourself if you want to try and work through the evaluation of a Pyret program by hand (or in your head).

3.4.6 Composing Functions We started this chapter wanting to account for shipping costs on an order of pens. So far, we have written two functions:

- `pen-cost` for computing the cost of the pens
- `add-shipping` for adding shipping costs to a total amount

What if we now wanted to compute the price of an order of pens including shipping? We would have to use both of these functions together, sending the output of `pen-cost` to the input of `add-shipping`.

Do Now!

Write an expression that computes the total cost, with shipping, of an order of 10 pens that say "bravo".

There are two ways to structure this computation. We could pass the result of `pen-cost` directly to `add-shipping`:

```
add-shipping(pen-cost(10, "bravo"))
```

Alternatively, you might have named the result of `pen-cost` as an intermediate step:

```
pens = pen-cost(10, "bravo")
add-shipping(pens)
```

Both methods would produce the same answer.

3.4.6.1 How Function Compositions Evaluate Let's review how these programs evaluate in the context of substitution and the directory. We'll start with the second version, in which we explicitly name the result of calling `pen-cost`.

Evaluating the second version: At a high level, Pyret goes through the following steps:

- Substitute 10 for `num-pens` and "bravo" for `message` in the body of `pen-cost`, then evaluate the substituted body
- Store `pens` in the directory, with a value of 3.5
- As a first step in evaluating `add-shipping(pens)`, look up the value of `pens` in the directory
- Substitute 3.5 for `order-amt` in the body of `add-shipping` then evaluate the resulting expression, which results in 7.5

Evaluating the first version: As a reminder, the first version consisted of a single expression:

```
add-shipping(pen-cost(10, "bravo"))
```

- Since arguments are evaluated before functions get called, start by evaluating `pen-cost(10, "bravo")` (again using substitution), which reduces to 3.5
- Substitute 3.5 for `order-amt` in the body of `add-shipping` then evaluate the resulting expression, which results in 7.5

Do Now!

Contrast these two summaries. Where do they differ? What about the code led to those differences?

The difference lies in the use of the directory: the version that explicitly named `pens` uses the directory. The other version doesn't use the directory at all. Yet both approaches lead to the same result, since the same value (the result of calling `pen-cost`) gets substituted into the body of `add-shipping`.

This analysis might suggest that the version that uses the directory is somehow wasteful: it seems to take more steps just to end up at the same result. Yet one might argue that the version that uses the directory is easier to read (different readers will have different opinions on this, and that's fine). So which should we use?

Use whichever makes more sense to you on a given problem. There will be times when we prefer each of these styles. Furthermore, it will turn out (once we've learned more about nuances of how programs evaluate) that the two versions aren't as different as they appear right now.

3.4.6.2 Function Composition and the Directory Let's try one more variation on this problem. Perhaps seeing us name the intermediate result of `pen-cost` made you wish that we had used intermediate names to make the body of `pen-cost` more readable. For example, we could have written it as:

```
fun pen-cost(num-pens :: Number, message :: String)
  -> Number:
  doc: ```total cost for pens, each 25 cents
      plus 2 cents per message character```
  message-cost = (string-length(message) * 0.02)
  num-pens * (0.25 + message-cost)
where:
  ...
end
```

Do Now!

Write out the high level steps for how Pyret will evaluate the following program using this new version of `pen-cost`:

```
pens = pen-cost(10, "bravo")
add-shipping(pens)
```

Hopefully, you made two entries into the directory, one for `message-cost` inside the body of `pen-cost` and one for `pens` as we did earlier.

Do Now!

Consider the following program. What result do you think Pyret should produce?

```
pens = pen-cost(10, "bravo")
cheap-message = (message-cost > 0.5)
add-shipping(pens)
```

Using the directory you envisioned for the previous activity, what answer do you think you will get?

Something odd is happening here. The new program tries to use `message-cost` to define `cheap-message`. But the name `message-cost` doesn't appear anywhere in the program, unless we peek inside the function bodies. But letting code peek inside function bodies doesn't make sense: you might not be able to see inside the functions (if they are defined in libraries, for example), so this program should report an error that `message-cost` is undefined.

Okay, so that's what should happen. But our discussion of the directory suggests that both `pens` and `message-cost` will be in the directory, meaning Pyret would be able to use `message-cost`. What's going on?

This example prompts us to explain one more nuance about the directory. Precisely to avoid problems like the one illustrated here (which should produce an error), directory entries made within a function are local (private) to the function body. When you call a function, Pyret sets up a local directory that other functions can't see. A function body can add or refer to names in either its local, private directory (as with `message-cost`) or the overall (global) directory (as with `pens`). But in no case can one function call peek inside the local directory for another function call. Once a function call completes, its local directory disappears (because nothing else would be able to use it anyway).

3.4.7 Nested Conditionals We showed that the results in if-expressions are themselves expressions (such as `order-amt + 4` in the following function):

```
fun add-shipping(order-amt :: Number) -> Number:
  doc: "add shipping costs to order total"
  if order-amt <= 10:
    order-amt + 4
  else:
    order-amt + 8
  end
end
```

The result expressions can be more complicated. In fact, they could be entire if-expressions!. To see an example of this, let's develop another function. This time, we want a function that will compute the cost of movie tickets. Let's start with a simple version in which tickets are \$10 apiece.

```
fun buy-tickets1(count :: Number) -> Number:
  doc: "Compute the price of tickets at $10 each"
  count * 10
where:
  buy-tickets1(0) is 0
  buy-tickets1(2) is 2 * 10
  buy-tickets1(6) is 6 * 10
end
```

Now, let's augment the function with an extra parameter to indicate whether the purchaser is a senior citizen who is entitled to a discount. In such cases, we will reduce the overall price by 15%.

```

fun buy-tickets2(count :: Number, is-senior :: Boolean)
    -> Number:
    doc: ```Compute the price of tickets at $10 each with
        senior discount of 15%```
    if is-senior == true:
        count * 10 * 0.85
    else:
        count * 10
    end
where:
buy-tickets2(0, false) is 0
buy-tickets2(0, true) is 0
buy-tickets2(2, false) is 2 * 10
buy-tickets2(2, true) is 2 * 10 * 0.85
buy-tickets2(6, false) is 6 * 10
buy-tickets2(6, true) is 6 * 10 * 0.85
end

```

There are a couple of things to notice here:

- The function now has an additional parameter of type `Boolean` to indicate whether the purchaser is a senior citizen.
- We have added an `if` expression to check whether to apply the discount.
- We have more examples, because we have to vary both the number of tickets and whether a discount applies.

Now, let's extend the program once more, this time also offering the discount if the purchaser is not a senior but has bought more than 5 tickets. Where should we modify the code to do this? One option is to first check whether the senior discount applies. If not, we check whether the number of tickets qualifies for a discount:

```

fun buy-tickets3(count :: Number, is-senior :: Boolean)
    -> Number:
    doc: ```Compute the price of tickets at $10 each with
        discount of 15% for more than 5 tickets
        or being a senior```
    if is-senior == true:
        count * 10 * 0.85
    else:
        if count > 5:
            count * 10 * 0.85
        else:
            count * 10
    end
end
where:
buy-tickets3(0, false) is 0
buy-tickets3(0, true) is 0
buy-tickets3(2, false) is 2 * 10
buy-tickets3(2, true) is 2 * 10 * 0.85
buy-tickets3(6, false) is 6 * 10 * 0.85
buy-tickets3(6, true) is 6 * 10 * 0.85
end

```

Notice here that we have put a second `if` expression within the `else` case. This is valid code. (We could have also made an `else if` here, but we didn't so that we could show that nested conditionals are also valid).

Exercise

Show the steps through which this function would evaluate in a situation where no discount applies, such as `buy-tickets3(2, false)`.

Do Now!

Look at the current code: do you see a repeated computation that we might end up having to modify later?

Part of good code style is making sure that our programs would be easy to maintain later. If the theater changes its discount policy, for example, the current code would require us to change the discount (0.85) in two places. It would be much better to have that computation written only one time. We can achieve that by asking which conditions lead to the discount applying, and writing them as the check within just one `if` expression.

Do Now!

Under what conditions should the discount apply?

Here, we see that the discount applies if either the purchaser is a senior or more than 5 tickets have been bought. We can therefore simplify the code by using `or` as follows (we've left out the examples because they haven't changed from the previous version):

```
fun buy-tickets4(count :: Number, is-senior :: Boolean)
    -> Number:
    doc: ```Compute the price of tickets at $10 each with
        discount of 15% for more than 5 tickets
        or being a senior```
    if (is-senior == true) or (count > 5):
        count * 10 * 0.85
    else:
        count * 10
    end
end
```

This code is much tighter, and all of the cases where the discount applies are described together in one place. There are still two small changes we want to make to really clean this up though.

Do Now!

Take a look at the expression `is-senior == true`. What will this evaluate to when the value of `is-senior` is `true`? What will it evaluate to when the value of `is-senior` is `false`?

Notice that the `== true` part is redundant. Since `is-senior` is already a boolean, we can check its value without using the `==` operator. Here's the revised code:

```
fun buy-tickets5(count :: Number, is-senior :: Boolean)
    -> Number:
    doc: ```Compute the price of tickets at $10 each with
        discount of 15% for more than 5 tickets
        or being a senior```
    if is-senior or (count > 5):
        count * 10 * 0.85
    else:
        count * 10
    end
end
```

Notice the revised question in the `if` expression. As a general rule, your code should never include `== true`. You can always take that out and just use the expression you were comparing to `true`.

Do Now!

What do you write to eliminate `== false`? For example, what might you write instead of `is-senior == false`?

Finally, notice that we still have one repeated computation: the base cost of the tickets (`count * 10`): if the ticket price changes, it would be better to have only one place to update that price. We can clean that up by first computing the base price, then applying the discount when appropriate:

```

fun buy-tickets6(count :: Number, is-senior :: Boolean)
    -> Number:
    doc: ```Compute the price of tickets at $10 each with
        discount of 15% for more than 5 tickets
        or being a senior```
    base = count * 10
    if is-senior or (count > 5):
        base * 0.85
    else:
        base
    end
end

```

3.4.8 Recap: Booleans and Conditionals With this chapter, our computations can produce different results in different situations. We ask questions using if-expressions, in which each question or check uses an operator that produces a boolean.

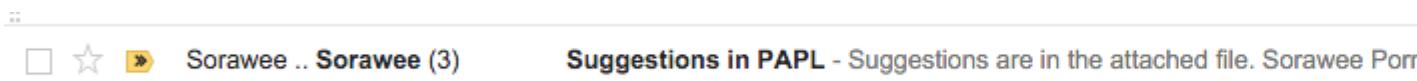
- There are two Boolean values: `true` and `false`.
- A simple kind of check (that produces a boolean) compares values for equality (`==`) or inequality(`<>`). Other operations that you know from math, like `<` and `>=`, also produce booleans.
- We can build larger expressions that produce booleans from smaller ones using the operators `and`, `or`, `not`.
- We can use `if` expressions to ask true/false questions within a computation, producing different results in each case.
- We can nest conditionals inside one another if needed.
- You never need to use `==` to compare a value to `true` or `false`: you can just write the value or expression on its own (perhaps with `not` to get the same computation).

contents ← prev up next →

4.1 Introduction to Tabular Data

4.1 Introduction to Tabular Data Many interesting data in computing are tabular—i.e., like a table—in form. First we'll see a few examples of them, before we try to identify what they have in common. Here are some of them:

- An email inbox is a list of messages. For each message, your inbox stores a bunch of information: its sender, the subject line, the conversation it's part of, the body, and quite a bit more.



A screenshot of an email inbox interface. At the top, there are icons for reply, forward, and delete. The subject line "Sorawee .. Sorawee (3)" is displayed. Below the subject, the text "Suggestions in PAPL - Suggestions are in the attached file. Sorawee Pom..." is visible. The inbox lists three messages:

Name	Time	Artist	Album	Genre	Rating
You Never Can Tell	3:31	Emmylou Harris	Luxury Liner	Country	
Imagine	2:54	Chet Atkins & Mar...	The Secret Police...	Rock	
The Wheel	4:21	Rosanne Cash	The Wheel	Country	

- A music playlist. For each song, your music player maintains a bunch of information: its name, the singer, its length, its genre, and so on.

Name	Time	Artist	Album	Genre	Rating
You Never Can Tell	3:31	Emmylou Harris	Luxury Liner	Country	
Imagine	2:54	Chet Atkins & Mar...	The Secret Police...	Rock	
The Wheel	4:21	Rosanne Cash	The Wheel	Country	

- A filesystem folder or directory. For each file, your filesystem records a name, a modification date, size, and other information.

Name	Date Modified	Size	Kind
Alloy4.2	Sep 25, 2012, 3:55 PM	4.6 MB	Application
Android File Transfer	Oct 15, 2012, 12:25 PM	6 MB	Application
App Store	Mar 24, 2016, 11:28 AM	2.8 MB	Application
Aquamacs	Nov 7, 2014, 10:36 AM	160 MB	Application
Automator	Mar 24, 2016, 11:28 AM	14.6 MB	Application

Do Now!

Can you come up with more examples?

How about:

- Responses to a party invitation.
- A gradebook.
- A calendar agenda.

You can think of many more in your life!

What do all these have in common? The characteristics of tabular data are:

- They contain information about zero or more items (i.e., individuals or artifacts) that share characteristics. Each item is stored in a row. Each column tracks one of the shared attributes across the rows. For example, each song or email message or file is a row. Each of their characteristics—the song title, the message subject, the filename—is a column. While some spreadsheets might swap the roles of rows and columns, we stick to this organization as it aligns with the design of data-science software libraries. This is an example of what Hadley Wickham calls tidy data.

- Each row has the same columns as the other rows, in the same order.
- A given column has the same type, but different columns can have different types. For instance, an email message has a sender's name, which is a string; a subject line, which is a string; a sent date, which is a date; whether it's been read, which is a Boolean; and so on.
- The rows might be in some particular order. For instance, the emails are ordered by which was most recently sent.

Exercise

Find the characteristics of tabular data in the other examples described above, as well as in the ones you described.

We will now learn how to program with tables and to think about decomposing tasks involving them. You can also look up the full Pyret documentation for table operations. The programs later in this chapter use a function-based notation for processing tables, which you can access via the following:

```
include shared-gdrive(
  "dcic-2021",
  "1wyQZj_L0qqV9Ekgr9au6RX2iqt2Ga8Ep")
```

Documentation on the function-based table operators is available on a separate page outside of the Pyret documentation.

4.1.1 Creating Tabular Data Pyret provides multiple easy ways of creating tabular data. The simplest is to define the datum in a program as follows:

```
table: name, age
  row: "Alicia", 30
  row: "Meihui", 40
  row: "Jamal", 25
end
```

That is, a `table` is followed by the names of the columns in their desired order, followed by a sequence of `rows`. Each row must contain as many data as the column declares, and in the same order.

Exercise

Change different parts of the above example—e.g., remove a necessary value from a row, add an extraneous one, remove a comma, add an extra comma, leave an extra comma at the end of a row—and see what errors you get.

Note that in a table, the order of columns matters: two tables that are otherwise identical but with different column orders are not considered equal.

```
check:
  table: name, age
    row: "Alicia", 30
    row: "Meihui", 40
    row: "Jamal", 25
  end
  is-not
  table: age, name
    row: 30, "Alicia"
    row: 40, "Meihui"
    row: 25, "Jamal"
  end
end
```

Observe that the example above uses `is-not`, i.e., the test passes, meaning that the tables are not equal.

The `check:` annotation here is a way of writing `is` assertions about expressions outside of the context of a function (and its `where` block). We'll learn more about `check` in From Examples to Tests.

Table expressions create table values. These can be stored in variables just like numbers, strings, and images:

```
people = table: name, age
  row: "Alicia", 30
  row: "Meihui", 40
  row: "Jamal", 25
end
```

We call these literal tables when we create them with `table`. Pyret provides other ways to get tabular data, too! In particular, you can import tabular data from a spreadsheet, so any mechanism that lets you create such a sheet can also be used. You might:

- create the sheet on your own,
- create a sheet collaboratively with friends,
- find data on the Web that you can import into a sheet,
- create a Google Form that you get others to fill out, and obtain a sheet out of their responses

and so on. Let your imagination run wild! Once the data are in Pyret, it doesn't matter where they came from.

With tables, we begin to explore data that contain other (smaller) pieces of data. We'll refer to such data as structured data. Structured data organize their inner data in a structured way (here, rows and columns). As with images, when we wrote code that reflected the structure of the final image, we will see that code that works with tables also follows the structure of the data.

4.1.2 Extracting Rows and Cell Values Given a table, we sometimes want to look up the value of a particular cell. We'll work with the following table showing the number of riders on a shuttle service over several months:

```
shuttle = table: month, riders
  row: "Jan", 1123
  row: "Feb", 1045
  row: "Mar", 1087
  row: "Apr", 999
end
```

Do Now!

If you put this table in the definitions pane and press Run, what will be in the Pyret directory once the interactions prompt appears? Would the column names be listed in the directory?

As a reminder, the directory contains only those names that we assign values to using the form `name = .` The directory here would contain `shuttle`, which would be bound to the table (yes, the entire table would be in the directory!). The column names would not have their own entries in the directory. If we did try to put a column name in the directory, which value would it map to? There is a different value in the column for every row. Names in the directory map to only one value.

Let's explore how to extract the value of a given cell (row and column) in the table. Concretely, assume we want to extract the number of riders in March (1087) so we can use it in another computation. How do we do that?

Pyret (and most other programming languages designed for data analysis) organizes tables as collections of rows with shared columns. Given that organization, we get to a specific cell by first isolating the row we are interested in, then retrieving the contents of the cell.

Pyret numbers the rows of a table from top to bottom starting at 0 (most programming languages use 0 as the first position in a piece of data, for reasons we will see later). So if we want to see the data for March, we need to isolate row 2. We write:

```
shuttle.row-n(2)
```

We use the period notation to dig into a piece of structured data. Here, we are saying "dig into the `shuttle` table, extracting row number 2" (which is really the third row since Pyret counts positions from 0).

If we run this expression at the prompt, we get

"month"	"Mar"	"riders"	1087
---------	-------	----------	------

This is a new type of data called a `Row`. When Pyret displays a `Row` value, it shows you the column names and the corresponding values within the row.

To extract the value of a specific column within a row, we write the row followed by the name of the column (as a string) in square brackets. Here are two equivalent ways of getting the value of the `riders` column from the row for March:

```
shuttle.row-n(2)["riders"]
march-row = shuttle.row-n(2)
march-row["riders"]
```

Do Now!

What names would be in the Pyret directory when using each of these approaches?

Once we have the cell value (here a `Number`), we can use it in any other computation, such as

```
shuttle.row-n(2)["riders"] >= 1000
```

(which checks whether there were at least 1000 riders in March).

Do Now!

What do you expect would happen if you forgot the quotation marks and instead wrote:

```
shuttle.row-n(2)[riders]
```

What would Pyret do and why?

4.1.3 Functions over Rows Now that we have the ability to isolate Rows from tables, we can write functions that ask questions about individual rows. We just saw an example of doing a computation over row data, when we checked whether the row for March had more than 1000 riders. What if we wanted to do this comparison for an arbitrary row of this table? Let's write a function! We'll call it `cleared-1K`.

Let's start with a function header and some examples:

```
fun cleared-1K(r :: Row) -> Boolean:
  doc: "determine whether given row has at least 1000 riders"
  ...
where:
  cleared-1K(shuttle.row-n(2)) is true
  cleared-1K(shuttle.row-n(3)) is false
end
```

This shows you what examples for `Row` functions look like, as well as how we use `Row` as an input type.

To fill in the body of the function, we extract the content of the "`riders`" cell and compare it to 1000:

```
fun cleared-1K(r :: Row) -> Boolean:
  doc: "determine whether given row has at least 1000 riders"
  r["riders"] >= 1000
where:
  cleared-1K(shuttle.row-n(2)) is true
  cleared-1K(shuttle.row-n(3)) is false
end
```

Do Now!

Looking at the examples, both of them share the `shuttle.row-n` portion. Would it have been better to instead make `cleared-1K` a function that takes just the row position as input, such as:

```
fun cleared-1K(row-pos :: Number) -> Boolean:  
    ...  
where:  
    cleared-1K(2) is true  
    cleared-1K(3) is false  
end
```

What are the benefits and limitations to doing this?

In general, the version that takes the `Row` input is more flexible because it can work with a row from any table that has a column named "`riders`". We might have another table with more columns of information or different data tables for different years. If we modify `cleared-1K` to only take the row position as input, that function will have to fix which table it works with. In contrast, our original version leaves the specific table (`shuttle`) outside the function, which leads to flexibility.

Exercise

Write a function `is-winter` that takes a `Row` with a "`month`" column as input and produces a `Boolean` indicating whether the month in that row is one of "`Jan`", "`Feb`", or "`Mar`".

Exercise

Write a function `low-winter` that takes in `Row` with both "`month`" and "`riders`" columns and produces a `Boolean` indicating whether the row is a winter row with fewer than 1050 riders.

Exercise

Practice with the program directory! Take a `Row` function and one of its `where` examples, and show how the program directory evolves as you evaluate the example.

4.1.4 Processing Rows So far, we have looked at extracting individual rows by their position in the table and computing over them. Extracting rows by position isn't always convenient: we might have hundreds or thousands of rows, and we might not know where the data we want even is in the table. We would much rather be able to write a small program that identifies the row (or rows!) that meets a specific criterion.

Pyret offers three different notations for processing tables: one uses functions, one uses methods, and one uses a SQL-like notation. This chapter uses the function-based notation. The SQL-like notation and the methods-based notation are shown in the Pyret Documentation. To use the function-based notation, you'll need to include the file specified in the main narrative.

The rest of this section assumes that you have loaded the functions notation for working with tables, using the following line in your Pyret file:

```
include shared-gdrive(  
    "dcic-2021",  
    "1wyQZj_L0qqV9Ekgr9au6RX2iqt2Ga8Ep")
```

4.1.4.1 Finding Rows Imagine that we wanted to write a program to locate a row that has fewer than 1000 riders from our `shuttle` table. With what we've studied so far, how might we try to write this? We could imagine using a conditional, like follows:

```
if shuttle.row-n(0)["riders"] < 1000:  
    shuttle.row-n(0)  
else if shuttle.row-n(1)["riders"] < 1000:  
    shuttle.row-n(1)  
else if shuttle.row-n(2)["riders"] < 1000:  
    shuttle.row-n(2)  
else if shuttle.row-n(3)["riders"] < 1000:  
    shuttle.row-n(3)
```

```
else: ... # not clear what to do here
end
```

Do Now!

What benefits and limitations do you see to this approach?

There are a couple of reasons why we might not care for this solution. First, if we have thousands of rows, this will be terribly painful to write. Second, there's a lot of repetition here (only the row positions are changing). Third, it isn't clear what to do if there aren't any matching rows. In addition, what happens if there are multiple rows that meet our criterion? In some cases, we might want to be able to identify all of the rows that meet a condition and use them for a subsequent computation (like seeing whether some months have more low-ridership days than others).

This conditional is, however, the spirit of what we want to do: go through the rows of the table one at a time, identifying those that match some criterion. We just don't want to be responsible for manually checking each row. Fortunately for us, Pyret knows how to do that. Pyret knows which rows are in a given table. Pyret can pull out those rows one position at a time and check a criterion about each one.

We just need to tell Pyret what criterion we want to use.

As before, we can express our criterion as a function that takes a `Row` and produces a `Boolean` (a Boolean because our criterion was used as the question part of an `if` expression in our code sketch). In this case, we want:

```
fun below-1K(r :: Row) -> Boolean:
  doc: "determine whether row has fewer than 1000 riders"
  r["riders"] < 1000
where:
  below-1K(shuttle.row-n(2)) is false
  below-1K(shuttle.row-n(3)) is true
end
```

Now, we just need a way to tell Pyret to use this criterion as it searches through the rows. We do this with a function called `filter-with` which takes two inputs: the table to process and the criterion to check on each row of the table:

```
filter-with(shuttle, below-1K)
```

Under the hood, `filter-with` works roughly like the `if` statement we outlined above: it takes each row one at a time and calls the given criterion function on it. But what does it do with the results?

If you run the above expression, you'll see that `filter-with` produces a table containing the matching row, not the row by itself. This behavior is handy if multiple rows match the criterion. For example, try:

```
filter-with(shuttle, is-winter)
```

(using the `is-winter` function from an exercise earlier in this chapter). Now we get a table with the three rows corresponding to winter months. If we want to be able to name this table for use in future computations, we can do so with our usual notation for naming values:

```
winter = filter-with(shuttle, is-winter)
```

4.1.4.2 Ordering Rows Let's ask a new question: which winter month had the fewest number of riders?. This question requires us to identify a specific row, namely, the winter row with the smallest value in the `"riders"` column.

Do Now!

Can we do this with `filter-with`? Why or why not?

Think back to the `if` expression that motivated `filter-with`: each row is evaluated independently of the others. Our current question, however, requires comparing across rows. That's a different operation, so we will need more than `filter-with`.

Tools for analyzing data (whether programming languages or spreadsheets) provide ways for users to sort rows of a table based on the values in a single column. That would help us here: we could sort the winter rows from smallest

to largest value in the "riders" column, then extract the "riders" value from the first row. First, let's sort the rows:

```
order-by(winter, "riders", true)
```

The `order-by` function takes three inputs: the table to sort (`winter`), the column to sort on ("riders"), and a Boolean to indicate whether we want to sort in increasing order. (Had the third argument been `false`, the rows would be sorted in decreasing order of the values in the named column.)

month	riders
"Feb"	1045
"Jan"	1123

In the sorted table, the row with the fewest riders is in the first position. Our original question asked us to lookup the month with the fewest riders. We did this earlier.

Do Now!

Write the code to extract the name of the winter month with the fewest riders.

Here are two ways to write that computation:

```
order-by(winter, "riders", true).row-n(0)["month"]  
sorted = order-by(winter, "riders", true)  
least-row = sorted.row-n(0)  
least-row["month"]
```

Do Now!

Which of these two ways do you prefer? Why?

Do Now!

How does each of these programs affect the program directory?

Note that this problem asked us to combine several actions that we've already seen on rows: we identify rows from within a table (`filter-with`), order the rows (`order-by`), extract a specific row (`row-n`), then extract a cell (with square brackets and a column name). This is typical of how we will operate on tables, combining multiple operations to compute a result (much as we did with programs that manipulate images).

4.1.4.3 Adding New Columns Sometimes, we want to create a new column whose value is based on those of existing columns. For instance, our table might reflect employee records, and have columns named `hourly-wage` and `hours-worked`, representing the corresponding quantities. We would now like to extend this table with a new column to reflect each employee's total wage. Assume we started with the following table:

```
employees =  
  table: name, hourly-wage, hours-worked  
  row: "Harley", 15, 40  
  row: "Obi", 20, 45  
  row: "Anjali", 18, 39  
  row: "Miyako", 18, 40  
end
```

The table we want to end up with is:

```
employees =  
  table: name, hourly-wage, hours-worked, total-wage
```

```

row: "Harley", 15,      40,      15 * 40
row: "Obi",     20,      45,      20 * 45
row: "Anjali",   18,      39,      18 * 39
row: "Miyako",   18,      40,      18 * 40
end

```

(with the expressions in the `total-wage` column computed to their numeric equivalents: we used the expressions here to illustrate what we are trying to do).

Previously, when we have had a computation that we performed multiple times, we created a helper function to do the computation.

Do Now!

Propose a helper function for computing total wages given the hourly wage and number of hours worked.

Perhaps you came up with something like:

```

fun compute-wages(wage :: Number, hours :: Number) -> Number:
    wage * hours
end

```

which we could use as follows:

```

employees =
  table: name, hourly-wage, hours-worked, total-wage
  row: "Harley", 15, 40, compute-wages(15, 40)
  row: "Obi",     20, 45, compute-wages(20, 45)
  row: "Anjali",   18, 39, compute-wages(18, 39)
  row: "Miyako",   18, 40, compute-wages(18, 40)
end

```

This is the right idea, but we can actually have this function do a bit more work for us. The `wage` and `hours` values are in cells within the same row. So if we could instead get the current row as an input, we could write:

```

fun compute-wages(r :: Row) -> Number:
    r["hourly-wage"] * r["hours-worked"]
end

employees =
  table: name, hourly-wage, hours-worked, total-wage
  row: "Harley", 15, 40, compute-wages(<row0>)
  row: "Obi",     20, 45, compute-wages(<row1>)
  row: "Anjali",   18, 39, compute-wages(<row2>)
  row: "Miyako",   18, 40, compute-wages(<row3>)
end

```

But now, we are writing calls to `compute-wages` over and over! Adding computed columns is a sufficiently common operation that Pyret provides a table function called `build-column` for this purpose. We use it by providing the function to use to populate values in the new column as an input:

```

fun compute-wages(r :: Row) -> Number:
  doc: "compute total wages based on wage and hours worked"
  r["hourly-wage"] * r["hours-worked"]
end

build-column(employees, "total-wage", compute-wages)

```

This creates a new column, `total-wage`, whose value in each row is the product of the two named columns in that row. Pyret will put the new column at the right end.

4.1.4.4 Calculating New Column Values Sometimes, we just want to calculate new values for an existing column, rather than create an entirely new column. Giving raises to employees is one such example. Assume we wanted to give a 10% raise to all employees making less than 20 an hour. We could write:

```
fun new-rate(rate :: Number) -> Number:
    doc: "Raise rates under 20 by 10%"
    if rate < 20:
        rate * 1.1
    else:
        rate
    end
where:
    new-rate(20) is 20
    new-rate(10) is 11
    new-rate(0) is 0
end

fun give-raises(t :: Table) -> Table:
    doc: "Give a 10% raise to anyone making under 20"
    transform-column(t, "hourly-wage", new-rate)
end
```

Here, `transform-column` takes a table, the name of an existing column in the table, and a function to update the value. The updating function takes the current value in the column as input and produces the new value for the column as output.

Do Now!

Run `give-raises` on the `employees` table. What wage will show for "Miyako" in the `employees` table after `give-raises` completes. Why?

Like all other Pyret Table operations, `transform-column` produces a new table, leaving the original intact. Editing the original table could be problematic—what if you made a mistake? How would you recover the original table in that case? In general, producing new tables with any modifications, then creating a new name for the updated table once you have the one you want, is a less error-prone way of working with datasets.

4.1.5 Examples for Table-Producing Functions How do we write examples for functions that produce tables? Conceptually, the answer is simply "make sure you got the output table that you expected". Logistically, writing examples for table functions seems more painful because writing out an expected output tables is more work than simply writing the output of a function that produces numbers or strings. What can we do to manage that complexity?

Do Now!

How might you write the `where` block for `give-raises`?

Here are some ideas for writing the examples practically:

- Simplify the input table. Rather than work with a large table with all of the columns you have, create a small table that has sufficient variety only in the columns that the function uses. For our example, we might use:

```
wages-test =
    table: hourly-wage
    row: 15
    row: 20
    row: 18
    row: 18
end
```

Do Now!

Would any table with a column of numbers work here? Or are there some constraints on the rows or columns of the table?

The only constraint is that your input table has to have the column names used in your function.

- Remember that you can write computations in the code to construct tables. This saves you from doing calculations by hand.

where:

```
give-raises(wages-test) is
  table: hourly-wage
    row: 15 * 1.1
    row: 20
    row: 18 * 1.1
    row: 18 * 1.1
  end
```

This example shows that you can write an output table directly in the **where:** block – the table doesn't need to be named outside the function.

- Create a new table by taking rows from an existing table. If you were instead writing examples for a function that involves filtering out rows of a table, it helps to know how to create a new table using rows of an existing one. For example, if we were writing a function to find all rows in which employees were working exactly 40 hours, we'd like to make sure that the resulting table had the first and fourth rows of the `employees` table. Rather than write a new `table` expression to create that table, we could write it as follows:

```
emps-at-40 =
  add-row(
    add-row(employees.empty(),
      employees.row-n(0)),
    employees.row-n(3))
```

Here, `employees.empty()` creates a new, empty table with the same column headers as `employees`. We've already seen how `row-n` extracts a row from a table. The `add-row` function places the given row at the end of the given table.

Another tip to keep in mind: when the only thing your function does is call a built-in function like `transform-column` it usually suffices to write examples for the function you wrote to compute the new column value. It is only when your code is combining table operations, or doing more complex processing than a single call to a built-in table operation that you really need to present your own examples to a reader of your code.

contents ← prev up next →

4.2 Processing Tables

4.2 Processing Tables In data analysis, we often work with large datasets, some of which were collected by someone else. Datasets don't necessarily come in a form that we can work with. We might need the raw data pulled apart or condensed to coarser granularity. Some data might be missing or entered incorrectly. On top of that, we have to plan for long-term maintenance of our datasets or analysis programs. Finally, we typically want to use visualizations to either communicate our data or to check for issues with our data.

As a concrete example, assume that you are doing data analysis and support for a company that manages ticket sales for events. People purchase tickets through an online form. The form software creates a spreadsheet with all the entered data, which is what you have to work with. Here's a screenshot of a sample spreadsheet:

	A	B	C	D	E
1	Name	Email	Num Tickets	Discount Code	Delivery
2	Josie Zhao	jo@mail.com	2	BIRTHDAY	email
3	Sam Ochibe	s@sweb.com	1		pickup
4	Bart Simple	bart@simpson.org	5	STUDENT	yes
5	Ernie O'Malley	ernie.mail.com	0	none	email
6	Alvina Velasquez	alvie@schooledu	3	student	email
7	Zander	zandaman	10		email
8	Shweta Chowpatti	snc@this.org	three		pickup

Do Now!

Take a look at the table. What do you notice that might affect using the data in an analysis?
Or for the operations for managing an event?

Some issues jump out quickly: the `three` in the "Num Tickets" column, differences in capitalization in the "Discount Code" column, and the use of each of "none" and blank spaces in the the "Discount Code" column (you may have spotted additional issues). Before we do any analysis with this dataset, we need to clean it up so that our analysis will be reliable. In addition, sometimes our dataset is clean, but it needs to be adjusted or prepared to fit the questions we want to ask. This chapter looks at both steps, and the programming techniques that are helpful for them.

4.2.1 Cleaning Data Tables

4.2.1.1 Loading Data Tables If you want to load a csv file, first import it into a Google Sheet, then load it from the Google Sheet into Pyret. The first step to working with an outside data source is to load it into your programming and analysis environment. In Pyret, we do this using the `load-table` command, which loads tables from Google Sheets.

```
include gdrive-sheets

ssid = "1DKngiBfI2cGTVEazFEyXf7H4mh18IU5yv2TfZWv6Rc8"
event-data =
  load-table: name, email, tickcount, discount, delivery
    source: load-spreadsheet(ssid).sheet-by-name("Orig Data", true)
  end
```

In this example:

- `ssid` is the identifier of the Google Sheet we want to load (the identifier is the long sequence of letters and numbers in the Google Sheet URL).
- `load-table` says to create a Pyret table via loading. The sequence of names following `load-table` is used for the column headers in the Pyret version of the table. These do NOT have to match the names used in the Sheets version of the table.
- `source` tells Pyret which sheet to load. The `load-spreadsheet` operation takes the Google Sheet identifier (here, `ssid`), as well as the name of the individual worksheet (or tab) as named within the Google Sheet (here, "`Orig Data`"). The final boolean indicates whether there is a header row in the table (`true` means there is a header row).

When we try to run this code, Pyret complains about the `three` in the Num Tickets column: it was expecting a number, but instead found a string. Pyret expects all columns to hold values of the same type. When loading a table from file, Pyret bases the type of each column on the corresponding value in the first row of the table.

This is an example of a data error that we have to fix in the source file, rather than by using programs within Pyret. Not all languages will reject programs on loading. Languages embody philosophies of what programmers should expect from them. Some will try to make whatever the programmer provided work, while others will ask the programmer to fix issues upfront. Pyret tends more towards the latter philosophy, while relaxing it in some places (such as making types optional). Within the source Google Sheet for this chapter, there is a separate worksheet/tab named "`Data`" in which the `three` has been replaced with a number. If we use "`Data`" instead of "`Orig Data`" in the above `load-spreadsheet` command, the event table loads into Pyret.

Exercise

Why might we have created a separate worksheet with the corrected data, rather than just correct the original sheet?

4.2.1.2 Dealing with Missing Entries When we create tables manually in Pyret, we have to provide a value for each cell – there's no way to "skip" a cell. When we create tables in a spreadsheet program (such as Excel, Google Sheets, or something similar), it is possible to leave cells completely empty. What happens when we load a table with empty cells into Pyret?

```
event-data =
  load-table: name, email, tickcount, discount, delivery
  source: load-spreadsheet(ssid).sheet-by-name("Data", true)
end
```

The original data file has a blank in the `discount` column. If we load the table and look at how Pyret reads it in, we find something new in that column:

name	email	tickcount	discount	delivery
"Josie Zhao"	"jo@mail.com"	2	some("BIRTHDAY")	"email"
"Sam Ochibe"	"s@swb.com"	1	none	"pickup"
"Bart Simple"	"bart@simpson.org"	5	some("STUDENT")	"yes"
"Ernie O'Malley"	"ernie.mail.com"	0	some("none")	"email"
"Alvina Velasquez"	"alvie@schooledu"	3	some("student")	"email"
"Zander"	"zandaman"	10	none	"email"
"Shweta Chowpatti"	"snc@this.org"	3	some(" ")	"pickup"

Note that those cells that had discount codes in them now have an odd-looking notation like `some("student")`, while the cells that were empty contain `none`, but `none` isn't a string. What's going on?

Pyret supports a special type of data called option. As the name suggests, option is for data that may or may not be present. `none` is the value that stands for "the data are missing". If a datum are present, it appears wrapped in `some`.

Do Now!

Look at the `discount` value for Ernie's row: it reads `some("none")`. What does this mean?
How is this different from `none` (as in Sam's row)?

In Pyret, the right way to address this is to indicate how to handle missing values for each column, so that the data are as you expect after you read them in. We do this with an additional aspect of `load-table` called sanitizers. Here's how we modify the code:

```
include data-source # to get the sanitizers

event-data =
  load-table: name, email, tickcount, discount, delivery
  source: load-spreadsheet(ssid).sheet-by-name("Data", true)
  sanitize name using string-sanitizer
  sanitize email using string-sanitizer
  sanitize tickcount using num-sanitizer
  sanitize discount using string-sanitizer
  sanitize delivery using string-sanitizer
end
```

Each of the `sanitize` lines tells Pyret what to do in the case of missing data in the respective column. `string-sanitizer` says to load missing data as an empty string (""). `num-sanitizer` says to load missing data as zero (0). The sanitizers also handle simple data conversions. If the `string-sanitizer` were applied to a column with a number (like 3), the sanitizer would convert that number to a string (like "3"). Using the sanitizers, the `event-data` table reads in as follows:

name	email	tickcount	discount	delivery
"Josie Zhao"	"jo@mail.com"	2	"BIRTHDAY"	"email"
"Sam Ochibe"	"s@sweb.com"	1	" "	"pickup"
"Bart Simple"	"bart@simpson.org"	5	"STUDENT"	"yes"
"Ernie O'Malley"	"ernie.mail.com"	0	"none"	"email"
"Alvina Velasquez"	"alvie@schooledu"	3	"student"	"email"
"Zander"	"zandaman"	10	" "	"email"
"Shweta Chowpatti"	"snc@this.org"	3	" "	"pickup"

Wait – wouldn't putting types on the columns (like `discount :: String`) in the `load-table` also solve this problem? No, because the type isn't enough to know which value should be the default! In some situations, you might want the default value to be something other than an empty string or 0. Sanitizers actually let you tailor this for yourself (a sanitizer is just a Pyret function: see the Pyret documentation for details on sanitizer inputs).

Rule of thumb: when you load a table, use a sanitizer to guard against errors in case the original sheet is missing data in some cells.

4.2.1.3 Normalizing Data Next, let's look at the "Discount Code" column. Our goal is to be able to accurately answer the question "How many orders were placing under each discount code". We would like to have the answer summarized in a table, where one column names the discount code and another gives a count of the rows that used that code.

Do Now!

Examples first! What table do we want from this computation on the fragment of table that we gave you?

You can't answer this question without making some decisions about how to standardize the names and how to handle missing values. The term normalization refers to making sure that a collection of data (such as a column) shares structure and formatting. Our solution will aim to produce the following table, but you could have made different choices from what we have here:

discount-code	num-orders
"BIRTHDAY"	1
"STUDENT"	2
"none"	4

How do we get to this table? How do we figure this out if we aren't sure?

Start by looking in the tables documentation for any library functions that might help with this task. In the case of Pyret, we find:

```
# count(tab :: Table, colname :: String) -> Table
# Produces a table that summarizes how many rows have
# each value in the named column.
```

This sounds useful, as long as every column has a value in the "Discount code" column, and that the only values in the column are those in our desired output table. What do we need to do to achieve this?

- Get "none" to appear in every cell that currently lacks a value
- Convert all the codes that aren't "none" to upper case

Fortunately, these tasks align with functions we've already seen how to use: each one is an example of a column transformation, where the second one involves the upper-case conversion functions from the `String` library.

We can capture these together in a function that takes in and produces a string:

```
fun cell-to-discount-code(str :: String) -> String:  
  doc: ````uppercase all strings other than none,  
        convert blank cells to contain none````  
  if (str == "") or (str == "none"):  
    "none"  
  else:  
    string-to-upper(str)  
  end  
where:  
  cell-to-discount-code("") is "none"  
  cell-to-discount-code("none") is "none"  
  cell-to-discount-code("birthday") is "BIRTHDAY"  
  cell-to-discount-code("Birthday") is "BIRTHDAY"  
end
```

Do Now!

Assess the examples included with `cell-to-discount-code`. Is this a good set of examples, or are any key ones missing?

The current examples consider different capitalizations for "birthday", but not for "none". Unless you are confident that the data-gathering process can't produce different capitalizations of "none", we should include that as well:

```
cell-to-discount-code("NoNe") is "none"
```

Oops! If we add this example to our `where` block and run the code, Pyret reports that this example fails.

Do Now!

Why did the "NoNe" case fail?

Since we check for the string "none" in the `if` expression, we need to normalize the input to match what our `if` expression expects. Here's the modified code, on which all the examples pass.

```
fun cell-to-discount-code(str :: String) -> String:  
  doc: ````uppercase all strings other than none,  
        convert blank cells to contain none````  
  if (str == "") or (string-to-lower(str) == "none"):  
    "none"  
  else:  
    string-to-upper(str)  
  end  
where:  
  cell-to-discount-code("") is "none"  
  cell-to-discount-code("none") is "none"  
  cell-to-discount-code("NoNe") is "none"  
  cell-to-discount-code("birthday") is "BIRTHDAY"  
  cell-to-discount-code("Birthday") is "BIRTHDAY"  
end
```

Using this function with `transform-column` yields a table with a standardized formatting for discount codes (reminder that you need to be working with the function operators for tables for this to work):

```

include shared-gdrive(
  "dcic-2021",
  "1wyQZj_L0qqV9Ekgr9au6RX2iqt2Ga8Ep")

discount-fixed =
  transform-column(event-data, "discount", cell-to-discount-code)

```

Exercise

Try it yourself: normalize the "delivery" column so that all "yes" values are converted to "email".

Now that we've cleaned up the codes, we can proceed to using the "count" function to extract our summary table:

```
count(discount-fixed, "discount")
```

This produces the following table:

value	count
" "	1
"STUDENT"	2
"none"	3
"BIRTHDAY"	1

Do Now!

What's with that first row, with the discount code " "? Where might that have come from?

Maybe you didn't notice this before (or wouldn't have noticed it within a larger table), but there must have been a cell of the source data with a string of blanks, rather than missing content. How do we approach normalization to avoid missing cases like this?

4.2.1.4 Normalization, Systematically As the previous example showed, we need a way to think through potential normalizations systematically. Our initial discussion of writing examples gives an idea of how to do this. One of the guidelines there says to think about the domain of the inputs, and ways that inputs might vary. If we apply that in the context of loaded datasets, we should think about how the original data were collected.

Do Now!

Based on what you know about websites, where might the event code contents come from? How might they have been entered? What do these tell you about different plausible mistakes in the data?

In this case, for data that came from a web-based form (as we revealed at the beginning), the data was likely entered in one of two ways:

- via a drop-down menu
- in a text-entry box

A drop-down menu automatically normalizes the data, so that's not a plausible source (this is why you should use drop-downs on forms when you want users to select from a fixed collection of options). So let's assume this is from a text-entry box.

A text-entry box means that any sort of typical human typing error could show up in your data: swapped letters, missing letters, leading spaces, capitalization, etc. You could also get data where someone just typed the wrong thing (or something random, just to see what your form would do).

Do Now!

Which of swapped letters, missing errors, and random text do you think a program can correct for automatically?

Swapped and missing letters are the sorts of things a spell-checker might be able to fix (especially if the program knew all of the valid discount codes). Random junk, by definition, is random. There, you'd have to talk to the events company to decide how they wanted those handled (convert them to "`none`", reach out to the customer, etc. – these are questions of policy, not of programming).

But really, the moral of this is to just use drop-downs or other means to prevent incorrect data at the source whenever possible.

As you get more experience with programming, you will also learn to anticipate certain kinds of errors. Issues such as cells that appear empty will become second nature once you've processed enough tables that have them, for example. Needing to anticipate data errors is one reason why good data scientists have to understand the domain that they are working in.

The takeaway from this is how we talked through what to expect. We thought about where the data came from, and what errors would be plausible in that situation. Having a clear error model in mind will help you develop more robust programs. In fact, such adversarial thinking is a core skill of working in security, but now we're getting ahead of ourselves.

Exercise

In spreadsheets, cells that appear empty sometimes have actual content, in the form of strings made up of spaces: both "" and " " appear the same when we look at a spreadsheet, but they are actually different values computationally.

How would you modify `cell-to-discount-code` so that strings containing only spaces were also converted to "`none`"? (Hint: look for `string-replace` in the strings library.)

4.2.1.4.1 Using Programs to Detect Data Errors Sometimes, we also look for errors by writing functions to check whether a table contains unexpected values. Let's consider the "`email`" column: that's a place where we should be able to write a program to flag any rows with invalid email addresses. What makes for a valid email address? Let's consider two rules:

- Valid email addresses should contain an @ sign
- Valid email addresses should end in one of ".com", ".edu" or ".org"

This is admittedly an outdated, limited, and US-centric definition of email addresses, but expanding the formats does not fundamentally change the point of this section.

Exercise

Write a function `is-email` that takes a string and returns a boolean indicating whether the string satisfies the above two rules for being valid email addresses. For a bit more of a challenge, also include a rule that there must be some character between the @ and the .-based ending.

Assuming we had such a function, a routine `filter-with` could then produce a table identifying all rows that need to have their email addresses corrected. The point here is that programs are often helpful for finding data that need correcting, even if a program can't be written to perform the fixing.

4.2.2 Task Plans Before we move on, it's worth stepping back to reflect on our process for producing the discount-summary table. We started from a concrete example, checked the documentation for a built-in function that might help, then manipulated our data to work with that function. These are part of a more general process that applies to data and problems beyond tables. We'll refer to this process as task planning. Specifically, a task plan is a sequence of steps (tasks) that decompose a computational problem into smaller steps (sub-tasks). A useful task plan contains sub-tasks that you know how to implement, either by using a built-in function or writing your

own. There is no single notation or format for task plans. For some problems, a bulleted-list of steps will suffice. For others, a diagram showing how data transform through a problem is more helpful. This is a personal choice tailored to a specific problem. The goal is simply to decompose a problem into something of a programming to-do list, to help you manage the process.

Strategy: Creating a Task Plan

1. Develop a concrete example showing the desired output on a given input (you pick the input: a good one is large enough to show different features of your inputs, but small enough to work with manually during planning. For table problems, roughly 4-6 rows usually works well in practice).
2. Mentally identify functions that you already know (or that you find in the documentation) that might be useful for transforming the input data to the output data.
3. Develop a sequence of steps—whether as pictures, textual descriptions of computations, or a combination of the two—that could be used to solve the problem. If you are using pictures, draw out the intermediate data values from your concrete example and make notes on what operations might be useful to get from one intermediate value to the next. The functions you identified in the previous step should show up here.
4. Repeat the previous step, breaking down the subtasks until you believe you could write expressions or functions to perform each step or data transformation.

Here's a diagram-based task plan for the `discount-summary` program that we just developed. We've drawn this on paper to highlight that task plans are not written within a programming environment.

Name	tickcount	discount
"Jaime"	2	"BIRTHDAY"
"Daren"	4	"birthday"
"Nya"	3	" "
"Luis"	5	"Student"

using transform-column

name	tickcount	discount
"Jaime"	2	"BIRTHDAY"
"Daren"	4	"BIRTHDAY"
"Nya"	3	"none"
"Luis"	5	"STUDENT"

using built-in count

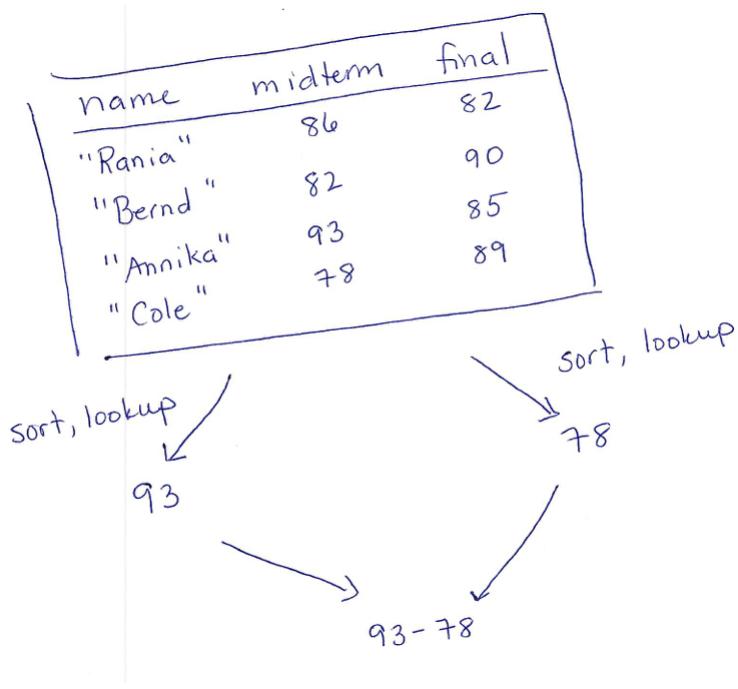
value	count
"BIRTHDAY"	2
"none"	1
"STUDENT"	1

Once you have a plan, you turn it into a program by writing expressions and functions for the intermediate steps, passing the output of one step as the input of the next. Sometimes, we look at a problem and immediately know how to write the code for it (if it is a kind of problem that you've solved many times before). When you don't immediately see the solution, use this process and break down the problem by working with concrete examples of data.

Exercise

You've been asked to develop a program that identifies the student with the largest improvement from the midterm to the final exam in a course. Your input table will have columns for each exam as well as for student names. Write a task plan for this problem.

Some task plans involve more than just a sequence of table values. Sometimes, we do multiple transformations to the same table to extract different pieces of data, then compute over those data. In that case, we draw our plan with branches that show the different computations that come together in the final result. Continuing with the gradebook, for example, you might be asked to write a program to compute the difference between the largest and lowest scores on the midterm. That task plan might look like:



Exercise

You've been given a table of weather data that has columns for the date, amount of precipitation, and highest temperature for the day. You've been asked to compute whether there were more snowy days in January than in February, where a day is snowy if the highest temperature is below freezing and the precipitation was more than zero.

The takeaway of this strategy is easy to state:

If you aren't sure how to approach a problem, don't start by trying to write code. Plan until you understand the problem.

Newer programmers often ignore this advice, assuming that the fastest way to produce working code for a programming problem is to start writing code (especially if you see classmates who are able to jump directly to writing code). Experienced programmers know that trying to write all the code before you've understood the problem will take much longer than stepping back and understanding the problem first. As you develop your programming skills, the specific format of your task plans will evolve (and indeed, we will see some cases of this later in the book as well). But the core idea is the same: use concrete examples to help identify the intermediate computations that will need, then convert those intermediate computations to code after or as you figure them out.

4.2.3 Preparing Data Tables Sometimes, the data we have is clean (in that we've normalized the data and dealt with errors), but it still isn't in a format that we can use for the analysis that we want to run. For example, what if we want to look at the distribution of small, medium, and large ticket orders? In our current table, we have the number of tickets in an order, but not an explicit label on the scale of that order. If we wanted to produce some sort of chart showing our order scales, we will need to make those labels explicit.

4.2.3.1 Creating bins The act of reducing one set of values (such as the `tickcounts` values) into a smaller set of categories (such as small/medium/large for orders, or morning/afternoon/etc. for timestamps) is known as binning. The bins are the categories. To put rows into bins, we create a function to compute the bin for a raw data value, then create a column for the new bin labels.

Here's an example of creating bins for the scale of the ticket orders:

```
fun order-scale-label(r :: Row) -> String:
    doc: "categorize the number of tickets as small, medium, large"
    numtickets = r["tickcount"]
```

```

if numtickets >= 10: "large"
else if numtickets >= 5: "medium"
else: "small"
end
end

order-bin-data =
build-column(cleaned-event-data, "order-scale", order-scale-label)

```

4.2.3.2 Splitting Columns The events table currently uses a single string to represent the name of a person. This single string is not useful if we want to sort data by last names, however. Splitting one column into several columns can be a useful step in preparing a dataset for analysis or use. Programming languages usually provide a variety of operations for splitting apart strings: Pyret has operations called `string-split` and `string-split-all` that split one string into several around a given character (like a space). You could, for example, write `string-split("Josie Zhao", " ")` to extract "Josie" and "Zhao" as separate strings.

Exercise

Write a task plan (not the code, just the plan) for a function that would replace the current `name` column in the events table with two columns called `last-name` and `first-name`.

Do Now!

Write down a collection of specific name strings on which you would want to test a name-splitting function.

Hopefully, you at least looked at the table and noticed that we have one individual, "Zander" whose entire name is a single string, rather than having both a first name and a last name. How would we handle middle names? Or names from cultures where a person's name has the last names of both of their parents as part of their name? Or cultures that put the family name before the given name? Or cultures where names are not written as in the Latin alphabet. This is definitely getting more complicated.

Responsible Computing: Representing Names

Representing names as data is heavily context- and culture-dependent. Think carefully about the individuals your dataset needs to include and design your table structure accordingly. It's okay to have a table structure that excludes names outside of the population you are trying to represent. The headache comes from realizing later that your dataset or program excludes data that need to be supported. In short, examine your table structure for assumptions it makes about your data and choose table structure after thinking about which observations or individuals it needs to represent.

For a deeper look at the complexity of representing real-world names and dates in programs, search for "falsehoods programmers believe about ...", which turns up articles such as Falsehoods Programmers Believe About Names and Falsehoods Programmers Believe About Time.

Exercise

Write a program that filters a table to only include rows in which the name is not comprised of two strings separated by a space.

Exercise

Write a program that takes a table with a `name` column in "`first-name last-name`" format and replaces the `name` column with two columns called `last-name` and `first-name`. To extract the first- and last-names from a single name string, use:

```

string-split(name-string, " ").get(0) # get first name
string-split(name-string, " ").get(1) # get last name

```

4.2.4 Managing and Naming Data Tables At this point, we have worked with several versions of the events table:

- The original dataset that we tried to load
- The new sheet of the dataset with manual corrections
- The version with the discount codes normalized
- Another version that normalized the delivery mode
- The version extended with the order-scale column

Which of these versions should get explicit names within our code file?

Usually, we keep both the original raw source datasheet, as well as the copy with our manual corrections. Why? In case we ever have to look at the original data again, either to identify kinds of errors that people were making or to apply different fixes.

For similar reasons, we want to keep the cleaned (normalized) data separate from the version that we initially loaded. Fortunately, Pyret helps with this since it creates new tables, rather than modify the prior ones. If we have to normalize multiple columns, however, do we really need a new name for every intermediate table?

As a general rule, we usually maintain separate names for the initially-loaded table, the cleaned table, and for significant variations for analysis purposes. In our code, this might mean having names:

```
event-data = ... # the loaded table

cleaned-event-data =
  transform-column(
    transform-column(event-data, "discount", cell-to-discount-code),
    "delivery", yes-to-email)

order-bin-data =
  build-column(
    cleaned-event-data, "order-scale", order-scale-label)
```

where `yes-to-email` is a function we have not written, but that might have normalized the "yes" value in the "delivery" column. Note that we applied each of the normalizations in sequence, naming only the final table with all normalizations applied. In professional practice, if you were working with a very large dataset, you might just write the cleaned dataset out to a file, so that you loaded only the clean version during analysis. We will look at writing to file later. Having only a few table names will reduce your own confusion when working with your files. If you work on multiple data-analyses, developing a consistent strategy for how you name your tables will likely help you better manage your code as you switch between projects.

4.2.5 Visualizations and Plots Now that our data are cleaned and prepared, we are ready to analyze it. What might we want to know? Perhaps we want to know which discount code has been used most often. Maybe we want to know whether the time when a purchase was made correlates with how many tickets people buy. There's a host of different kinds of visualizations and plots that people use to summarize data.

Which plot type to use depends on both the question and the data at hand. The nature of variables in a dataset helps determine relevant plots or statistical operations. An attribute or variable in a dataset (i.e., a single column of a table) can be classified as one of several different kinds, including:

- quantitative: a variable whose values are numeric and can be ordered with a consistent interval between values. They are meaningful to use in computations.
- categorical: a variable with a fixed set of values. The values may have an order, but there are no meaningful computational operations between the values other than ordering. Such variables usually correspond to characteristics of your samples.

Do Now!

Which kind of variable are last names? Grades in courses? Zipcodes?

Common plots and the kinds of variables they require include:

- Scatterplots show relationships between two quantitative variables, with one variable on each axis of a 2D chart.
- Frequency Bar charts show the frequency of each categorical value within a column of a dataset.
- Histograms segment quantitative data into equal-size intervals, showing the distribution of values across each interval.
- Pie charts show the proportion of cells in a column across the categorical values in a dataset.

Do Now!

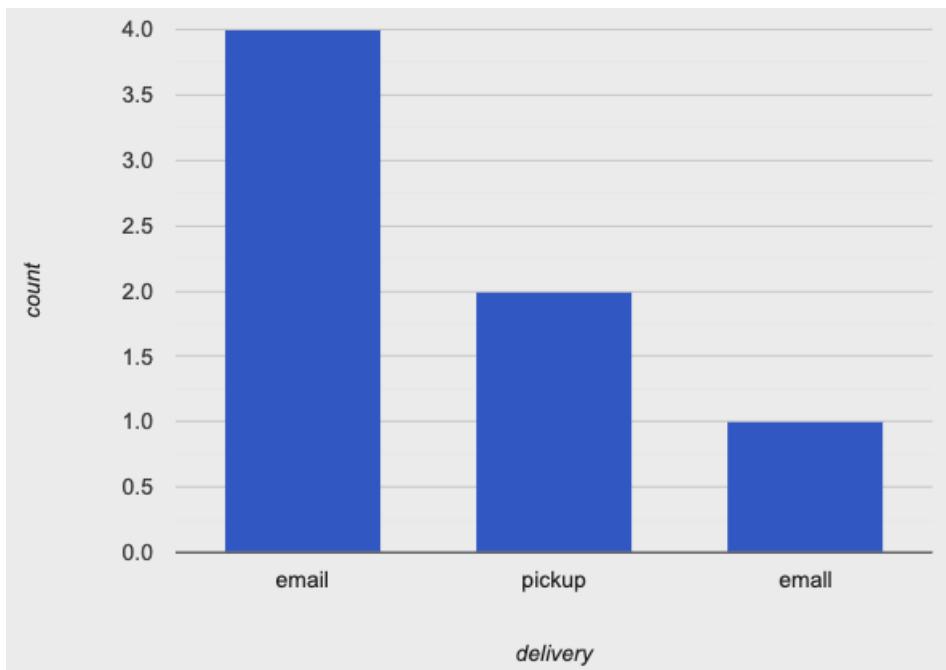
Map each of the following questions to a chart type, based on the kinds of variables involved in the question:

- Which discount code has been used most often?
- Is there a relationship between the number of tickets purchased in one order and the time of purchase?
- How many orders have been made for each delivery option?

For example, we might use a frequency-bar-chart to answer the third question. Based on the `Table` documentation, we would generate this using the following code (with similar style for the other kinds of plots):

```
freq-bar-chart(cleaned-event-data, "delivery")
```

Which yields the following chart (assuming we had not actually normalized the contents of the "delivery" column):



Whoa – where did that extra "email" column come from? If you look closely, you'll spot the error: in the row for "Alvina", there's a typo ("emall" with an l instead of an i) in the discount column (drop-down menus, anyone?).

The lesson here is that plots and visualizations are valuable not only in the analysis phase, but also early on, when we are trying to sanity check that our data are clean and ready to use. Good data scientists never trust a dataset without first making sure that the values make sense. In larger datasets, manually inspecting all of the data is often infeasible. But creating some plots or other summaries of the data is also useful for identifying errors.

4.2.6 Summary: Managing a Data Analysis This chapter has given you a high-level overview of how to use coding for managing and processing data. When doing any data analysis, a good data practitioner undergoes several steps:

1. Think about the data in each column: what are plausible values in the column, and what kinds of errors might be in that column based on what you know about the data collection methods?
2. Check the data for errors, using a combination of manual inspection of the table, plots, and `filter-with` expressions that check for unexpected values. Normalize or correct the data, either at the source (if you control that) or via small programs.
3. Store the normalized/cleaned data table, either as a name in your program, or by saving it back out to a new file. Leave the raw data intact (in case you need to refer to the original later).
4. Prepare the data based on the questions you want to ask about it: compute new columns, bin existing columns, or combine data from across tables. You can either finish all preparations and name the final table, or you can make separate preparations for each question, naming the per-question tables.
5. At last, perform your analysis, using the statistical methods, visualizations, and interpretations that make sense for the question and kinds of variables involved. When you report out on the data, always store notes about the file that holds your analysis code, and which parts of the file were used to generate each graph or interpretation in your report.

There's a lot more to managing data and performing analysis than this book can cover. There are entire books, degrees, and careers in each of the management of data and its analysis. One area we have not discussed, for example, is machine learning, in which programs (that others have written) are used to make predictions from datasets (in contrast, this chapter has focused on projects in which you will use summary statistics and visualizations to perform analysis). These skills covered in this chapter are all prerequisites for using machine learning effectively and responsibly. But we still have much more to explore and understand about data themselves, which we turn to in the coming chapters. Onward!

Responsible Computing: Bias in Statistical Prediction

In a book that is discussing data and social responsibility, we would be remiss in not at least mentioning some of the many issues that arise when using data to make predictions (via techniques like machine learning). Some issues arise from problems with the data themselves (e.g., whether samples are representative, or whether correlations between variables lead to discrimination as in algorithmic hiring). Others arise with how data collected for one purpose is misused to make predictions for another. Still more arise with the interpretation of results.

These are all rich topics. There are myriad articles which you could read at this point to begin to understand the pitfalls (and benefits) of algorithmic decision making. This book will focus instead on issues that arise from the programs we are teaching you to write, leaving other courses, or the interests of instructors, to augment the material as appropriate for readers' contexts.

contents ← prev up next →

5.1 From Tables to Lists

5.1 From Tables to Lists Previously [Introduction to Tabular Data] we began to process collective data in the form of tables. Though we saw several powerful operations that let us quickly and easily ask sophisticated questions about our data, they all had two things in common. First, all were operations by rows. None of the operations asked questions about an entire column at a time. Second, all the operations not only consumed but also produced tables. However, we already know [Getting Started] there are many other kinds of data, and sometimes we will want to compute one of them. We will now see how to achieve both of these things, introducing an important new type of data in the process.

5.1.1 Basic Statistical Questions There are many more questions we might want to ask of our events data. For instance:

- The most-frequently used discount code.
- The average number of tickets per order.
- The largest ticket order.
- The most common number of tickets in an order.
- The collection of unique discount codes that were used (many might have been available).
- The collection of distinct email addresses associated with orders, so we can contact customers (some customers may have placed multiple orders).
- Which school lead to the largest number of orders with a "STUDENT" discount.

Notice the kinds of operations that we are talking about: computing the maximum, minimum, average, median, and other basic statistics. Pyret has several built-in statistics functions in the math and statistics packages.

Do Now!

Think about whether and how you would express these questions with the operations you have already seen.

In each of these cases, we need to perform a computation on a single column of data (even in the last question about the "STUDENT" discount, as we would filter the table to those rows, then do a computation over the `email` column). In order to capture these in code, we need to extract a column from the table.

For the rest of this chapter, we will work with a cleaned copy of the `event-data` from the previous chapter. The cleaned data, which applies the transformations at the end of the previous chapter, is in a different tab of the same Google Sheet as the other versions of the event data.

```
ssid = "1DKngiBfI2cGTVEazFEyXf7H4mh18IU5yv2TfZWv6Rc8"
cleaned-data =
  load-table: name, email, tickcount, discount, delivery
    source: load-spreadsheet(ssid).sheet-by-name("Cleaned", true)
    sanitize name using string-sanitizer
    sanitize email using string-sanitizer
    sanitize tickcount using num-sanitizer
    sanitize discount using string-sanitizer
    sanitize delivery using string-sanitizer
end
```

5.1.2 Extracting a Column from a Table Our collection of table functions includes one that we haven't yet used, called `select-columns`. As the name suggests, this function produces a new table containing only certain columns from an existing table. Let's extract the `tickcount` column so we can compute some statistics over it. We use the following expression:

```
select-columns(cleaned-data, [list: "tickcount"])
```

tickcount
2
1
5
0
3
10
3

This focuses our attention on the numeric ticket sales, but we're still stuck with a column in a table, and none of the other tables functions let us do the kinds of computations we might want over these numbers. Ideally, we want the collection of numbers on their own, without being wrapped up in the extra layer of table cells.

In principle, we could have a collection of operations on a single column. In some languages that focus solely on tables, such as SQL, this is what you'll find. However, in Pyret we have many more kinds of data than just columns (as we'll soon see [Introduction to Structured Data], we can even create our own!), so it makes sense to leave the gentle cocoon of tables sooner or later. An extracted column is a more basic kind of datum called a list, which can be used to represent a sequence of data outside of a table.

Just as we have used the notation `.row-n` to pull a single row from a table, we use a similar dot-based notion to pull out a single column. Here's how we extract the `tickcount` column:

```
cleaned-data.get-column("tickcount")
```

In response, Pyret produces the following value:

```
[list: 2, 1, 5, 0, 3, 10, 3]
```

Now, we seem to have only the values that were in the cells in the column, without the enclosing table. Yet the numbers are still bundled up, this time in the `[list: ...]` notation. What is that?

5.1.3 Understanding Lists A list has much in common with a single-column table:

- The elements have an order, so it makes sense to talk about the “first”, “second”, “last”—and so on—element of a list.
- All elements of a list are expected to have the same type.

The crucial difference is that a list does not have a “column name”; it is anonymous. That is, by itself a list does not describe what it represents; this interpretation is done by our program.

5.1.3.1 Lists as Anonymous Data This might sound rather abstract—and it is—but this isn't actually a new idea in our programming experience. Consider a value like `3` or `-1`: what is it? It's the same sort of thing: an anonymous value that does not describe what it represents; the interpretation is done by our program. In one setting `3` may represent an age, in another a play count; in one setting `-1` may be a temperature, in another the average of

several temperatures. Similarly with a string: Is "project" a noun (an activity that one or more people perform) or a verb (as when we display something on a screen)? Likewise with images and so on. In fact, tables have been the exception so far in having description built into the data rather than being provided by a program!

This genericity is both a virtue and a problem. Because, like other anonymous data, a list does not provide any interpretation of its use, if we are not careful we can accidentally mis-interpret the values. On the other hand, it means we can use the same datum in several different contexts, and one operation can be used in many settings.

Indeed, if we look at the list of questions we asked earlier, we see that there are several common operations—maximum, minimum, average, and so on—that can be asked of a list of values without regard for what the list represents (heights, ages, playcounts). In fact, some are specific to numbers (like average) while some (like maximum) can be asked of any type on which we can perform a comparison (like strings).

5.1.3.2 Creating Literal Lists We have already seen how we can create lists from a table, using `get-column`. As you might expect, however, we can also create lists directly:

```
[list: 1, 2, 3]
[list: -1, 5, 2.3, 10]
[list: "a", "b", "c"]
[list: "This", "is", "a", "list", "of", "words"]
```

Of course, lists are values so we can name them using variables—

```
shopping-list = [list: "muesli", "fiddleheads"]
```

—pass them to functions (as we will soon see), and so on.

Do Now!

Based on these examples, can you figure out how to create an empty list?

As you might have guessed, it's `[list:]` (the space isn't necessary, but it's a useful visual reminder of the void).

5.1.4 Operating on Lists

5.1.4.1 Built-In Operations on Lists of Numbers Pyret handily provides a useful set of operations we can already perform on lists. The lists documentation describes these operations. As you might have guessed, we can already compute most of the answers we've asked for at the start of the chapter. First we need to include some libraries that contain useful functions:

```
import math as M
import statistics as S
```

We can then access several useful functions:

```
tickcounts = cleaned-data.get-column("tickcount")

M.max(tickcounts)      # largest number in a list
M.sum(tickcounts)      # sum of numbers in a list
S.mean(tickcounts)     # mean (average) of numbers in a list
S.median(tickcounts)   # median of numbers in a list
```

The `M.` notation means "the function inside the library `M`". The `import` statement in the above code gave the name `M` to the `math` library.

5.1.4.2 Built-In Operations on Lists in General Some of the useful computations in our list at the start of the chapter involved the `discount` column, which contains strings rather than numbers. Specifically, let's consider the following question:

- Compute the collection of unique discount codes that were used (many might have been available).

None of the table functions handle a question like this. However, this is a common kind of question to ask about a collection of values (How many unique artists are in your playlist? How many unique faculty are teaching courses?).

As such, Pyret (as most languages) provides a way to identify the unique elements of a list. Here's how we get the list of all discount codes that were used in our table:

```
import lists as L
codes = cleaned-data.get-column("discount")
L.distinct(codes)
```

The `distinct` function produces a list of the unique values from the input list: every value in the input list appears exactly once in the output list. For the above code, Pyret produces:

```
[list: "BIRTHDAY", "STUDENT", "none"]
```

What if we wanted to exclude "`none`" from that list? After all, "`none`" isn't an actual discount code, but rather one that we introduced while cleaning up the table. Is there a way to easily remove "`none`" from the list?

There are two ways we could do it. In the Pyret lists documentation, we find a function called `remove`, which removes a specific element from a list:

```
L.remove(L.distinct(codes), "none")
```

```
[list: "BIRTHDAY", "STUDENT"]
```

But this operation should also sound familiar: with tables, we used `filter-with` to keep only those elements that meet a specific criterion. The filtering idea is so common that Pyret (and most other languages) provide a similar operation on lists. In the case of the discount codes, we could also have written:

```
fun real-code(c :: String) -> Boolean:
  not(c == "none")
end
L.filter(real-code, L.distinct(codes))
```

The difference between these two approaches is that `filter` is more flexible: we can check any characteristic of a list element using `filter`, but `remove` only checks whether the entire element is equal to the value that we provide. If instead of removing the specific string "`none`", we had wanted to remove all strings that were in all-lowercase, we would have needed to use `filter`.

Exercise

Write a function that takes a list of words and removes those words in which all letters are in lowercase. (Hint: combine `string-to-lower` and `==`).

5.1.4.3 An Aside on Naming Conventions Our use of the plural `codes` for the list of values in the column named `discount` (singular) is deliberate. A list contains multiple values, so a plural is appropriate. In a table, in contrast, we think of a column header as naming a single value that appears in a specific row. Often, we speak of looking up a value in a specific row and column: the singular name for the column supports thinking about lookup in an individual row.

5.1.4.4 Getting Elements By Position Let's look at a new analysis question: the events company recently ran an advertising campaign on `web.com`, and they are curious whether it paid off. To do this, they need to determine how many sales were made by people with `web.com` email addresses.

Do Now!

Propose a task plan (Task Plans) for this computation.

Here's a proposed plan, annotated with how we might implement each part:

1. Get the list of email addresses (use `get-column`)
2. Extract those that came from `web.com` (use `L.filter`)
3. Count how many email addresses remain (using `L.length`, which we hadn't discussed yet, but it is in the documentation)

(As a reminder, unless you immediately see how to solve a problem, write out a task plan and annotate the parts you know how to do. It helps break down a programming problem into more manageable parts.)

Let's discuss the second task: identifying messages from `web.com`. We know that email addresses are strings, so if we could determine whether an email string ends in `@web.com`, we'd be set. You could consider doing this by looking at the last 7 characters of the email string. Another option is to use a string operation that we haven't yet seen called `string-split-all`, which splits a string into a list of substrings around a given character. For example:

```
string-split-all("this-has-hyphens", "-")
[list: "this", "has", "hyphens"]

string-split("bonnie@pyret.org", "@")
[list: "bonnie", "pyret.org"]
```

This seems pretty useful. If we split each email string around the `@` sign, then we can check whether the second string in the list is `web.com` (since email addresses should have only one `@` sign). But how would we get the second element out of the list produced by `string-split-all`? Here we dig into the list, as we did to extract rows from tables, this time using the `get` operation.

```
string-split("bonnie@pyret.org", "@").get(1)
"pyret.org"
```

Do Now!

Why do we use `1` as the input to `get` if we want the second item in the list?

Here's the complete program for doing this check:

```
fun web-com-address(email :: String) -> Boolean:
    doc: "determine whether email is from web.com"
    string-split(email, "@").get(1) == "web.com"
where:
    web-com-address("bonnie@pyret.org") is false
    web-com-address("parrot@web.com") is true
end

emails = cleaned-data.get-column("email")
L.length(L.filter(web-com-address, emails))
```

Exercise

What happens if there is a malformed email address string that doesn't contain the `@` string?

What would happen? What could you do about that?

5.1.4.5 Transforming Lists Imagine now that we had a list of email addresses, but instead just wanted a list of usernames. This doesn't make sense for our event data, but it does make sense in other contexts (such as connecting messages to folders organized by students' usernames).

Specifically, we want to start with a list of addresses such as:

```
[list: "parrot@web.com", "bonnie@pyret.org"]
```

and convert it to

```
[list: "parrot", "bonnie"]
```

Do Now!

Consider the list functions we have seen so far (`distinct`, `filter`, `length`) – are any of them useful for this task? Can you articulate why?

One way to articulate a precise answer to this is think in terms of the inputs and outputs of the existing functions. Both `filter` and `distinct` return a list of elements from the input list, not transformed elements. `length` returns a number, not a list. So none of these are appropriate.

This idea of transforming elements is similar to the `transform-column` operation that we previously saw on tables. The corresponding operation on lists is called `map`. Here's an example:

```
fun extract-username(email :: String) -> String:  
  doc: "extract the portion of an email address before the @ sign"  
  string-split(email, "@").get(0)  
where:  
  extract-username("bonnie@pyret.org") is "bonnie"  
  extract-username("parrot@web.com") is "parrot"  
end  
  
L.map(extract-username,  
      [list: "parrot@web.com", "bonnie@pyret.org"])
```

5.1.4.6 Recap: Summary of List Operations At this point, we have seen several useful built-in functions for working with lists:

- `filter` :: $(A \rightarrow \text{Boolean})$, $\text{List} < A > \rightarrow \text{List} < A >$, which produces a list of elements from the input list on which the given function returns `true`.
- `map` :: $(A \rightarrow B)$, $\text{List} < A > \rightarrow \text{List} < B >$, which produces a list of the results of calling the given function on each element of the input list.
- `distinct` :: $\text{List} < A > \rightarrow \text{List} < A >$, which produces a list of the unique elements that appear in the input list.
- `length` :: $\text{List} < A > \rightarrow \text{Number}$, which produces the number of elements in the input list.

Here, a type such as `List < A >` says that we have a list whose elements are of some (unspecified) type which we'll call `A`. A type variable such as this is useful when we want to show relationships between two types in a function contract. Here, the type variable `A` captures that the type of elements is the same in the input and output to `filter`. In `map`, however, the type of element in the output list could differ from that in the input list.

One additional built-in function that is quite useful in practice is:

- `member` :: $\text{List} < A >$, $\text{Any} \rightarrow \text{Boolean}$, which determines whether the given element is in the list. We use the type `Any` when there are no constraints on the type of value provided to a function.

Many useful computations can be performed by combining these operations.

Exercise

Assume you used a list of strings to represent the ingredients in a recipe. Here are three examples:

```
stir-fry =  
  [list: "peppers", "pork", "onions", "rice"]  
dosa = [list: "rice", "lentils", "potato"]  
misir-wot =  
  [list: "lentils", "berbere", "tomato"]
```

Write the following functions on ingredient lists:

- `recipes-uses`, which takes an ingredient list and an ingredient and determines whether the recipe uses the ingredient.
- `make-vegetarian`, which takes an ingredient list and replaces all meat ingredients with "tofu". Meat ingredients are "pork", "chicken", and "beef".
- `protein-veg-count`, which takes an ingredient list and determines how many ingredients are in the list that aren't "rice" or "noodles".

Exercise

More challenging: Write a function that takes two ingredient lists and returns all of the ingredients that are common to both lists.

Exercise

Another more challenging: Write a function that takes an ingredient and a list of ingredient lists and produces a list of all the lists that contain the given ingredient.

Hint: write examples first to make sense of the problem as needed.

5.1.5 Lambda: Anonymous Functions Let's revisit the program we wrote earlier in this chapter for finding all of the discount codes that were used in the events table:

```
fun real-code(c :: String) -> Boolean:  
    not(c == "none")  
end  
L.filter(real-code, codes)
```

This program might feel a bit verbose: do we really need to write a helper function just to perform something as simple as a `filter`? Wouldn't it be easier to just write something like:

```
L.filter(not(c == "none"), codes)
```

Do Now!

What will Pyret produce if you run this expression?

Pyret will produce an `unbound identifier` error around the use of `c` in this expression. What is `c`? We mean for `c` to be the elements from `codes` in turn. Conceptually, that's what `filter` does, but we don't have the mechanics right. When we call a function, we evaluate the arguments before the body of the function. Hence, the error regarding `c` being unbound. The whole point of the `real-code` helper function is to make `c` a parameter to a function whose body is only evaluated once values for `c` is available.

To tighten the notation as in the one-line `filter` expression, then, we have to find a way to tell Pyret to make a temporary function that will get its inputs once `filter` is running. The following notation achieves this:

```
L.filter(lam(c): not(c == "none") end, codes)
```

We have added `lam(c)` and `end` around the expression that we want to use in the `filter`. The `lam(c)` says "make a temporary function that takes `c` as an input". The `end` serves to end the function definition, as when we use `fun`. `lam` is short for `lambda`, a form of function definition that exists in many, though not all, languages.

The main difference between our original expression (using the `real-code` helper) and this new one (using `lam`) can be seen through the program directory. To explain this, a little detail about how `filter` is defined under the hood. In part, it looks like:

```
fun filter(keep :: (A -> Boolean), lst :: List<A>) -> List<A>:  
    if keep(<elt-from-list>):  
        ...  
    else:  
        ...  
    end  
end
```

Whether we pass `real-code` or the `lam` version to `filter`, the `keep` parameter ends up referring to a function with the same parameter and body. Since the function is only actually called through the `keep` name, it doesn't matter whether or not a name is associated with it when it is initially defined.

In practice, we use `lam` when we have to pass simple (single line) functions to operations like `filter` (or `map`). We could have just as easily used them when we were working with tables (`build-column`, `filter-with`, etc). Of course, you can continue to write out names for helper functions as we did with `real-code` if that makes more sense to you.

Exercise

Write the program to extract the list of usernames from a list of email addresses using `lmap` rather than a named helper-function.

5.1.6 Combining Lists and Tables The table functions we studied previously were primarily for processing rows. The list functions we've learned in this chapter have been primarily for processing columns (but there are many more uses in the chapters ahead). If an analysis involves working with only some rows and some columns, we'll use a combination of both table and list functions in our program.

Exercise

Given the events table, produce a list of names of all people who will pick up their tickets.

Exercise

Given the events table, produce the average number of tickets that were ordered by people with email addresses that end in `".org"`.

Sometimes, there will be more than one way to perform a computation:

Do Now!

Consider a question such as "how many people with `".org"` email addresses bought more than 8 tickets". Propose multiple task plans that would solve this problem, including which table and list functions would accomplish each task.

There are several options here:

1. Get the `event-data` rows with no more than 8 tickets (using `filter-with`), get those rows that have `".org"` addresses (another `filter-with`), then ask for how many rows are in the table (using `<table>.length()`).
2. Get the `event-data` rows with no more than 8 tickets and `".org"` address (using `filter-with` with a function that checks both conditions at once), then ask for how many rows are in the table (using `<table>.length()`).
3. Get the `event-data` rows with no more than 8 tickets (using `filter-with`), extract the email addresses (using `get-column`), limit those to `".org"` (using `L.filter`), then get the length of the resulting list (using `L.length`).

There are others, but you get the idea.

Do Now!

Which approach do you like best? Why?

While there is no single correct answer, there are various considerations:

- Are any of the intermediate results useful for other computations? While the second option might seem best because it filters the table once rather than twice, perhaps the events company has many computations to perform on larger ticket orders. Similarly, the company may want the list of email addresses on large orders for other purposes (the third option)
- Do you want to follow a discipline of doing operations on individuals within the table, extracting lists only when needed to perform aggregating computations that aren't available on tables?
- Does one approach seem less resource-intensive than the other? This is actually a subtle point: you might be tempted to think that filtering over a table uses more resources than filtering over a list of values from one column, but this actually isn't the case. We'll return to this discussion later.

A company or project team sometimes sets design standards to help you make those decisions. In the absence of that, and especially as you are learning to program, consider multiple approaches when faced with such problems, then pick one to implement. Maintaining the ability to think flexibly about approaches is a useful skill in any form of design.

Until now we've only seen how to use built-in functions over lists. Next [Processing Lists], we will study how to create our own functions that process lists. Once we learn that, these list processing functions will remain powerful but will no longer seem quite so magical, because we'll be able to build them for ourselves!

5.2 Processing Lists

5.2 Processing Lists We have already seen [From Tables to Lists] several examples of list-processing functions. They have been especially useful for advanced processing of tables. However, lists arise frequently in programs, and they do so naturally because so many things in our lives—from shopping lists to to-do lists to checklists—are naturally lists. Thinking about the functions that we might want when processing lists, we can observe that there are some interesting categories regarding the types of the data in the list:

- some list functions are generic and operate on any kind of list: e.g., the length of a list is the same irrespective of what kind of values it contains;
- some are specific at least to the type of data: e.g., the sum assumes that all the values are numbers (though they may be ages or prices or other information represented by numbers); and
- some are somewhere in-between: e.g., a maximum function applies to any list of comparable values, such as numbers or strings.

This seems like a great variety, and we might worry about how we can handle this many different kinds of functions. Fortunately, and perhaps surprisingly, there is one standard way in which we can think about writing all these functions! Understanding and internalizing this process is the goal of this chapter.

5.2.1 Making Lists and Taking Them Apart So far we've seen one way to make a list: by writing `[list: ...]`. While useful, writing lists this way actually hides their true nature. Every list actually has two parts: a first element and the rest of the list. The rest of the list is itself a list, so it too has two parts...and so on.

Consider the list `[list: 1, 2, 3]`. Its first element is 1, and the rest of it is `[list: 2, 3]`. For this second list, the first element is 2 and the rest is `[list: 3]`.

Do Now!

Take apart this third list.

For the third list, the first element is 3 and the rest is `[list:]`, i.e., the empty list. In Pyret, we have another way of writing the empty list: `empty`.

Lists are an instance of structured data: data with component parts and a well-defined format for the shape of the parts. Lists are formatted by the first element and the rest of the elements. Tables are somewhat structured: they are formatted by rows and columns, but the column names aren't consistent across all tables. Structured data is valuable in programming because a predictable format (the structure) lets us write programs based on that structure. What do we mean by that?

Programming languages can (and do!) provide built-in operators for taking apart structured data. These operators are called accessors. Accessors are defined on the structure of the datatype alone, independent of the contents of the data. In the case of lists, there are two accessors: `first` and `rest`. We use an accessor by writing an expression, followed by a dot (`.`), followed by the accessor's name. As we saw with tables, the dot means "dig into". Thus:

```
l1 = [list: 1, 2, 3]
e1 = l1.first
l2 = l1.rest
e2 = l2.first
l3 = l2.rest
e3 = l3.first
l4 = l3.rest
```

```

check:
  e1 is 1
  e2 is 2
  e3 is 3
  l2 is [list: 2, 3]
  l3 is [list: 3]
  l4 is empty
end

```

Do Now!

What are the accessors for tables?

Accessors give a way to take data apart based on their structure (there is another way that we will see shortly). Is there a way to also build data based on its structure? So far, we have been building lists using the `[list: ...]` form, but that doesn't emphasize the structural constraint that the `rest` is itself a list. A structured operator for building lists would clearly show both a `first` element and a `rest` that is itself a list. Operators for building structured data are called constructors.

The constructor for lists is called `link`. It takes two arguments: a `first` element, and the list to build on (the `rest` part). Here's an example of using `link` to create a three-element list.

```
link(1, link(2, link(3, empty)))
```

The `link` form creates the same underlying list datum as our previous `[list: ...]` operation, as confirmed by the following check:

```

check:
  [list: 1, 2, 3] is link(1, link(2, link(3, empty)))
end

```

Do Now!

Look at these two forms of writing lists: what differences do you notice?

Do Now!

Use the `link` form to write a four-element list of fruits containing "lychee", "dates", "mango", and "durian".

After doing this exercise, you might wonder why anyone would use the `link` form: it's more verbose, and makes the individual elements harder to discern. This form is not very convenient to humans. But it will prove very valuable to programs!

In particular, the `link` form highlights that we really have two different structures of lists. Some lists are empty. All other lists are non-empty lists, meaning they have at least one `link`. There may be more interesting structure to some lists (as we will see later), but all lists have this much in common. Specifically, a list is either

- empty (written `empty` or `[list:]`), or
- non-empty (written `link(..., ...)` or `[list:]` with at least one value inside the brackets), where the rest is also a list (and hence may in turn be empty or non-empty, ...).

This means we actually have two structural features of lists, both of which are important when writing programs over lists:

1. Lists can be empty or non-empty
2. Non-empty lists have a first element and a rest of the list

Let's leverage these two structural features to write some programs to process lists!

5.2.2 Some Example Exercises To illustrate our thinking, let's work through a few concrete examples of list-processing functions. All of these will consume lists; some will even produce them. Some will transform their inputs (like `map`), some will select from their inputs (like `filter`), and some will aggregate their inputs. Since some of these functions already exist in Pyret, we'll name them with the prefix `my-` to avoid errors. Be sure to use the `my-`

name consistently, including inside the body of the function. As we will see, there is a standard strategy that we can use to approach writing all of these functions: having you learn this strategy is the goal of this chapter.

5.2.3 Structural Problems with Scalar Answers Let's write out examples for a few of the functions described above. We'll approach writing examples in a very specific, stylized way. First of all, we should always construct at least two examples: one with `empty` and the other with at least one `link`, so that we've covered the two very broad kinds of lists. Then, we should have more examples specific to the kind of list stated in the problem. Finally, we should have even more examples to illustrate how we think about solving the problem.

5.2.3.1 `my-len`: Examples We haven't precisely defined what it means to be "the length" of a list. We confront this right away when trying to write an example. What is the length of the list `empty`?

Do Now!

What do you think?

Two common examples are 0 and 1. The latter, 1, certainly looks reasonable. However, if you write the list as `[list:]`, now it doesn't look so right: this is clearly (as the name `empty` also suggests) an empty list, and an empty list has zero elements in it. Therefore, it's conventional to declare that

```
my-len(empty) is 0
```

How about a list like `[list: 7]`? Well, it's clearly got one element (7) in it, so

```
my-len([list: 7]) is 1
```

Similarly, for a list like `[list: 7, 8, 9]`, we would say

```
my-len([list: 7, 8, 9]) is 3
```

Now let's look at that last example in a different light. Consider the argument `[list: 7, 8, 9]`. Its first element is 7 and the rest of it is `[list: 8, 9]`. Well, 7 is a number, not a list; but `[list: 8, 9]` certainly is a list, so we can ask for its length. What is `my-len([list: 8, 9])`? It has two elements, so

```
my-len([list: 8, 9]) is 2
```

The first element of that list is 8 while its rest is `[list: 9]`. What is its length? Note that we asked a very similar question before, for the length of the list `[list: 7]`. But `[list: 7]` is not a sub-list of `[list: 7, 8, 9]`, which we started with, whereas `[list: 9]` is. And using the same reasoning as before, we can say

```
my-len([list: 9]) is 1
```

The rest of this last list is, of course, the empty list, whose length we have already decided is 0.

Putting together these examples, and writing out `empty` in its other form, here's what we get:

```
my-len([list: 7, 8, 9]) is 3
my-len([list:     8, 9]) is 2
my-len([list:         9]) is 1
my-len([list:             ]) is 0
```

Another way we can write this (paying attention to the right side) is

```
my-len([list: 7, 8, 9]) is 1 + 2
my-len([list:     8, 9]) is 1 + 1
my-len([list:         9]) is 1 + 0
my-len([list:             ]) is 0
```

Where did the 2, 1, and 0 on the right sides of each `+` operation come from? Those are the lengths of the `rest` component of the input list. In the previous example block, we wrote those lengths as explicit examples. Let's substitute the numbers 2, 1, and 0 with the `my-len` expressions that produce them:

```
my-len([list: 7, 8, 9]) is 1 + my-len([list: 8, 9])
my-len([list:     8, 9]) is 1 + my-len([list:         9])
my-len([list:         9]) is 1 + my-len([list:             ])
my-len([list:             ]) is 0
```

From this, maybe you can start to see a pattern. For an empty list, the length is 0. For a non-empty list, it's the sum of 1 (the first element's “contribution” to the list's length) to the length of the rest of the list. In other words, we can use the result of computing `my-len` on the rest of the list to compute the answer for the entire list.

Do Now!

Each of our examples in this section has written a different check on the expression `my-len([list: 7, 8, 9])`. Here are those examples presented together, along with one last one that explicitly uses the `rest` operation:

```
my-len([list: 7, 8, 9]) is 3
my-len([list: 7, 8, 9]) is 1 + 2
my-len([list: 7, 8, 9]) is 1 + my-len([list: 8, 9])
my-len([list: 7, 8, 9]) is 1 + my-len([list: 7, 8, 9].rest)
```

Check that you agree with each of these assertions. Also check whether you understand how the right-hand side of each `is` expression derives from the right-hand-side just above it. The goal of this exercise is to make sure that you believe that the last check (which we will turn into code) is equivalent to the first (which we wrote down when understanding the problem).

5.2.3.2 `my-sum`: Examples Let's repeat this process of developing examples on a second function, this time one that computes the sum of the elements in a list of numbers. What is the sum of the list `[list: 7, 8, 9]`? Just adding up the numbers by hand, the result should be 24. Let's see how that works out through the examples.

Setting aside the empty list for a moment, here are examples that show the sum computations:

```
my-sum([list: 7, 8, 9]) is 7 + 8 + 9
my-sum([list:    8, 9]) is      8 + 9
my-sum([list:      9]) is          9
```

which (by substitution) is the same as

```
my-sum([list: 7, 8, 9]) is 7 + my-sum([list: 8, 9])
my-sum([list:    8, 9]) is 8 + my-sum([list:      9])
my-sum([list:      9]) is 9 + my-sum([list:        ])
```

From this, we can see that the sum of the empty list must be 0:Zero is called the additive identity: a fancy way of saying, adding zero to any number N gives you N. Therefore, it makes sense that it would be the length of the empty list, because the empty list has no items to contribute to a sum. Can you figure out what the multiplicative identity is?

```
my-sum(empty) is 0
```

Observe, again, how we can use the result of computing `my-sum` of the rest of the list to compute its result for the whole list.

5.2.3.3 From Examples to Code Having developed these examples, we now want to use them to develop a program that can compute the length or the sum of any list, not just the specific ones we used in these examples. As we have done up in earlier chapters, we will leverage patterns in the examples to figure out how to define the general-purpose function.

Here is one last version of the examples for `my-len`, this time making the `rest` explicit on the right-hand sides of `is`:

```
my-len([list: 7, 8, 9]) is 1 + my-len([list: 7, 8, 9].rest)
my-len([list:    8, 9]) is 1 + my-len([list:      8, 9].rest)
my-len([list:      9]) is 1 + my-len([list:          9].rest)
my-len([list:        ]) is 0
```

As we did when developing functions over images, let's try to identify the common parts of these examples. We start by noticing that most of the examples have a lot in common, except for the `[list:]` (`empty`) case. So let's separate this into two sets of examples:

```
my-len([list: 7, 8, 9]) is 1 + my-len([list: 7, 8, 9].rest)
my-len([list:    8, 9]) is 1 + my-len([list:      8, 9].rest)
```

```
my-len([list:      9]) is 1 + my-len([list:      9].rest)
```

```
my-len([list:      ]) is 0
```

With this separation (which follows one of the structural features of lists that we mentioned earlier), a clearer pattern emerges: for a non-empty list (called `someList`), we compute its length via the expression:

```
1 + my-len(someList.rest)
```

In general, then, our `my-len` program needs to determine whether its input list is empty or non-empty, using this expression with `.rest` in the non-empty case. How do we indicate different code based on the structure of the list?

Pyret has a construct called `cases` which is used to distinguish different forms within a structured datatype. When working with lists, the general shape of a `cases` expression is:

```
cases (List) e:  
| empty      => ...  
| link(f, r) => ... f ... r ...  
end
```

where most parts are fixed, but a few you're free to change:

- `e` is an expression whose value needs to be a list; it could be a variable bound to a list, or some complex expression that evaluates to a list.
- `f` and `r` are names given to the first and rest of the list. You can choose any names you like, though in Pyret, it's conventional to use `f` and `r`. Occasionally using different names can help students recall that they can choose how to label the `first` and `rest` components. This can be particularly useful for `first`, which has a problem-specific meaning (such as `price` in a list of prices, and so on).

The right-hand side of every `=>` is an expression.

Here's how `cases` works in this instance. Pyret first evaluates `e`. It then checks that the resulting value truly is a list; otherwise it halts with an error. If it is a list, Pyret examines what kind of list it is. If it's an empty list, it runs the expression after the `=>` in the `empty` clause. Otherwise, the list is not empty, which means it has a first and rest; Pyret binds `f` and `r` to the two parts, respectively, and then evaluates the expression after the `=>` in the `link` clause.

Exercise

Try using a non-list—e.g., a number—in the `e` position and see what happens!

Now let's use `cases` to define `my-len`:

```
fun my-len(l):  
  cases (List) l:  
    | empty      => 0  
    | link(f, r) => 1 + my-len(r)  
  end  
end
```

This follows from our examples: when the list is empty `my-len` produces 0; when it is not empty, we add one to the length of the rest of the list (here, `r`).

Note that while our most recent collection of `my-len` examples explicitly said `.rest`, when using `cases` we instead use just the name `r`, which Pyret has already defined (under the hood) to be `l.rest`.

Similarly, let's define `my-sum`:

```
fun my-sum(l):  
  cases (List) l:  
    | empty      => 0  
    | link(f, r) => f + my-sum(r)  
  end  
end
```

Notice how similar they are in code, and how readily the structure of the data suggest a structure for the program. This is a pattern you will get very used to soon!

Strategy: Developing Functions Over Lists

Leverage the structure of lists and the power of concrete examples to develop list-processing functions.

- Pick a concrete list with (at least) three elements. Write a sequence of examples for each of the entire list and each suffix of the list (including the empty list).
- Rewrite each example to express its expected answer in terms of the `first` and `rest` data of its input list. You don't have to use the `first` and `rest` operators in the new answers, but you should see the `first` and `rest` values represented explicitly in the answer.
- Look for a pattern across the answers in the examples. Use these to develop the code: write a `cases` expression, filling in the right side of each `=>` based on your examples.

This strategy applies to structured data in general, leveraging components of each datum rather than specifically `first` and `rest` as presented so far.

5.2.4 Structural Problems that Transform Lists Now that we have a systematic way to develop functions that take lists as input, let's apply that same strategy to functions that produce a list as the answer.

5.2.4.1 `my-doubles`: Examples and Code As always, we'll begin with some examples. Given a list of numbers, we want a list that doubles each number (in the order of the original list). Here's a reasonable example with three numbers:

```
my-doubles([list: 3, 5, 2]) is [list: 6, 10, 4]
```

As before, let's write out the answers for each suffix of our example list as well, including for the `empty` list:

```
my-doubles([list: 5, 2]) is [list: 10, 4]
my-doubles([list: 2]) is [list: 4]
my-doubles([list: ]) is [list: ]
```

Now, we rewrite the answer expressions to include the concrete `first` and `rest` data for each example. Let's start with just the `first` data, and just on the first example:

```
my-doubles([list: 3, 5, 2]) is [list: 3 * 2, 10, 4]
my-doubles([list: 5, 2]) is [list: 10, 4]
my-doubles([list: 2]) is [list: 4]
my-doubles([list: ]) is [list: ]
```

Next, let's include the `rest` data (`[list: 5, 2]`) in the first example. The current answer in the first example is

```
[list: 3 * 2, 10, 4]
```

and that `[list: 10, 4]` is the result of using the function on `[list: 5, 2]`. We might therefore be tempted to replace the right side of the first example with:

```
[list: 3 * 2, my-doubles([list: 5, 2])]
```

Do Now!

What value would this expression produce? You might want to try this example that doesn't use `my-doubles` directly:

```
[list: 3 * 2, [list: 10, 4]]
```

Oops! We want a single (flat) list, not a list-within-a-list. This feels like it is on the right track in terms of reworking the answer to use the `first` and `rest` values, but we're clearly not quite there yet.

Do Now!

What value does the following expression produce?

```
link(3 * 2, [list: 10, 4])
```

Notice the difference between the two expressions in these last two exercises: the latter used `link` to put the value involving `first` into the conversion of the `rest`, while the former tried to do this with `list:`.

Do Now!

How many elements are in the lists that result from each of the following expressions?

```
[list: 25, 16, 32]  
[list: 25, [list: 16, 32]]  
link(25, [list: 16, 32])
```

Do Now!

Summarize the difference between how `link` and `list:` combine an element and a list. Try additional examples at the interactions prompt if needed to explore these ideas.

The takeaway here is that we use `link` to insert an element into an existing list, whereas we use `list:` to make a new list that contains the old list as an element. Going back to our examples, then, we include `rest` in the first example by writing it as follows:

```
my-doubles([list: 3, 5, 2]) is link(3 * 2, [list: 10, 4])  
my-doubles([list: 5, 2]) is [list: 10, 4]  
my-doubles([list: 2]) is [list: 4]  
my-doubles([list: ]) is [list: ]
```

which we then convert to

```
my-doubles([list: 3, 5, 2]) is link(3 * 2, my-doubles([list: 5, 2]))  
my-doubles([list: 5, 2]) is [list: 10, 4]  
my-doubles([list: 2]) is [list: 4]  
my-doubles([list: ]) is [list: ]
```

Applying this idea across the examples, we get:

```
my-doubles([list: 3, 5, 2]) is link(3 * 2, my-doubles([list: 5, 2]))  
my-doubles([list: 5, 2]) is link(5 * 2, my-doubles([list: 2]))  
my-doubles([list: 2]) is link(2 * 2, my-doubles([list: ]))  
my-doubles([list: ]) is [list: ]
```

Now that we have examples that explicitly use the `first` and `rest` elements, we can produce to write the `my-doubles` function:

```
fun my-doubles(l):  
  cases (List) l:  
    | empty => empty  
    | link(f, r) =>  
      link(f * 2, my-doubles(r))  
  end  
end
```

5.2.4.2 my-str-len: Examples and Code In `my-doubles`, the input and output lists have the same type of element. Functions can also produce lists whose contents have a different type from the input list. Let's work through an example. Given a list of strings, we want the lengths of each string (in the same order as in the input list). Thus, here's a reasonable example:

```
my-str-len([list: "hi", "there", "mateys"]) is [list: 2, 5, 6]
```

As we have before, we should consider the answers for each sub-problem of the above example:

```
my-str-len([list: "there", "mateys"]) is [list: 5, 6]  
my-str-len([list: "mateys"]) is [list: 6]
```

Or, in other words:

```

my-str-len([list: "hi", "there", "mateys"]) is link(2, [list: 5, 6])
my-str-len([list:      "there", "mateys"]) is link(5, [list:      6])
my-str-len([list:          "mateys"]) is link(6, [list:          ])

```

which tells us that the response for the empty list should be `empty`:

```
my-str-len(empty) is empty
```

The next step is to rework the answers in the examples to make the `first` and `rest` parts explicit. Hopefully by now you are starting to detect a pattern: The result on the rest of the list appears explicitly as another example. Therefore, we'll start by getting the `rest` value of each example input into the answer:

```

my-str-len([list: "hi", "there", "mateys"]) is link(2, my-str-len([list: "there", "mateys"]))
my-str-len([list:      "there", "mateys"]) is link(5, my-str-len([list:      "mateys"]))
my-str-len([list:          "mateys"]) is link(6, my-str-len([list:          ]))
my-str-len([list:          ]) is [list: ]

```

All that remains now is to figure out how to work the `first` values into the outputs. In the context of this problem, this means we need to convert `"hi"` into 2, `"there"` into 5, and so on. From the problem statement, we know that 2 and 5 are meant to be the lengths (character counts) of the corresponding strings. The operation that determines the length of a string is called `string-length`. Thus, our examples appear as:

```

my-str-len([list: "hi", "there", "mateys"]) is link(string-length("hi"), my-str-len([list: "there", "mateys"]))
my-str-len([list:      "there", "mateys"]) is link(string-length("there"), my-str-len([list:      "mateys"]))
my-str-len([list:          "mateys"]) is link(string-length("mateys"), my-str-len([list:          ]))
my-str-len([list:          ]) is [list: ]

```

From here, we write a function that captures the pattern developed across our examples:

```

fun my-str-len(l):
    cases (List) l:
        | empty => empty
        | link(f, r) =>
            link(string-length(f), my-str-len(r))
    end
end

```

5.2.5 Structural Problems that Select from Lists In the previous section, we saw functions that transform list elements (by doubling numbers or counting characters). The type of the output list may or may not be the same as the type of the input list. Other functions that produce lists instead select elements: every element in the output list was in the input list, but some input-list elements might not appear in the output list. This section adapts our method of deriving functions from examples to accommodate selection of elements.

5.2.5.1 my-pos-nums: Examples and Code As our first example, we will select the positive numbers from a list that contains both positive and non-positive numbers.

Do Now!

Construct the sequence of examples that we obtain from the input `[list: 1, -2, 3, -4]`.

Here we go:

```

my-pos-nums([list: 1, -2, 3, -4]) is [list: 1, 3]
my-pos-nums([list:      -2, 3, -4]) is [list:      3]
my-pos-nums([list:          3, -4]) is [list:      3]
my-pos-nums([list:          -4]) is [list:          ]
my-pos-nums([list:          ]) is [list:          ]

```

We can write this in the following form:

```

my-pos-nums([list: 1, -2, 3, -4]) is link(1, [list: 3])
my-pos-nums([list:      -2, 3, -4]) is           [list: 3]
my-pos-nums([list:          3, -4]) is link(3, [list: ])

```

```
my-pos-nums([list: -4]) is [list: ]
my-pos-nums([list: ]) is [list: ]
```

or, even more explicitly,

```
my-pos-nums([list: 1, -2, 3, -4]) is link(1, my-pos-nums([list: -2, 3, -4]))
my-pos-nums([list: -2, 3, -4]) is my-pos-nums([list: 3, -4])
my-pos-nums([list: 3, -4]) is link(3, my-pos-nums([list: -4]))
my-pos-nums([list: -4]) is my-pos-nums([list: ])
my-pos-nums([list: ]) is [list: ]
```

Unlike in the example sequences for functions that transform lists, here we see that the answers have different shapes: some involve a `link`, while others simply process the `rest` of the list. Whenever we need different shapes of outputs across a set of examples, we will need an `if` expression in our code to distinguish the conditions that yield each shape.

What determines which shape of output we get? Let's rearrange the examples (other than the empty-list input) by output shape:

```
my-pos-nums([list: 1, -2, 3, -4]) is link(1, my-pos-nums([list: -2, 3, -4]))
my-pos-nums([list: 3, -4]) is link(3, my-pos-nums([list: -4]))

my-pos-nums([list: -2, 3, -4]) is my-pos-nums([list: 3, -4])
my-pos-nums([list: -4]) is my-pos-nums([list: ])
```

Re-organized, we can see that the examples that use `link` have a positive number in the `first` position, while the ones that don't simply process the `rest` of the list. That indicates that our `if` expression needs to ask whether the `first` element in the list is positive. This yields the following program:

```
fun my-pos-nums(l):
    cases (List) l:
        | empty => empty
        | link(f, r) =>
            if f > 0:
                link(f, my-pos-nums(r))
            else:
                my-pos-nums(r)
            end
        end
    end
```

Do Now!

Is our set of examples comprehensive?

Not really. There are many examples we haven't considered, such as lists that end with positive numbers and lists with 0.

Exercise

Work through these examples and see how they affect the program!

5.2.5.2 my-alternating: Examples and Code Now let's consider a problem that selects elements not by value, but by position. We want to write a function that selects alternating elements from a list. Once again, we're going to work from examples.

Do Now!

Work out the results for `my-alternating` starting from the list `[list: 1, 2, 3, 4, 5, 6]`.

Here's how they work out:

<alternating-egs-1> ::=

```

check:
my-alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
my-alternating([list: 2, 3, 4, 5, 6]) is [list: 2, 4, 6]
my-alternating([list: 3, 4, 5, 6]) is [list: 3, 5]
my-alternating([list: 4, 5, 6]) is [list: 4, 6]
end

```

Wait, what's that? The two answers above are each correct, but the second answer does not help us in any way construct the first answer. That means the way we've solved these problems until now is not enough for this new kind of problem. It's still useful, though: notice that there's a connection between the first example and the third, as well as between the second example and the fourth. This observation is consistent with our goal of selecting alternating elements.

What would something like this look like in code? Before we try to write the function, let's rewrite the first example in terms of the third:

```

my-alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
my-alternating([list: 3, 4, 5, 6]) is [list: 3, 5]

my-alternating([list: 1, 2, 3, 4, 5, 6]) is link(1, my-alternating([list: 3, 4, 5, 6]))

```

Note that in the rewritten version, we are dropping two elements from the list before using `my-alternating` again, not just one. We will have to figure out how to handle that in our code.

Let's start with our usual function pattern with a `cases` expression:

```

fun my-alternating(l):
  cases (List) l:
    | empty => [list:]
    | link(f, r) => link(f, ... r ...)
  end
end

```

Note that we cannot simply call `my-alternating` on `r`, because `r` excludes only one item from the list, not two as this problem requires. We have to break down `r` as well, in order to get to the `rest` of the `rest` of the original list. To do this, we use another `cases` expression, nested within the first `cases` expression:

```

fun my-alternating(l):
  cases (List) l:
    | empty => [list:]
    | link(f, r) =>
      cases (List) r: # note: deconstructing r, not l
        | empty => ??? # note the ???
        | link(fr, rr) =>
          # fr = first of rest, rr = rest of rest
          link(f, my-alternating(rr))
      end
  end
end

```

This code is consistent with the example that we just worked out. But note that we still have a bit of unfinished work to do: we need to decide what to do in the `empty` case of the inner `cases` expression (marked by `???` in the code).

A common temptation at this point is to replace the `???` with `[list:]`. After all, haven't we always returned `[list:]` in the `empty` cases?

Do Now!

Replace `???` with `[list:]` and test the program on our original examples:

```

my-alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
my-alternating([list: 2, 3, 4, 5, 6]) is [list: 2, 4, 6]

```

```
my-alternating([list:      3, 4, 5, 6]) is [list:    3, 5]
my-alternating([list:          4, 5, 6]) is [list:    4, 6]
```

What do you observe?

Oops! We've written a program that appears to work on lists with an even number of elements, but not on lists with an odd number of elements. How did that happen? The only part of this code that we guessed at was how to fill in the `empty` case of the inner `cases`, so the issue must be there. Rather than focus on the code, however, focus on the examples. We need a simple example that would land on that part of the code. We get to that spot when the list `l` is not empty, but `r` (the rest of `l`) is empty. In other words, we need an example with only one element.

Do Now!

Finish the following example:

```
my-alternating([list: 5]) is ???
```

Given a list with one element, that element should be included in a list of alternating elements. Thus, we should finish this example as

```
my-alternating([list: 5]) is [list: 5]
```

Do Now!

Use this example to update the result of `my-alternating` when `r` is `empty` in our code.

Leveraging this new example, the final version of `my-alternating` is as follows:

```
fun my-alternating(l):
  cases (List) l:
    | empty => empty
    | link(f, r) =>
      cases (List) r: # note: deconstructing r, not l
        | empty =>      # the list has an odd number of elements
          [list: f]
        | link(fr, rr) =>
          # fr = first of rest, rr = rest of rest
          link(f, my-alternating(rr))
      end
    end
  end
```

What's the takeaway from this problem? There are two:

- Don't skip the small examples: the result of a list-processing function on the `empty` case won't always be `empty`.
- If a problem asks you to work with multiple elements from the front of a list, you can nest `cases` expressions to access later elements.

These takeaways will matter again in future examples: keep an eye out for them!

5.2.6 Structural Problems Over Relaxed Domains

5.2.6.1 `my-max`: Examples Now let's find the maximum value of a list. Let's assume for simplicity that we're dealing with just lists of numbers. What kinds of lists should we construct? Clearly, we should have empty and non-empty lists...but what else? Is a list like `[list: 1, 2, 3]` a good example? Well, there's nothing wrong with it, but we should also consider lists where the maximum at the beginning rather than at the end; the maximum might be in the middle; the maximum might be repeated; the maximum might be negative; and so on. While not comprehensive, here is a small but interesting set of examples:

```
my-max([list: 1, 2, 3]) is 3
my-max([list: 3, 2, 1]) is 3
my-max([list: 2, 3, 1]) is 3
```

```
my-max([list: 2, 3, 1, 3, 2]) is 3
my-max([list: 2, 1, 4, 3, 2]) is 4
my-max([list: -2, -1, -3]) is -1
```

What about `my-max(empty)`?

Do Now!

Could we define `my-max(empty)` to be 0? Returning 0 for the empty list has worked well twice already!

We'll return to this in a while.

Before we proceed, it's useful to know that there's a function called `num-max` already defined in Pyret, that compares two numbers:

```
num-max(1, 2) is 2
num-max(-1, -2) is -1
```

Exercise

Suppose `num-max` were not already built in. Can you define it? You will find what you learned about Booleans handy. Remember to write some tests!

Now we can look at `my-max` at work:

```
my-max([list: 1, 2, 3]) is 3
my-max([list: 2, 3]) is 3
my-max([list: 3]) is 3
```

Hmm. That didn't really teach us anything, did it? Maybe, we can't be sure. And we still don't know what to do with `empty`.

Let's try the second example input:

```
my-max([list: 3, 2, 1]) is 3
my-max([list: 2, 1]) is 2
my-max([list: 1]) is 1
```

This is actually telling us something useful as well, but maybe we can't see it yet. Let's take on something more ambitious:

```
my-max([list: 2, 1, 4, 3, 2]) is 4
my-max([list: 1, 4, 3, 2]) is 4
my-max([list: 4, 3, 2]) is 4
my-max([list: 3, 2]) is 3
my-max([list: 2]) is 2
```

Observe how the maximum of the rest of the list gives us a candidate answer, but comparing it to the first element gives us a definitive one:

```
my-max([list: 2, 1, 4, 3, 2]) is num-max(2, 4)
my-max([list: 1, 4, 3, 2]) is num-max(1, 4)
my-max([list: 4, 3, 2]) is num-max(4, 3)
my-max([list: 3, 2]) is num-max(3, 2)
my-max([list: 2]) is ...
```

The last one is a little awkward: we'd like to write

```
my-max([list: 2]) is num-max(2, ...)
```

but we don't really know what the maximum (or minimum, or any other element) of the empty list is, but we can only provide numbers to `num-max`. Therefore, leaving out that dodgy case, we're left with

```
my-max([list: 2, 1, 4, 3, 2]) is num-max(2, my-max([list: 1, 4, 3, 2]))
my-max([list: 1, 4, 3, 2]) is num-max(1, my-max([list: 4, 3, 2]))
my-max([list: 4, 3, 2]) is num-max(4, my-max([list: 3, 2]))
```

```
my-max([list:           3, 2]) is num-max(3, my-max([list:           2]))
```

Our examples have again helped: they've revealed how we can use the answer for each rest of the list to compute the answer for the whole list, which in turn is the rest of some other list, and so on. If you go back and look at the other example lists we wrote above, you'll see the pattern holds there too.

However, it's time we now confront the `empty` case. The real problem is that we don't have a maximum for the empty list: for any number we might provide, there is always a number bigger than it (assuming our computer is large enough) that could have been the answer instead. In short, it's nonsensical to ask for the maximum (or minimum) of the empty list: the concept of "maximum" is only defined on non-empty lists! That is, when asked for the maximum of an empty list, we should signal an error:

```
my-max(empty) raises ""
```

(which is how, in Pyret, we say that it will generate an error; we don't care about the details of the error, hence the empty string).

5.2.6.2 `my-max`: From Examples to Code Once again, we can codify the examples above, i.e., turn them into a uniform program that works for all instances. However, we now have a twist. If we blindly followed the pattern we've used earlier, we would end up with:

```
fun my-max(l):
  cases (List) l:
    | empty      => raise("not defined for empty lists")
    | link(f, r) => num-max(f, my-max(r))
  end
end
```

Do Now!

What's wrong with this?

Consider the list `[list: 2]`. This turns into

```
num-max(2, my-max([list: ]))
```

which of course raises an error. Therefore, this function never works for any list that has one or more elements!

That's because we need to make sure we aren't trying to compute the maximum of the empty list. Going back to our examples, we see that what we need to do, before calling `my-max`, is check whether the rest of the list is empty. If it is, we do not want to call `my-max` at all. That is:

```
fun my-max(l):
  cases (List) l:
    | empty      => raise("not defined for empty lists")
    | link(f, r) =>
      cases (List) r:
        | empty => ...
        | ...
      end
    end
  end
```

We'll return to what to do when the rest is not empty in a moment.

If the rest of the list `l` is empty, our examples above tell us that the maximum is the first element in the list. Therefore, we can fill this in:

```
fun my-max(l):
  cases (List) l:
    | empty      => raise("not defined for empty lists")
    | link(f, r) =>
      cases (List) r:
        | empty => f
```

```

| ...
end
end

```

Note in particular the absence of a call to `my-max`. If the list is not empty, however, our examples above tell us that `my-max` will give us the maximum of the rest of the list, and we just need to compare this answer with the first element (`f`):

```

fun my-max(l):
  cases (List) l:
    | empty      => raise("not defined for empty lists")
    | link(f, r) =>
      cases (List) r:
        | empty => f
        | else   => num-max(f, my-max(r))
      end
    end
  end
end

```

And sure enough, this definition does the job!

5.2.7 More Structural Problems with Scalar Answers

5.2.7.1 `my-avg`: Examples Let's now try to compute the average of a list of numbers. Let's start with the example list `[list: 1, 2, 3, 4]` and work out more examples from it. The average of numbers in this list is clearly $(1 + 2 + 3 + 4)/4$, or $10/4$.

Based on the list's structure, we see that the rest of the list is `[list: 2, 3, 4]`, and the rest of that is `[list: 3, 4]`, and so on. The resulting averages are:

```

my-avg([list: 1, 2, 3, 4]) is 10/4
my-avg([list: 2, 3, 4]) is 9/3
my-avg([list: 3, 4]) is 7/2
my-avg([list: 4]) is 4/1

```

The problem is, it's simply not clear how we get from the answer for the sub-list to the answer for the whole list. That is, given the following two bits of information:

- The average of the remainder of the list is $9/3$, i.e., 3 .
- The first number in the list is 1 .

How do we determine that the average of the whole list must be $10/4$? If it's not clear to you, don't worry: with just those two pieces of information, it's impossible!

Here's a simpler example that explains why. Let's suppose the first value in a list is 1 , and the average of the rest of the list is 2 . Here are two very different lists that fit this description:

```

[list: 1, 2]  # the rest has one element with sum 2
[list: 1, 4, 0] # the rest has two elements with sum 4

```

The average of the entire first list is $3/2$, while the average of the entire second list is $5/3$, and the two are not the same.

That is, to compute the average of a whole list, it's not even useful to know the average of the rest of the list. Rather, we need to know the sum and the length of the rest of the list. With these two, we can add the first to the sum, and 1 to the length, and compute the new average.

In principle, we could try to make a `average` function that returns all this information. Instead, it will be a lot simpler to simply decompose the task into two smaller tasks. After all, we have already seen how to compute the length and how to compute the sum. The average, therefore, can just use these existing functions:

```

fun my-avg(l):
    my-sum(l) / my-len(l)
end

```

Do Now!

What should be the average of the empty list? Does the above code produce what you would expect?

Just as we argued earlier about the maximum [Structural Problems Over Relaxed Domains], the average of the empty list isn't a well-defined concept. Therefore, it would be appropriate to signal an error. The implementation above does this, but poorly: it reports an error on division. A better programming practice would be to catch this situation and report the error right away, rather than hoping some other function will report the error.

Exercise

Alter `my-avg` above to signal an error when given the empty list.

Therefore, we see that the process we've used—of inferring code from examples—won't always suffice, and we'll need more sophisticated techniques to solve some problems. However, notice that working from examples helps us quickly identify situations where this approach does and doesn't work. Furthermore, if you look more closely you'll notice that the examples above do hint at how to solve the problem: in our very first examples, we wrote answers like $10/4$, $9/3$, and $7/2$, which correspond to the sum of the numbers divided by the length. Thus, writing the answers in this form (as opposed, for instance, to writing the second of those as 3) already reveals a structure for a solution.

5.2.8 Structural Problems with Accumulators

5.2.8.1 `my-running-sum`: First Attempt

One more time, we'll begin with an example.

Do Now!

Work out the results for `my-running-sum` starting from the list `[list: 1, 2, 3, 4, 5]`.

Here's what our first few examples look like:

```

<running-sum-egs-1> ::=
check:
  my-running-sum([list: 1, 2, 3, 4, 5]) is [list: 1, 3, 6, 10, 15]
  my-running-sum([list:      2, 3, 4, 5]) is [list: 2, 5, 9, 14]
  my-running-sum([list:          3, 4, 5]) is [list: 3, 7, 12]
end

```

Again, there doesn't appear to be any clear connection between the result on the rest of the list and the result on the entire list.

(That isn't strictly true: we can still line up the answers as follows:

```

my-running-sum([list: 1, 2, 3, 4, 5]) is [list: 1, 3, 6, 10, 15]
my-running-sum([list:      2, 3, 4, 5]) is [list:      2, 5, 9, 14]
my-running-sum([list:          3, 4, 5]) is [list:          3, 7, 12]

```

and observe that we're computing the answer for the rest of the list, then adding the first element to each element in the answer, and linking the first element to the front. In principle, we can compute this solution directly, but for now that may be more work than finding a simpler way to answer it.)

5.2.8.2 `my-running-sum`: Examples and Code Recall how we began in `my-running-sum`: First Attempt. Our examples [`<running-sum-egs-1>`] showed the following problem. When we process the rest of the list, we have forgotten everything about what preceded it. That is, when processing the list starting at `2` we forget that we've seen a `1` earlier; when starting from `3`, we forget that we've seen both `1` and `2` earlier; and so on. In other words, we keep forgetting the past. We need some way of avoiding that.

The easiest thing we can do is simply change our function to carry along this “memory”, or what we’ll call an accumulator. That is, imagine we were defining a new function, called `my-rs`. It will consume a list of numbers and produce a list of numbers, but in addition it will also take the sum of numbers preceding the current list.

Do Now!

What should the initial sum be?

Initially there is no “preceding list”, so we will use the additive identity: 0. The type of `my-rs` is

```
my-rs :: Number, List<Number> -> List<Number>
```

Let’s now re-work our examples from `<running-sum-egs-1>` as examples of `my-rs` instead. The examples use the `+` operator to append two lists into one (the elements of the first list followed by the elements of the second):

```
my-rs( 0, [list: 1, 2, 3, 4, 5]) is [list: 0 + 1] + my-rs( 0 + 1, [list: 2, 3, 4, 5])
my-rs( 1, [list: 2, 3, 4, 5]) is [list: 1 + 2] + my-rs( 1 + 2, [list: 3, 4, 5])
my-rs( 3, [list: 3, 4, 5]) is [list: 3 + 3] + my-rs( 3 + 3, [list: 4, 5])
my-rs( 6, [list: 4, 5]) is [list: 6 + 4] + my-rs( 6 + 4, [list: 5])
my-rs(10, [list: 5]) is [list: 10 + 5] + my-rs(10 + 5, [list: ])
my-rs(15, [list: ]) is empty
```

That is, `my-rs` translates into the following code:

```
fun my-rs(acc, l):
    cases (List) l:
        | empty => empty
        | link(f, r) =>
            new-sum = acc + f
            link(new-sum, my-rs(new-sum, r))
    end
end
```

All that’s then left is to call it from `my-running-sum`:

```
fun my-running-sum(l):
    my-rs(0, l)
end
```

Observe that we do not change `my-running-sum` itself to take extra arguments. The correctness of our code depends on the initial value of `acc` being 0. If we added a parameter for `acc`, any code that calls `my-running-sum` could supply an unexpected value, which would distort the result. In addition, since the value is fixed, adding the parameter would amount to shifting additional (and needless) work onto others who use our code.

5.2.8.3 my-alternating: Examples and Code Recall our examples in `my-alternating: Examples and Code`. There, we noticed that the code built on every-other example. We might have chosen our examples differently, so that from one example to the next we skipped two elements rather than one. Here we will see another way to think about the same problem.

Return to the examples we’ve already seen [`<alternating-egs-1>`]. We wrote `my-alternating` to traverse the list essentially two elements at a time. Another option is to traverse it just one element at a time, but keeping track of whether we’re at an odd or even element—i.e., add “memory” to our program. Since we just need to track that one piece of information, we can use a `Boolean` to do it. Let’s define a new function for this purpose:

```
my-alt :: List<Any>, Boolean -> List<Any>
```

The extra argument accumulates whether we’re at an element to keep or one to discard.

We can reuse the existing template for list functions. When we have an element, we have to consult the accumulator whether to keep it or not. If its value is `true` we link it to the answer; otherwise we ignore it. As we process the rest of the list, however, we have to remember to update the accumulator: if we kept an element we don’t wish to keep the next one, and vice versa.

```

fun my-alt(l, keep):
    cases (List) l:
        | empty => empty
        | link(f, r) =>
            if keep:
                link(f, my-alt(r, false))
            else:
                my-alt(r, true)
            end
        end
    end
end

```

Finally, we have to determine the initial value of the accumulator. In this case, since we want to keep alternating elements starting with the first one, its initial value should be `true`:

```

fun my-alternating(l):
    my-alt(l, true)
end

```

Exercise

Define `my-max` using an accumulator. What does the accumulator represent? Do you encounter any difficulty?

5.2.9 Dealing with Multiple Answers Our discussion above has assumed there is only one answer for a given input. This is often true, but it also depends on how the problem is worded and how we choose to generate examples. We will study this in some detail now.

5.2.9.1 uniq: Problem Setup Consider the task of writing `uniq`:`uniq` is the name of a Unix utility with similar behavior; hence the spelling of the name. Given a list of values, it produces a collection of the same elements while avoiding any duplicates (hence `uniq`, short for “unique”).

Consider the following input: `[list: 1, 2, 1, 3, 1, 2, 4, 1]`.

Do Now!

What is the sequence of examples this input generates? It’s really important you stop and try to do this by hand. As we will see there are multiple solutions, and it’s useful for you to consider what you generate. Even if you can’t generate a sequence, trying to do so will better prepare you for what you read next.

How did you obtain your example? If you just “thought about it for a moment and wrote something down”, you may or may not have gotten something you can turn into a program. Programs can only proceed systematically; they can’t “think”. So, hopefully you took a well-defined path to computing the answer.

5.2.9.2 uniq: Examples It turns out there are several possible answers, because we have (intentionally) left the problem unspecified. Suppose there are two instances of a value in the list; which one do we keep, the first or the second? On the one hand, since the two instances must be equivalent it doesn’t matter, but it does for writing concrete examples and deriving a solution.

For instance, you might have generated this sequence:

```

examples:
  uniq([list: 1, 2, 1, 3, 1, 2, 4, 1]) is [list: 3, 2, 4, 1]
  uniq([list: 2, 1, 3, 1, 2, 4, 1]) is [list: 3, 2, 4, 1]
  uniq([list: 1, 3, 1, 2, 4, 1]) is [list: 3, 2, 4, 1]
  uniq([list: 3, 1, 2, 4, 1]) is [list: 3, 2, 4, 1]
  uniq([list: 1, 2, 4, 1]) is [list: 2, 4, 1]
  uniq([list: 2, 4, 1]) is [list: 2, 4, 1]
  uniq([list: 4, 1]) is [list: 4, 1]
  uniq([list: 1]) is [list: 1]

```

```
    uniq([list:           ]) is [list:           ]  
end
```

However, you might have also generated sequences that began with

```
uniq([list: 1, 2, 1, 3, 1, 2, 4, 1]) is [list: 1, 2, 3, 4]
```

or

```
uniq([list: 1, 2, 1, 3, 1, 2, 4, 1]) is [list: 4, 3, 2, 1]
```

and so on. Let's work with the examples we've worked out above.

5.2.9.3 uniq: Code What is the systematic approach that gets us to this answer? When given a non-empty list, we split it into its first element and the rest of the list. Suppose we have the answer to `uniq` applied to the rest of the list. Now we can ask: is the first element in the rest of the list? If it is, then we can ignore it, since it is certain to be in the `uniq` of the rest of the list. If, however, it is not in the rest of the list, it's critical that we `link` it to the answer.

This translates into the following program. For the empty list, we return the empty list. If the list is non-empty, we check whether the first is in the rest of the list. If it is not, we include it; otherwise we can ignore it for now.

This results in the following program:

```
fun uniq-rec(l :: List<Any>) -> List<Any>:  
  cases (List) l:  
    | empty => empty  
    | link(f, r) =>  
      if r.member(f):  
        uniq-rec(r)  
      else:  
        link(f, uniq-rec(r))  
      end  
    end  
  end
```

which we've called `uniq-rec` instead of `uniq` to differentiate it from other versions of `uniq`.

Exercise

Note that we're using `.member` to check whether an element is a member of the list. Write a function `member` that consumes an element and a list, and tells us whether the element is a member of the list.

Exercise

Uniqueness checking has many practical applications. For example, one might have a list of names of people who have registered to vote in an election. To keep the voting fair, with only one vote allowed per person, we should remove duplicate names from the list.

1. Propose a set of examples for a function `rem-duplicate-voters` that takes a list of voter names and returns a list in which duplicate registrations have been removed. In developing your examples, consider real-world scenarios that you can imagine arising when identifying duplicate names. Can you identify cases in which two names might appear to be the same person, but not be? Cases in which two names might appear different but be referring to the same person?
2. What might you need to change about our current `uniq-rec` function to handle a situation like removing duplicate voters?

Responsible Computing: Context Matters When Comparing Values

The data de-duplication context in the above exercise reminds us that different contexts may call for different notions of when two data values are the same. Sometimes, we want exact matching to determine that two strings are equal. Sometimes, we need methods that

normalize data, either in simple ways like capitalization or subtler ways based on middle initials. Sometimes, we need more information (like street addresses in addition to names) in order to determine whether two items in a list should be considered “the same”.

It is easy to write programs that encode assumptions about our data that might not apply in practice. This is again a situation that can be helped by thinking about the concrete examples on which your code needs to work in context.

5.2.9.4 uniq: Reducing Computation Notice that this function has a repeated expression. Instead of writing it twice, we could call it just once and use the result in both places:

```
fun uniq-rec2(l :: List<Any>) -> List<Any>:
    cases (List) l:
        | empty => empty
        | link(f, r) =>
            ur = uniq-rec2(r)
            if r.member(f):
                ur
            else:
                link(f, ur)
            end
        end
    end
```

You might think, because we replaced two function calls with one, that we’ve reduced the amount of computation the program does. It does not! The two function calls are both in the two branches of the same conditional; therefore, for any given list element, only one or the other call to `uniq` happens. In fact, in both cases, there was one call to `uniq` before, and there is one now. So we have reduced the number of calls in the source program, but not the number that take place when the program runs. In that sense, the name of this section was intentionally misleading!

However, there is one useful reduction we can perform, which is enabled by the structure of `uniq-rec2`. We currently check whether `f` is a member of `r`, which is the list of all the remaining elements. In our example, this means that in the very second turn, we check whether 2 is a member of the list [list: 1, 3, 1, 2, 4, 1]. This is a list of six elements, including three copies of 1. We compare 2 against two copies of 1. However, we gain nothing from the second comparison. Put differently, we can think of `uniq(r)` as a “summary” of the rest of the list that is exactly as good as `r` itself for checking membership, with the advantage that it might be significantly shorter. This, of course, is exactly what `ur` represents. Therefore, we can encode this intuition as follows:

```
fun uniq-rec3(l :: List<Any>) -> List<Any>:
    cases (List) l:
        | empty => empty
        | link(f, r) =>
            ur = uniq-rec3(r)
            if ur.member(f):
                ur
            else:
                link(f, ur)
            end
        end
    end
```

Note that all that changed is that we check for membership in `ur` rather than in `r`.

Exercise

Later [Predicting Growth] we will study how to formally study how long a program takes to run. By the measure introduced in that section, does the change we just made make any difference? Be careful with your answer: it depends on how we count “the length” of the list.

Observe that if the list never contained duplicates in the first place, then it wouldn’t matter which list we check membership in—but if we knew the list didn’t contain duplicates, we wouldn’t be using `uniq` in the first place! We

will return to the issue of lists and duplicate elements in Representing Sets as Lists.

5.2.9.5 uniq: Example and Code Variations As we mentioned earlier, there are other example sequences you might have written down. Here's a very different process:

- Start with the entire given list and with the empty answer (so far).
- For each list element, check whether it's already in the answer so far. If it is, ignore it, otherwise extend the answer with it.
- When there are no more elements in the list, the answer so far is the answer for the whole list.

Notice that this solution assumes that we will be accumulating the answer as we traverse the list. Therefore, we can't even write the example with one parameter as we did before. We would argue that a natural solution asks whether we can solve the problem just from the structure of the data using the computation we are already defining, as we did above. If we cannot, then we have to resort to an accumulator. But because we can, the accumulator is unnecessary here and greatly complicates even writing down examples (give it a try!).

5.2.9.6 uniq: Why Produce a List? If you go back to the original statement of the `uniq` problem [`uniq`: Problem Setup], you'll notice it said nothing about what order the output should have; in fact, it didn't even say the output needs to be a list (and hence have an order). In that case, we should think about whether a list even makes sense for this problem. In fact, if we don't care about order and don't want duplicates (by definition of `uniq`), then there is a much simpler solution, which is to produce a set. Pyret already has sets built in, and converting the list to a set automatically takes care of duplicates. This is of course cheating from the perspective of learning how to write `uniq`, but it is worth remembering that sometimes the right data structure to produce isn't necessarily the same as the one we were given. Also, later [Representing Sets as Lists], we will see how to build sets for ourselves (at which point, `uniq` will look familiar, since it is at the heart of set-ness).

5.2.10 Monomorphic Lists and Polymorphic Types Earlier we wrote contracts like:

```
my-len :: List<Any> -> Number  
my-max :: List<Any> -> Any
```

These are unsatisfying for several reasons. Consider `my-max`. The contract suggests that any kind of element can be in the input list, but in fact that isn't true: the input `[list: 1, "two", 3]` is not valid, because we can't compare 1 with "two" or "two" with 3.

Exercise

What happens if we run `1 > "two"` or `"two" > 3`?

Rather, what we mean is a list where all the elements are of the same kind, Technically, elements that are also comparable. and the contract has not captured that. Furthermore, we don't mean that `my-max` might return any old type: if we supply it with a list of numbers, we will not get a string as the maximum element! Rather, it will only return the kind of element that is in the provided list.

In short, we mean that all elements of the list are of the same type, but they can be of any type. We call the former monomorphic: "mono" meaning one, and "morphic" meaning shape, i.e., all values have one type. But the function `my-max` itself can operate over many of these kinds of lists, so we call it polymorphic ("poly" meaning many).

Therefore, we need a better way of writing these contracts. Essentially, we want to say that there is a type variable (as opposed to regular program variable) that represents the type of element in the list. Given that type, `my-max` will return an element of that type. We write this syntactically as follows:

```
fun my-max<T>(l :: List<T>) -> T: ... end
```

The notation `<T>` says that `T` is a type variable parameter that will be used in the rest of the function (both the header and the body).

Using this notation, we can also revisit `my-len`. Its header now becomes:

```
fun my-len<T>(l :: List<T>) -> Number: ... end
```

Note that `my-len` did not actually “care” that whether all the values were of the same type or not: it never looks at the individual elements, much less at pairs of them. However, as a convention we demand that lists always be monomorphic. This is important because it enables us to process the elements of the list uniformly: if we know how to process elements of type T, then we will know how to process a `List<T>`. If the list elements can be of truly any old type, we can’t know how to process its elements.

contents ← prev up next →

5.3 Recursive Data

5.3 Recursive Data In Telling Apart Variants of Conditional Data, we used `cases` to distinguish between different forms of conditional data. We had used `cases` earlier, specifically to distinguish between empty and non-empty lists in Processing Lists. This suggests that lists are also a form of conditional data, just one that is built into Pyret. Indeed, this is the case.

To understand lists as conditional data, let's create a data definition for a new type `NumList` which contains a list of numbers (this differs from built-in lists, which work with any type of element). To avoid conflicts with Pyret's built-in `empty` value and `link` operator, we'll have `NumList` use `nl-empty` as its empty value and `nl-link` as the operator that builds new lists. Here's a partial definition:

```
data NumList:  
| nl-empty  
| nl-link( _____ )  
end
```

Do Now!

Fill in the blank in the `nl-link` condition with the corresponding field(s) and corresponding types. The blank could contain anywhere from 0 through multiple fields.

From working with lists earlier, hopefully you remembered that list constructors take two inputs: the first element of the list and a list to build on (the rest of the list). That suggests that we need two fields here:

```
data NumList:  
| nl-empty  
| nl-link(first :: _____, rest :: _____ )  
end
```

Do Now!

Fill in the types for `first` and `rest` if you haven't already.

Since we're making a list of numbers, the `first` field should contain type `Number`. What about the `rest` field? It needs to be a list of numbers, so its type should be `NumList`.

```
data NumList:  
| nl-empty  
| nl-link(first :: Number, rest :: NumList)  
end
```

Notice something interesting (and new) here: the type of the `rest` field is the same type (`NumList`) as the conditional data that we are defining. We can, quite literally, draw the arrows that show the self-referential part of the definition:

```

data NumList:
| nl-empty
| nl-link(first :: Number, rest :: NumList)
end

```

Does that actually work? Yes. Think about how we might build up a list with the numbers 2, 7, and 3 (in that order). We start with `nl-empty`, which is a valid `NumList`. We then use `nl-link` to add the numbers onto that list, as follows:

```

nl-empty
nl-link(3, nl-empty)
nl-link(7, nl-link(3, nl-empty))
nl-link(2, nl-link(7, nl-link(3, nl-empty)))

```

In each case, the `rest` argument is itself a valid `NumList`. While defining data in terms of itself might seem problematic, it works fine because in order to build actual data, we had to start with the `nl-empty` condition, which does not refer to `NumList`.

Data definitions that build on fields of the same type are called recursive data. Recursive data definitions are powerful because they permit us to create data that are unbounded or arbitrarily-sized. Given a `NumList`, there is an easy way to make a new, larger one: just use `nl-link`. So, we need to consider larger lists:

```

nl-link(1,
       nl-link(2,
               nl-link(3,
                       nl-link(4,
                               nl-link(5,
                                       nl-link(6,
                                               nl-link(7,
                                                       nl-link(8,
                                                               nl-empty))))))

```

5.3.1 Functions to Process Recursive Data Let's try to write a function `contains-3`, which returns `true` if the `NumList` contains the value 3, and `false` otherwise.

First, our header:

```

fun contains-3(nl :: NumList) -> Boolean:
    doc: "Produces true if the list contains 3, false otherwise"
end

```

Next, some tests:

```

fun contains-3(nl :: NumList) -> Boolean:
    doc: "Produces true if the list contains 3, false otherwise"
where:
    contains-3(nl-empty) is false
    contains-3(nl-link(3, nl-empty)) is true
    contains-3(nl-link(1, nl-link(3, nl-empty))) is true
    contains-3(nl-link(1, nl-link(2, nl-link(3, nl-link(4, nl-empty)))))) is true
    contains-3(nl-link(1, nl-link(2, nl-link(5, nl-link(4, nl-empty)))))) is false
end

```

As we did in Processing Fields of Variants, we will use `cases` to distinguish the variants. In addition, since we are going to have to use the fields of `nl-link` to compute a result in that case, we will list those in the initial code outline:

```
fun contains-3(nl :: NumList) -> Boolean:  
  doc: "Produces true if the list contains 3, false otherwise"  
  cases (NumList) nl:  
    | nl-empty => ...  
    | nl-link(first, rest) =>  
      ... first ...  
      ... rest ...  
  end  
end
```

Following our examples, the answer must be false in the `nl-empty` case. In the `nl-link` case, if the `first` element is 3, we've successfully answered the question. That only leaves the case where the argument is an `nl-link` and the first element does not equal 3:

```
fun contains-3(nl :: NumList) -> Boolean:  
  cases (NumList) nl:  
    | nl-empty => false  
    | nl-link(first, rest) =>  
      if first == 3:  
        true  
      else:  
        # handle rest here  
    end  
  end
```

Since we know `rest` is a `NumList` (based on the data definition), we can use a `cases` expression to work with it. This is sort of like filling in a part of the template again:

```
fun contains-3(nl :: NumList) -> Boolean:  
  cases (NumList) nl:  
    | nl-empty => false  
    | nl-link(first, rest) =>  
      if first == 3:  
        true  
      else:  
        cases (NumList) rest:  
          | nl-empty => ...  
          | nl-link(first-of-rest, rest-of-rest) =>  
            ... first-of-rest ...  
            ... rest-of-rest ...  
        end  
      end  
    end  
  end
```

If the `rest` was empty, then we haven't found 3 (just like when we checked the original argument, `nl`). If the `rest` was a `nl-link`, then we need to check if the first thing in the rest of the list is 3 or not:

```
fun contains-3(nl :: NumList) -> Boolean:  
  cases (NumList) nl:  
    | nl-empty => false  
    | nl-link(first, rest) =>  
      if first == 3:  
        true  
      else:
```

```

cases (NumList) rest:
| nl-empty => false
| nl-link(first-of-rest, rest-of-rest) =>
  if first-of-rest == 3:
    true
  else:
    # fill in here ...
  end
end
end
end

```

Since `rest-of-rest` is a `NumList`, we can fill in a `cases` for it again:

```

fun contains-3(nl :: NumList) -> Boolean:
  cases (NumList) nl:
    | nl-empty => false
    | nl-link(first, rest) =>
      if first == 3:
        true
      else:
        cases (NumList) rest:
          | nl-empty => false
          | nl-link(first-of-rest, rest-of-rest) =>
            if first-of-rest == 3:
              true
            else:
              cases (NumList) rest-of-rest:
                | nl-empty => ...
                | nl-link(first-of-rest-of-rest, rest-of-rest-of-rest) =>
                  ... first-of-rest-of-rest ...
                  ... rest-of-rest-of-rest ...
                end
              end
            end
          end
        end
      end
    end
  end
end

```

See where this is going? Not anywhere good. We can copy this `cases` expression as many times as we want, but we can never answer the question for a list that is just one element longer than the number of times we copy the code.

So what to do? We tried this approach of using another copy of `cases` based on the observation that `rest` is a `NumList`, and `cases` provides a meaningful way to break apart a `NumList`; in fact, it's what the recipe seems to lead to naturally.

Let's go back to the step where the problem began, after filling in the template with the first check for 3:

```

fun contains-3(nl :: NumList) -> Boolean:
  cases (NumList) nl:
    | nl-empty => false
    | nl-link(first, rest) =>
      if first == 3:
        true
      else:
        # what to do with rest?
      end
    end
  end

```

```
end
```

We need a way to compute whether or not the value `3` is contained in `rest`. Looking back at the data definition, we see that `rest` is a perfectly valid `NumList`, simply by the definition of `nl-link`. And we have a function (or, most of one) whose job is to figure out if a `NumList` contains `3` or not: `contains-3`. That ought to be something we can call with `rest` as an argument, and get back the value we want:

```
fun contains-3(nl :: NumList) -> Boolean:
  cases (NumList) nl:
    | nl-empty => false
    | nl-link(first, rest) =>
      if first == 3:
        true
      else:
        contains-3(rest)
      end
    end
  end
```

And lo and behold, all of the tests defined above pass. It's useful to step through what's happening when this function is called. Let's look at an example:

```
contains-3(nl-link(1, nl-link(3, nl-empty)))
```

First, we substitute the argument value in place of `nl` everywhere it appears; that's just the usual rule for function calls.

```
=> cases (NumList) nl-link(1, nl-link(3, nl-empty)):
  | nl-empty => false
  | nl-link(first, rest) =>
    if first == 3:
      true
    else:
      contains-3(rest)
    end
  end
```

Next, we find the case that matches the constructor `nl-link`, and substitute the corresponding pieces of the `nl-link` value for the `first` and `rest` identifiers.

```
=> if 1 == 3:
  true
else:
  contains-3(nl-link(3, nl-empty))
end
```

Since `1` isn't `3`, the comparison evaluates to `false`, and this whole expression evaluates to the contents of the `else` branch.

```
=> if false:
  true
else:
  contains-3(nl-link(3, nl-empty))
end

=> contains-3(nl-link(3, nl-empty))
```

This is another function call, so we substitute the value `nl-link(3, nl-empty)`, which was the `rest` field of the original input, into the body of `contains-3` this time.

```
=> cases (NumList) nl-link(3, nl-empty):
  | nl-empty => false
```

```

| nl-link(first, rest) =>
  if first == 3:
    true
  else:
    contains-3(rest)
  end
end

```

Again, we substitute into the `nl-link` branch.

```

=> if 3 == 3:
  true
else:
  contains-3(nl-empty)
end

```

This time, since 3 is 3, we take the first branch of the `if` expression, and the whole original call evaluates to `true`.

```

=> if true:
  true
else:
  contains-3(nl-empty)
end

=> true

```

An interesting feature of this trace through the evaluation is that we reached the expression `contains-3(nl-link(3, nl-empty))`, which is a normal call to `contains-3`; it could even be a test case on its own. The implementation works by doing something (checking for equality with 3) with the non-recursive parts of the datum, and combining that result with the result of the same operation (`contains-3`) on the recursive part of the datum. This idea of doing recursion with the same function on self-recursive parts of the datatype lets us extend our template to handle recursive fields.

5.3.2 A Template for Processing Recursive Data Stepping back, we have actually derived a new way to approach writing functions over recursive data. Back in Processing Lists, we had you write functions over lists by writing a sequence of related examples, using substitution across examples to derive a program that called the function on the rest of the list. Here, we are deriving that structure from the shape of the data itself.

In particular, we can develop a function over recursive data by breaking a datum into its variants (using `cases`), pulling out the fields of each variant (by listing the field names), then calling the function itself on any recursive fields (fields of the same type). For `NumList`, these steps yield the following code outline:

```

#|
fun num-list-fun(nl :: NumList) -> ???:
  cases (NumList) nl:
    | nl-empty => ...
    | nl-link(first, rest) =>
      ... first ...
      ... num-list-fun(rest) ...
  end
end
|#
```

Here, we are using a generic function name, `num-list-fun`, to illustrate that this is the outline for any function that processes a `NumList`.

We refer to this code outline as a template. Every `data` definition has a corresponding template which captures how to break a value of that definition into cases, pull out the fields, and use the same function to process any recursive fields.

Strategy: Writing a Template for Recursive Data

Given a recursive data definition, use the following steps to create the (reusable) template for that definition:

1. Create a function header (using a general-purpose placeholder name if you aren't yet writing a specific function)
2. Use `cases` to break the recursive-data input into its variants
3. In each case, list each of its fields in the answer portion of the case
4. Call the function itself on any recursive fields

The power of the template lies in its universality. If you are asked to write a specific function (such as `contains-3`) over recursive data (`NumList`), you can reproduce or copy (if you already wrote it down) the template, replace the generic function name in the template with the one for your specific function, then fill in the ellipses to finish the function. This leads to a revised description of our design recipe:

To handle recursive data, the design recipe just needs to be modified to have this extended template. When you see a recursive data definition (of which there will be many when programming in Pyret), you should naturally start thinking about what the recursive calls will return and how to combine their results with the other, non-recursive pieces of the datatype.

You have now seen two approaches to writing functions on recursive data: working out a sequence of related examples and modifying the template. Both approaches get you to the same final function. The power of the template, however, is that it scales to more complicated data definitions (where writing examples by hand would prove tedious). We will see examples of this as our data get more complex in coming chapters.

Exercise

Use the design recipe to write a function `contains-n` that takes a `NumList` and a `Number`, and returns whether that number is in the `NumList`.

Exercise

Use the design recipe to write a function `sum` that takes a `NumList`, and returns the sum of all the numbers in it. The sum of the empty list is 0.

Exercise

Use the design recipe to write a function `remove-3` that takes a `NumList`, and returns a new `NumList` with any 3's removed. The remaining elements should all be in the list in the same order they were in the input.

Exercise

Write a data definition called `NumListList` that represents a list of `NumLists`, and use the design recipe to write a function `sum-of-lists` that takes a `NumListList` and produces a `NumList` containing the sums of the sub-lists.

Exercise

Write a data definition and corresponding template for `StrList`, which captures lists of strings.

contents ← prev up next →

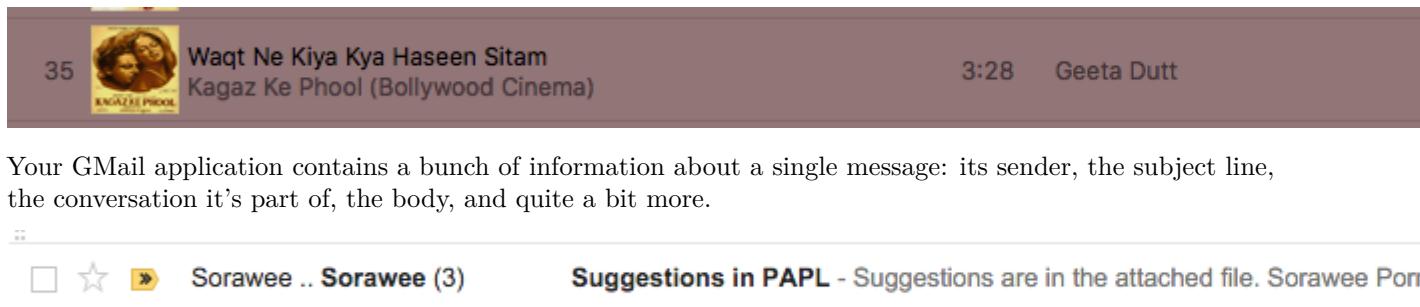
6.1 Introduction to Structured Data

6.1 Introduction to Structured Data Earlier we had our first look at types. Until now, we have only seen the types that Pyret provides us, which is an interesting but nevertheless quite limited set. Most programs we write will contain many more kinds of data.

6.1.1 Understanding the Kinds of Compound Data

6.1.1.1 A First Peek at Structured Data There are times when a datum has many attributes, or parts. We need to keep them all together, and sometimes take them apart. For instance:

- An iTunes entry contains a bunch of information about a single song: not only its name but also its singer, its length, its genre, and so on.



In examples like this, we see the need for structured data: a single datum has structure, i.e., it actually consists of many pieces. The number of pieces is fixed, but may be of different kinds (some might be numbers, some strings, some images, and different types may be mixed together in that one datum). Some might even be other structured data: for instance, a date usually has at least three parts, the day, month, and year. The parts of a structured datum are called its fields.

6.1.1.2 A First Peek at Conditional Data Then there are times when we want to represent different kinds of data under a single, collective umbrella. Here are a few examples:

- A traffic light can be in different states: red, yellow, or green. Yes, in some countries there are different or more colors and color-combinations. Collectively, they represent one thing: a new type called a traffic light state.
- A zoo consists of many kinds of animals. Collectively, they represent one thing: a new type called an animal. Some condition determines which particular kind of animal a zookeeper might be dealing with.
- A social network consists of different kinds of pages. Some pages represent individual humans, some places, some organizations, some might stand for activities, and so on. Collectively, they represent a new type: a social media page.
- A notification application may report many kinds of events. Some are for email messages (which have many fields, as we've discussed), some are for reminders (which might have a timestamp and a note), some for instant messages (similar to an email message, but without a subject), some might even be for the arrival of a package by physical mail (with a timestamp, shipper, tracking number, and delivery note). Collectively, these all represent a new type: a notification.

We call these “conditional” data because they represent an “or”: a traffic light is red or green or yellow; a social medium’s page is for a person or location or organization; and so on. Sometimes we care exactly which kind of thing

we're looking at: a driver behaves differently on different colors, and a zookeeper feeds each animal differently. At other times, we might not care: if we're just counting how many animals are in the zoo, or how many pages are on a social network, or how many unread notifications we have, their details don't matter. Therefore, there are times when we ignore the conditional and treat the datum as a member of the collective, and other times when we do care about the conditional and do different things depending on the individual datum. We will make all this concrete as we start to write programs.

6.1.2 Defining and Creating Structured and Conditional Data We have used the word “data” above, but that’s actually been a bit of a lie. As we said earlier, data are how we represent information in the computer. What we’ve been discussing above is really different kinds of information, not exactly how they are represented. But to write programs, we must wrestle concretely with representations. That’s what we will do now, i.e., actually show data representations of all this information.

6.1.2.1 Defining and Creating Structured Data Let’s start with defining structured data, such as an iTunes song record. Here’s a simplified version of the information such an app might store:

- The song’s name, which is a `String`.
- The song’s singer, which is also a `String`.
- The song’s year, which is a `Number`.

Let’s now introduce the syntax by which we can teach this to Pyret:

```
data ITunesSong: song(name, singer, year) end
```

This tells Pyret to introduce a new type of data, in this case called `ITunesSong`. We follow a convention that types always begin with a capital letter.. The way we actually make one of these data is by calling `song` with three parameters; for instance: It’s worth noting that music managers that are capable of making distinctions between, say, Dance, Electronica, and Electronic/Dance, classify two of these three songs by a single genre: “World”.

<structured-examples> ::=

```
song("La Vie en Rose", "Édith Piaf", 1945)
song("Stressed Out", "twenty one pilots", 2015)
song("Waqt Ne Kiya Kya Haseen Sitam", "Geeta Dutt", 1959)
```

Always follow a data definition with a few concrete instances of the data! This makes sure you actually do know how to make data of that form. Indeed, it’s not essential but a good habit to give names to the data we’ve defined, so that we can use them later:

```
lver = song("La Vie en Rose", "Édith Piaf", 1945)
so = song("Stressed Out", "twenty one pilots", 2015)
wnkkhs = song("Waqt Ne Kiya Kya Haseen Sitam", "Geeta Dutt", 1959)
```

In terms of the directory, structured data are no different from simple data. Each of the three definitions above creates an entry in the directory, as follows:

Directory

- `lver`
 -
 - `song("La Vie en Rose", "Édith Piaf", 1945)`
- `so`
 -
 - `song("Stressed Out", "twenty one pilots", 2015)`
- `wnkkhs`
 -
 - `song("Waqt Ne Kiya Kya Haseen Sitam", "Geeta Dutt", 1959)`

6.1.2.2 Annotations for Structured Data Recall that in [Type Annotations] we discussed annotating our functions. Well, we can annotate our data, too! In particular, we can annotate both the definition of data and their creation. For the former, consider this data definition, which makes the annotation information we'd recorded informally in text a formal part of the program:

```
data ITunesSong: song(name :: String, singer :: String, year :: Number) end
```

Similarly, we can annotate the variables bound to examples of the data. But what should we write here?

```
lver :: ___ = song("La Vie en Rose", "Édith Piaf", 1945)
```

Recall that annotations takes names of types, and the new type we've created is called `ITunesSong`. Therefore, we should write

```
lver :: ITunesSong = song("La Vie en Rose", "Édith Piaf", 1945)
```

Do Now!

What happens if we instead write this?

```
lver :: String = song("La Vie en Rose", "Édith Piaf", 1945)
```

What error do we get? How about if instead we write these?

```
lver :: song = song("La Vie en Rose", "Édith Piaf", 1945)  
lver :: 1 = song("La Vie en Rose", "Édith Piaf", 1945)
```

Make sure you familiarize yourself with the error messages that you get.

6.1.2.3 Defining and Creating Conditional Data The `data` construct in Pyret also lets us create conditional data, with a slightly different syntax. For instance, say we want to define the colors of a traffic light:

```
data TLCOLOR:  
| Red  
| Yellow  
| Green  
end
```

Conventionally, the names of the options begin in lower-case, but if they have no additional structure, we often capitalize the initial to make them look different from ordinary variables: i.e., `Red` rather than `red`. Each `|` (pronounced “stick”) introduces another option. You would make instances of traffic light colors as

```
Red  
Green  
Yellow
```

A more interesting and common example is when each condition has some structure to it; for instance:

```
data Animal:  
| boa(name :: String, length :: Number)  
| armadillo(name :: String, liveness :: Boolean)  
end
```

“In Texas, there ain’t nothin’ in the middle of the road except yellow stripes and a dead armadillo.”—Jim Hightower
We can make examples of them as you would expect:

```
b1 = boa("Ayisha", 10)  
b2 = boa("Bonito", 8)  
a1 = armadillo("Glypto", true)
```

We call the different conditions variants.

Do Now!

How would you annotate the three variable bindings?

Notice that the distinction between boas and armadillos is lost in the annotation.

```
b1 :: Animal = boa("Ayisha", 10)
b2 :: Animal = boa("Bonito", 8)
a1 :: Animal = armadillo("Glypto", true)
```

When defining a conditional datum the first stick is actually optional, but adding it makes the variants line up nicely. This helps us realize that our first example

```
data ITunesSong: song(name, singer, year) end
```

is really just the same as

```
data ITunesSong:
  | song(name, singer, year)
end
```

i.e., a conditional type with just one condition, where that one condition is structured.

6.1.3 Programming with Structured and Conditional Data So far we've learned how to create structured and conditional data, but not yet how to take them apart or write any expressions that involve them. As you might expect, we need to figure out how to

- take apart the fields of a structured datum, and
- tell apart the variants of a conditional datum.

As we'll see, Pyret also gives us a convenient way to do both together.

6.1.3.1 Extracting Fields from Structured Data Let's write a function that tells us how old a song is. First, let's think about what the function consumes (an `ITunesSong`) and produces (a `Number`). This gives us a rough skeleton for the function:

```
<song-age> ::=
fun song-age(s :: ITunesSong) -> Number:
  <song-age-body>
end
```

We know that the form of the body must be roughly:

```
<song-age-body> ::=
2016 - <get the song year>
```

We can get the song year by using Pyret's field access, which is a `.` followed by a field's name—in this case, `year`—following the variable that holds the structured datum. Thus, we get the `year` field of `s` (the parameter to `song-age`) with

```
s.year
```

So the entire function body is:

```
fun song-age(s :: ITunesSong) -> Number:
  2016 - s.year
end
```

It would be good to also record some examples (`<structured-examples>`), giving us a comprehensive definition of the function:

```
fun song-age(s :: ITunesSong) -> Number:
  2016 - s.year
where:
  song-age(lver) is 71
  song-age(so) is 1
  song-age(wnkhs) is 57
end
```

6.1.3.2 Telling Apart Variants of Conditional Data Now let's see how we tell apart variants. For this, we again use `cases`, as we saw for lists. We create one branch for each of the variants. Thus, if we wanted to compute advice for a driver based on a traffic light's state, we might write:

```
fun advice(c :: TLCOLOR) -> String:  
  cases (TLCOLOR) c:  
    | Red => "wait!"  
    | Yellow => "get ready..."  
    | Green => "go!"  
  end  
end
```

Do Now!

What happens if you leave out the `=>`?

Do Now!

What if you leave out a variant? Leave out the `Red` variant, then try both `advice(Yellow)` and `advice(Red)`.

6.1.3.3 Processing Fields of Variants In this example, the variants had no fields. But if the variant has fields, Pyret expects you to list names of variables for those fields, and will then automatically bind those variables—so you don't need to use the `.-notation` to get the field values.

To illustrate this, assume we want to get the name of any animal:

```
<animal-name> ::=  
  
fun animal-name(a :: Animal) -> String:  
  <animal-name-body>  
end
```

Because an `Animal` is conditionally defined, we know that we are likely to want a `cases` to pull it apart; furthermore, we should give names to each of the fields: Note that the names of the variables do not have to match the names of fields. Conventionally, we give longer, descriptive names to the field definitions and short names to the corresponding variables.

```
<animal-name-body> ::=  
  
cases (Animal) a:  
  | boa(n, 1) => ...  
  | armadillo(n, 1) => ...  
end
```

In both cases, we want to return the field `n`, giving us the complete function:

```
fun animal-name(a :: Animal) -> String:  
  cases (Animal) a:  
    | boa(n, 1) => n  
    | armadillo(n, 1) => n  
  end  
where:  
  animal-name(b1) is "Ayisha"  
  animal-name(b2) is "Bonito"  
  animal-name(a1) is "Glypto"  
end
```

Let's look at how Pyret would evaluate a function call like

```
animal-name(boa("Bonito", 8))
```

The argument `boa("Bonito", 8)` is a value. In the same way as we substitute simple data types like strings and numbers for parameters when we evaluate a function, we do the same thing here. After substituting, we are left with the following expression to evaluate:

```
cases (Animal) boa("Bonito", 8):
| boa(n, 1) => n
| armadillo(n, 1) => n
end
```

Next, Pyret determines which case matches the data (the first one, for `boa`, in this case). It then substitutes the field names with the corresponding components of the datum result expression for the matched case. In this case, we will substitute uses of `n` with "Bonito" and uses of `1` with 8. In this program, the entire result expression is a use of `n`, so the result of the program in this case is "Bonito".

contents ← prev up next →

6.2 Collections of Structured Data

6.2 Collections of Structured Data As we were looking at structured data [Introduction to Structured Data], we came across several situations where we have not one but many data: not one song but a playlist of them, not one animal but a zoo full of them, not one notification but several, not just one message (how we wish!) but many in our inbox, and so on. In general, then, we rarely have just a single structured datum: One notable exception: consider the configuration or preference information for a system. This might be stored in a file and updated through a user interface. Even though there is (usually) only one configuration at a time, it may have so many pieces that we won't want to clutter our program with a large number of variables; instead, we might create a structure representing the configuration, and load just one instance of it. In effect, what would have been unconnected variables now become a set of linked fields. if we know we have only one, we might just have a few separate variables representing the pieces without going to the effort of creating and taking apart a structure. In general, therefore, we want to talk about collections of structured data. Here are more examples:

- The set of messages matching a tag.
- The list of messages in a conversation.
- The set of friends of a user.

Do Now!

How are collective data different from structured data?

In structured data, we have a fixed number of possibly different kinds of values. In collective data, we have a variable number of the same kind of value. For instance, we don't say up front how many songs must be in a playlist or how many pages a user can have; but every one of them must be a song or a page. (A page may, of course, be conditionally defined, but ultimately everything in the collection is still a page.)

Observe that we've mentioned both sets and lists above. The difference between a set and a list is that a set has no order, but a list has an order. This distinction is not vital now but we will return to it later [Sets as Collective Data].

Of course, sets and lists are not the only kinds of collective data we can have. Here are some more:

- A family tree of people.
- The filesystem on your computer.
- A seating chart at a party.
- A social network of pages.

and so on. For the most part these are just as easy to program and manipulate as the earlier collective data once we have some experience, though some of them [Re-Examining Equality] can involve more subtlety.

We have already seen tables [Introduction to Tabular Data], which are a form of collective, structured data. Now we will look at a few more, and how to program them.

6.2.1 Lists as Collective Data We have already seen one example of a collection in some depth before: lists. A list is not limited to numbers or strings; it can contain any kind of value, including structured ones. For instance, using our examples from earlier [Defining and Creating Structured Data], we can make a list of songs:

```
song-list = [list: lver, so, wnkhs]
```

This is a three-element list where each element is a song:

```

check:
  song-list.length() is 3
  song-list.first is lver
end

```

Thus, what we have seen earlier about building functions over lists [Processing Lists] applies here too. To illustrate, suppose we wish to write the function `oldest-song-age`, which consumes a list of songs and produces the oldest song in the list. (There may be more than one song from the same year; the age—by our measure—of all those songs will be the same. If this happens, we just pick one of the songs from the list. Because of this, however, it would be more accurate to say “an” rather than “the” oldest song.)

Let’s work through this with examples. To keep our examples easy to write, instead of writing out the full data for the songs, we’ll refer to them just by their variable names. Clearly, the oldest song in our list is bound to `lvar`.

```

oldest-song([list: lver, so, wnkhs]) is lvar
oldest-song([list:      so, wnkhs]) is wnkhs
oldest-song([list:      wnkhs]) is wnkhs
oldest-song([list:          ]) is ???

```

What do we write in the last case? Recall that we saw this problem earlier [my-max: Examples]: there is no answer in the empty case. In fact, the computation here is remarkably similar to that of `my-max`, because it is essentially the same computation, just asking for the minimum year (which would make the song the oldest).

From our examples, we can see a solution structure echoing that of `my-max`. For the empty list, we signal an error. Otherwise, we compute the oldest song in the rest of the list, and compare its year against that of the first. Whichever has the older year is the answer.

```

fun oldest-song(sl :: List<ITunesSong>) -> ITunesSong:
  cases (List) sl:
    | empty => raise("not defined for empty song lists")
    | link(f, r) =>
      cases (List) r:
        | empty => f
        | else =>
          osr = oldest-song(r)
          if osr.year < f.year:
            osr
          else:
            f
          end
        end
      end
    end
  end
end

```

Note that there is no guarantee there will be only oldest song, and this is reflected in the possibility that `osr.year` may equal `f.year`. However, our problem statement allowed us to pick just one such song, which is what we’ve done.

Do Now!

Modify the solution above to `oldest-song-age`, which computes the age of the oldest song(s).

Haha, just kidding! You shouldn’t modify the previous solution at all! Instead, you should leave it alone—it may come in handy for other purposes—and instead build a new function to use it:

```

fun oldest-song-age(sl :: List<ITunesSong>) -> Number:
  os = oldest-song(sl)
  song-age(os)
where:
  oldest-song-age(song-list) is 71
end

```

6.2.2 Sets as Collective Data As we've already seen, for some problems we don't care about the order of inputs, nor about duplicates. Here are more examples where we don't care about order or duplicates:

- Your Web browser records which Web pages you've visited, and some Web sites use this information to color visited links differently than ones you haven't seen. This color is typically independent of how many times you have visited the page.
- During an election, a poll agent might record that you have voted, but does not need to record how many times you have voted, and does not care about the order in which people vote.

For such problems a list is a bad fit relative to a set. Here we will see how Pyret's built-in sets work. In [Several Variations on Sets] we will see how we can build sets for ourselves.

First, we can define sets just as easily as we can lists:

```
import sets as S
song-set = [S.set: lver, so, wnkhs]
```

Of course, due to the nature of the language's syntax, we have to list the elements in some order. Does it matter?

Do Now!

How can we tell whether Pyret cares about the order?

Here's the simplest way to check:

```
check:
  song-set2 = [S.set: so, wnkhs, lver]
  song-set is song-set2
end
```

If we want to be especially cautious, we can write down all the other orderings of the elements as well, and see that Pyret doesn't care.

Exercise

How many different orders are there?

Similarly for duplicates:

```
check:
  song-set3 = [S.set: lver, so, wnkhs, so, so, lver, so]
  song-set is song-set3
  song-set3.size() is 3
end
```

We can again try several different kinds of duplication and confirm that sets ignore them.

6.2.2.1 Picking Elements from Sets This lack of an ordering, however, poses a problem. With lists, it was meaningful to talk about the "first" and corresponding "rest". By definition, with sets there is not "first" element. In fact, Pyret does not even offer fields similar to `first` and `rest`. In its place is something a little more accurate but complex.

The `.pick` method returns a random element of a set. It produces a value of type `Pick` (which we get with `include pick`). When we pick an element, there are two possibilities. One is that the set is empty (analogous to a list being empty), which gives us a `pick-none` value. The other option is called `pick-some`, which gives us an actual member of the set.

The `pick-some` variant of `Pick` has two fields, not one. To understand why takes a moment's work. Let's explore it by choosing an element of a set:

```
fun an-elt(s :: S.Set):
  cases (Pick) s.pick():
    | pick-none => raise("empty set")
    | pick-some(e, r) => e
  end
```

```
end
```

(Notice that we aren't using the `r` field in the `pick-some` case.)

Do Now!

Can you guess why we didn't write examples for `an-elt`?

Do Now!

Run `an-elt(song-set)`. What element do you get?

Run it again. Run it five more times.

Do you get the same element every time?

No you don't! Well, actually, it's impossible to be certain you don't. There is a very, very small likelihood you get the exact same element on every one of six runs. If it happens to you, keep running it more times! Pyret is designed to not always return the same element when picking from a set. This is on purpose: it's to drive home the random nature of choosing from a set, and to prevent your program from accidentally depending on a particular order that Pyret might use.

Do Now!

Given that `an-elt` does not return a predictable element, what (if any) tests can we write for it?

Observe that though we can't predict which element `an-elt` will produce, we do know it will produce an element of the set. Therefore, what we can write are tests that ensure the resulting element is a member of the set—though in this case, that would not be particularly surprising.

6.2.2.2 Computing with Sets Once we have picked an element from a set, it's often useful to obtain the set consisting of the remaining elements. We have already seen that choosing the first field of a `pick-some` is similar to taking the “first” of a set. We therefore want a way to get the “rest” of the set. However, we want the rest to what we obtain after excluding this particular “first”. That's what the second field of a `pick-some` is: what's left of the set.

Given this, we can write functions over sets that look roughly analogous to functions over lists. For instance, suppose we want to compute the size of a set. The function looks similar to `my-len` [Some Example Exercises]:

```
fun my-set-size(shadow s :: S.Set) -> Number:  
  cases (Pick) s.pick():  
    | pick-none => 0  
    | pick-some(e, r) =>  
      1 + my-set-size(r)  
  end  
end
```

Though the process of deriving this is similar to that we used for `my-len`, the random nature of picking elements makes it harder to write examples that the actual function's behavior will match.

6.2.3 Combining Structured and Collective Data As the above examples illustrate, a program's data organization will often involve multiple kinds of compound data, often deeply intertwined. Let's first think of these in pairs.

Exercise

Come up with examples that combine:

- structured and conditional data,
- structured and collective data, and
- conditional and collective data.

You've actually seen examples of each of these above. Identify them.

Finally, we might even have all three at once. For instance, a filesystem is usually a list (collective) of files and folders (conditional) where each file has several attributes (structured). Similarly, a social network has a set of pages (collective) where each page is for a person, organization, or other thing (conditional), and each page has several attributes (structured). Therefore, as you can see, combinations of these arise naturally in all kinds of applications that we deal with on a daily basis.

Exercise

Take three of your favorite Web sites or apps. Identify the kinds of data they present. Classify these as structured, conditional, and collective. How do they combine these data?

6.2.4 Data Design Problem: Representing Quizzes Now that you can make collections of structured data, you can approach creating the data and programs for fairly sophisticated applications. Let's try out a data-design problem, where we will focus just on creating the data definition, but not on writing the actual functions.

Problem Statement: You've been hired to help create software for giving quizzes to students. The software will show the student a question, read in the student's answer, compare the student's answer to the expected answer (sort of like a Pyret example!), and produce the percentage of questions that the student got right.

Your task is to create a data definition for capturing quizzes and expected answers. Don't worry about representing the student responses.

Do Now!

Propose an initial data structure for quizzes. Start by identifying the pieces you might need and trying to write some sample questions.

We might imagine asking a quiz question like "what is $3 + 4$?". We would expect the student to answer 7. What would capture this? A piece of structured data with two fields like the following:

```
data Question:  
    basic-ques(text :: String, expect :: ???)  
end
```

What's a good type for the expected answer? This specific problem has a numeric answer, but other questions might have other types of answers. `Any` is therefore an appropriate type for the answer.

We would also need a list of `Question` to form an entire quiz.

Sometimes, quiz software allows students to ask for hints.

Do Now!

Assume we wanted to have some (but not all) questions with hints, which would be text that a student could request for help with a problem. Modify the current data definition to capture quizzes in which some questions have hints and some do not.

A quiz should still be a list of questions, but the `Question` data definition needs another variant in order to handle questions with hints. The following would work:

```
data Question:  
    | basic-ques(text :: String, expect :: Any)  
    | hint-ques(text :: String, expect :: Any, hint :: String)  
end
```

A quiz is a List<Question>

We could imagine extending this example to introduce dependencies between questions (such as one problem building on the skills of another), multiple choice questions, checkbox questions, and so on.

Responsible Computing: Consider the Process Being Displaced

Many companies have tried to improve education through software systems that automate tasks otherwise done by teachers. There are systems that show students a video, then give them quizzes (akin to what you just developed) to check what they have learned. A more

extreme version interleaves videos and quizzes, thus teaching entire courses at scale, without the need for teacher intervention.

Massively-online courses (MOOCs) are a style of course that makes heavy use to computer automation, to enable reaching many more students without needing more teachers. Proponents of MOOCs and related educational technology tools have promised game-changing impacts of such tools, promising to extend quality education to students around the world who otherwise might lack access to quality teachers. Technology investors (and indeed some universities) dove in behind these technologies, hoping for an educational revolution at scale.

Unfortunately, research and evaluation have shown that replacing education with automated systems, even ones with sophisticated features based on data analysis and predictions that identify skills that students haven't quite mastered, doesn't lead to the promised gains in learning. Why? It turns out that teaching is about more than choosing questions, gathering student work, and giving grades. Teachers provide encouragement, reassurance, and an understanding of an individual students' situation. Today's computational systems don't do this. The generally-accepted wisdom around these tools (backed by three prior decades of research) is that they are best used to complement direct instruction by a human teacher. In such a setting, some tools have resulted in solid performance gains on the parts of students.

The social-responsibility takeaway here is that you need to consider all the features of the system you might be trying to replace with a computational approach. Algorithmic quiz-taking tools have genuine value in some specific context, but they aren't a replacement for all of teaching. A failure to understand the many aspects of teaching, and which ones do and do not make it effective for educating students, could have avoided a lot of inaccurate hype about the promise of algorithmic instruction.

contents ← prev up next →

7.1 Trees

7.1 Trees

7.1.1 Data Design Problem – Ancestry Data Imagine that we wanted to manage ancestry information for purposes of a medical research study. Specifically, we want to record people's birthyear, eye colors, and genetic parents. Here's a sample table of such data, with one row for each person:

```
ancestors = table: name, birthyear, eyecolor, female-parent, male-parent
  row: "Anna", 1997, "blue", "Susan", "Charlie"
  row: "Susan", 1971, "blue", "Ellen", "Bill"
  row: "Charlie", 1972, "green", "", ""
  row: "Ellen", 1945, "brown", "Laura", "John"
  row: "John", 1922, "brown", "", "Robert"
  row: "Laura", 1922, "brown", "", ""
  row: "Robert", 1895, "blue", "", ""
end
```

For our research, we want to be able to answer questions such as the following:

- Who are the genetic grandparents of a specific person?
- How frequent is each eye color?
- Is one specific person an ancestor of another specific person?
- How many generations do we have information for?
- Does one's eye color correlate with the ages of their genetic parents when they were born?

Let's start with the first question:

Do Now!

How would you compute a list of the known grandparents for a given person? For purposes of this chapter, you may assume that each person has a unique name (while this isn't realistic in practice, it will simplify our computations for the time being; we will revisit it later in the chapter).

(Hint: Make a task plan. Does it suggest any particular helper functions?)

Our task plan has two key steps: find the names of the genetic parents of the named person, then find the names of the parents of each of those people. Both steps share the need to compute the known parents from a name, so we should create a helper function for that (we'll call it `parents-of`). Since this sounds like a routine table program, we can use it for a bit of review:

7.1.1.1 Computing Genetic Parents from an Ancestry Table How do we compute a list of someone's genetic parents? Let's sketch a task plan for that:

- filter the table to find the person
- extract the name of the female parent
- extract the name of the male parent
- make a list of those names

These are tasks we have seen before, so we can translate this plan directly into code:

```
fun parents-of(t :: Table, who :: String) -> List<String>:
    doc: "Return list of names of known parents of given name"
    matches = filter-with(t, lam(r): r["name"] == who end)
    if matches.length() > 0:
        person-row = matches.row-n(0)
        [list:
            person-row["female-parent"],
            person-row["male-parent"]]
    else:
        empty
    end
where:
    parents-of(ancestors, "Anna")
        is [list: "Susan", "Charlie"]
    parents-of(ancestors, "Kathi") is empty
end
```

Do Now!

Are you satisfied with this program? With the examples included in the `where` block? Write down any critiques you have.

There are arguably some issues here. How many of these did you catch?

- The examples are weak: none of them consider people for whom we are missing information on at least one parent.
- The list of names returned in the case of an unknown parent includes the empty string, which isn't actually a name. This could cause problems if we use this list of names in a subsequent computation (such as to compute the names of someone's grandparents).
- If empty strings are not part of the output list, then we'd get the same result from asking for the parents of "`Robert`" (who is in the table) as for "`Kathi`" (who is not). These are fundamentally different cases, which arguably demand different outputs so we can tell them apart.

To fix these problems, we need to remove the empty strings from the produced list of parents and return something other than the `empty` list when a name is not in the table. Since the output of this function is a list of strings, it's hard to see what to return that couldn't be confused for a valid list of names. Our solution for now is to have Pyret throw an error (like the ones you get when Pyret is not able to finish running your program). Here's a solution that handles both problems:

```
fun parents-of(t :: Table, who :: String) -> List<String>:
    doc: "Return list of names of known parents of given name"
    matches = filter-with(t, lam(r): r["name"] == who end)
    if matches.length() > 0:
        person-row = matches.row-n(0)
        names =
            [list: person-row["female-parent"],
             person-row["male-parent"]]
        L.filter(lam(n): not(n == "") end, names)
    else:
        raise("No such person " + who)
    end
where:
    parents-of(ancestors, "Anna") is [list: "Susan", "Charlie"]
    parents-of(ancestors, "John") is [list: "Robert"]
    parents-of(ancestors, "Robert") is empty
    parents-of(ancestors, "Kathi") raises "No such person"
end
```

The `raise` construct tells Pyret to halt the program and produce an error message. The error message does not have to match the expected output type of the program. If you run this function with a name that is not in the table, you'll see an error appear in the interactions pane, with no result returned.

Within the `where` block, we see how to check whether an expression will yield an error: instead of using `is` to check the equality of values, we use `raises` to check whether the provided string is a sub-string of the actual error produced by the program.

7.1.1.2 Computing Grandparents from an Ancestry Table Once we have the `parents-of` function, we should be able to compute the grandparents by computing parents of parents, as follows:

```
fun grandparents-of(anc-table: Table, person: String) -> List[String]:
    doc: "compute list of known grandparents in the table"
    # glue together lists of mother's parents and father's parents
    plist = parents-of(anc-table, person) # gives a list of two names
    parents-of(anc-table, plist.first) +
        parents-of(anc-table, plist.rest.first)
where:
    grandparents("Anna") is [list: "Laura", "John"]
    grandparents("Laura") is [list:]
    grandparents("Kathi") is [list:]
end
```

Do Now!

Look back at our sample ancestry tree: for which people would this correctly compute the list of grandparents?

This `grandparents-of` code works fine for someone who has both parents in the table. For someone without two parents, however, the `plist` will have fewer than two names, so the expression `plist.rest.first` (if not `plist.first`) will yield an error.

Here's a version that checks the number of parents before computing the set of grandparents:

```
fun grandparents-of(anc-table :: Table, name :: String) -> List<String>:
    doc: "compute list of known grandparents in the table"
    # glue together lists of mother's parents and father's parents
    plist = parents-of(anc-table, name) # gives a list of two names
    if plist.length == 2:
        parents-of(anc-table, plist.first) + parents-of(anc-table, plist.rest.first)
    else if plist.length == 1:
        parents-of(anc-table, plist.first)
    else:
        empty
    end
end
```

What if we now wanted to gather up all of someone's ancestors? Since we don't know how many generations there are, we'd need to use recursion. This approach would also be expensive, since we'd end up filtering over the table over and over, which checks every row of the table in each use of `filter`.

Look back at the ancestry tree picture. We don't do any complicated filtering there – we just follow the line in the picture immediately from a person to their mother or father. Can we get that idea in code instead? Yes, through datatypes.

7.1.1.3 Creating a Datatype for Ancestor Trees For this approach, we want to create a datatype for Ancestor Trees that has a variant (constructor) for setting up a person. Look back at our picture – what information makes up a person? Their name, their mother, and their father (along with birthyear and eyecolor, which aren't shown in the picture). This suggests the following datatype, which basically turns a row into a person value:

```
data AncTree:
    | person(
```

```

    name :: String,
    birthyear :: Number,
    eye :: String,
    mother :: _____,
    father :: _____
)

```

end

For example, Anna's row might look like:

```
anna-row = person("Anna", 1997, "blue", ???, ???)
```

What type do we put in the blanks? A quick brainstorm yields several ideas:

- person
- List<person>
- some new datatype
- AncTree
- String

Which should it be?

If we use a **String**, we're back to the table row, and we don't end up with a way to easily get from one person to another. We should therefore make this an **AncTree**.

```

data AncTree:
| person(
  name :: String,
  birthyear :: Number,
  eye :: String,
  mother :: AncTree,
  father :: AncTree
)

```

end

Do Now!

Write the **AncTree** starting from Anna using this definition.

Did you get stuck? What do we do when we run out of known people? To handle that, we must add an option in the **AncTree** definition to capture people for whom we don't know anything.

```

data AncTree:
| noInfo
| person(
  name :: String,
  birthyear :: Number,
  eye :: String,
  mother :: AncTree,
  father :: AncTree
)

```

end

Here's Anna's tree written in this datatype:

```

anna-tree =
person("Anna", 1997, "blue",
person("Susan", 1971, "blue",
person("Ellen", 1945, "brown",
person("Laura", 1920, "blue", noInfo, noInfo),
person("John", 1920, "green",

```

```

noInfo,
person("Robert", 1893, "brown", noInfo, noInfo)),
person("Bill", 1946, "blue", noInfo, noInfo)),
person("Charlie", 1972, "green", noInfo, noInfo))

```

We could also have named each person data individually.

```

robert-tree = person("Robert", 1893, "brown", noInfo, noInfo)
laura-tree = person("Laura", 1920, "blue", noInfo, noInfo)
john-tree = person("John", 1920, "green", noInfo, robert-tree)
ellen-tree = person("Ellen", 1945, "brown", laura-tree, john-tree)
bill-tree = person("Bill", 1946, "blue", noInfo, noInfo)
susan-tree = person("Susan", 1971, "blue", ellen-tree, bill-tree)
charlie-tree = person("Charlie", 1972, "green", noInfo, noInfo)
anna-tree2 = person("Anna", 1997, "blue", susan-tree, charlie-tree)

```

The latter gives you pieces of the tree to use as other examples, but loses the structure that is visible in the indentation of the first version. You could get to pieces of the first version by digging into the data, such as writing `anna-tree.mother.mother` to get to the tree starting from "Ellen".

Here's the `parents-of` function written against `AncTree`:

```

fun parents-of-tree(tr :: AncTree) -> List<String>:
  cases (AncTree) tr:
    | noInfo => empty
    | person(n, y, e, m, f) => [list: m.name, f.name]
      # person bit more complicated if parent is missing
  end
end

```

7.1.2 Programs to Process Ancestor Trees How would we write a function to determine whether anyone in the tree had a particular name? To be clear, we are trying to fill in the following code:

```

fun in-tree(at :: AncTree, name :: String) -> Boolean:
  doc: "determine whether name is in the tree"
  ...

```

How do we get started? Add some examples, remembering to check both cases of the `AncTree` definition:

```

fun in-tree(at :: AncTree, name :: String) -> Boolean:
  doc: "determine whether name is in the tree"
  ...
where:
  in-tree(anna-tree, "Anna") is true
  in-tree(anna-tree, "Ellen") is true
  in-tree(ellen-tree, "Anna") is false
  in-tree(noInfo, "Ellen") is false
end

```

What next? When we were working on lists, we talked about the template, a skeleton of code that we knew we could write based on the structure of the data. The template names the pieces of each kind of data, and makes recursive calls on pieces that have the same type. Here's the template over the `AncTree` filled in:

```

fun in-tree(at :: AncTree, name :: String) -> Boolean:
  doc: "determine whether name is in the tree"
  cases (AncTree) at:      # comes from AncTree being data with cases
    | noInfo => ...
    | person(n, y, e, m, f) => ... in-tree(m, name) ... in-tree(f, name)
  end
where:
  in-tree(anna-tree, "Anna") is true

```

```

in-tree(anna-tree, "Ellen") is true
in-tree(ellen-tree, "Anna") is false
in-tree(noInfo, "Ellen") is false
end

```

To finish the code, we need to think about how to fill in the ellipses.

- When the tree is `noInfo`, it has no more people, so the answer should be false (as worked out in the examples).
- When the tree is a person, there are three possibilities: we could be at a person with the name we're looking for, or the name could be in the mother's tree, or the name could be in the father's tree.

We know how to check whether the person's name matches the one we are looking for. The recursive calls already ask about the name being in the mother's tree or father's tree. We just need to combine those pieces into one Boolean answer. Since there are three possibilities, we should combine them with `or`

Here's the final code:

```

fun in-tree(at :: AncTree, name :: String) -> Boolean:
    doc: "determine whether name is in the tree"
    cases (AncTree) at:      # comes from AncTree being data with cases
        | noInfo => false
        | person(n, y, e, m, f) => (name == n) or in-tree(m, name) or in-tree(f, name)
            # n is the same as at.name
            # m is the same as at.mother
    end
where:
    in-tree(anna-tree, "Anna") is true
    in-tree(anna-tree, "Ellen") is true
    in-tree(ellen-tree, "Anna") is false
    in-tree(noInfo, "Ellen") is false
end

```

7.1.3 Summarizing How to Approach Tree Problems We design tree programs using the same design recipe that we covered on lists:

Strategy: Writing a Program Over Trees

- Write the datatype for your tree, including a base/leaf case
- Write examples of your trees for use in testing
- Write the function name, parameters, and types (the `fun` line)
- Write `where` checks for your code
- Write the template, including the cases and recursive calls. Here's the template again for an ancestor tree, for an arbitrary function called `treeF`:

```

fun treeF(name :: String, t :: AncTree) -> Boolean:
    cases (AncTree) anct:
        | unknown => ...
        | person(n, y, e, m, f) =>
            ... treeF(name, m) ... treeF(name, f)
    end
end

```

- Fill in the template with details specific to the problem
- Test your code using your examples

7.1.4 Study Questions

- Think of writing in-tree on a table (using filter-by) vs writing it on a tree. How many times might each approach compare the name being sought against a name in the table/tree?
- Why do we need to use a recursive function to process the tree?
- In what order will we check the names in the tree version?

For practice, try problems such as

- How many blue-eyed people are in the tree?
- How many people are in the tree?
- How many generations are in the tree?
- How many people have a given name in a tree?
- How many people have names starting with "A"?
- ... and so on

contents ← prev up next →

8.1 Functions as Data

8.1 Functions as Data It's interesting to consider how expressive the little programming we've learned so far can be. To illustrate this, we'll work through a few exercises of interesting concepts we can express using just functions as values. We'll write two quite different things, then show how they converge nicely.

8.1.1 A Little Calculus If you've studied the differential calculus, you've come across curious syntactic statements such as this:

```
\begin{equation*}\frac{d}{dx} x^2 = 2x\end{equation*}
```

Let's unpack what this means: the $\frac{d}{dx}$, the x^2 , and the $2x$.

First, let's take on the two expressions; we'll discuss one, and the discussion will cover the other as well. The correct response to "what does x^2 mean?" is, of course, an error: it doesn't mean anything, because x is an unbound identifier.

So what is it intended to mean? The intent, clearly, is to represent the function that squares its input, just as $2x$ is meant to be the function that doubles its input. We have nicer ways of writing those:

```
fun sq(x :: Number) -> Number: x * x end  
fun dbl(x :: Number) -> Number: 2 * x end
```

and what we're really trying to say is that the $\frac{d}{dx}$ (whatever that is) of `sq` is `dbl`. We're assuming functions of arity one in the variable that is changing.

So now let's unpack $\frac{d}{dx}$, starting with its type. As the above example illustrates, $\frac{d}{dx}$ is really a function from functions to functions. That is, we can write its type as follows:

```
d-dx :: ((Number -> Number) -> (Number -> Number))
```

(This type might explain why your calculus course never explained this operation this way—though it's not clear that obscuring its true meaning is any better for your understanding.)

Let us now implement `d-dx`. We'll implement numerical differentiation, though in principle we could also implement symbolic differentiation—using rules you learned, e.g., given a polynomial, multiply by the exponent and reduce the exponent by one—with a representation of expressions (a problem that will be covered in more detail in a future release).

In general, numeric differentiation of a function at a point yields the value of the derivative at that point. We have a handy formula for it: the derivative of f at x is

```
\begin{equation*}\frac{f(x + \epsilon) - f(x)}{\epsilon}\end{equation*}
```

as ϵ goes to zero in the limit. For now we'll give the infinitesimal a small but fixed value, and later [Combining Forces: Streams of Derivatives] see how we can improve on this.

```
epsilon = 0.00001
```

We can now translate the above formula into a function:

```
d-dx-at :: (Number -> Number), Number -> Number
```

```
fun d-dx-at(f, x):  
  (f(x + epsilon) - f(x)) / epsilon  
end
```

And sure enough, we can check and make sure it works as expected:

```
check:  
  d-dx-at(sq, 10) is-roughly dbl(10)  
end
```

Confession: We chose the value of `epsilon` so that the default tolerance `is-roughly` works for this example.

However, there is something unsatisfying about this. The function we've written clearly does not have the type we described earlier! What we wanted was an operation that takes just a function, and represents the platonic notion of differentiation; but we've been forced, by the nature of numeric differentiation, to describe the derivative at a point. We might instead like to write something like this:

```
fun d-dx(f):  
  (f(x + epsilon) - f(x)) / epsilon  
end
```

Do Now!

What's the problem with the above definition?

If you didn't notice, Pyret will soon tell you: `x` isn't bound. Indeed, what is `x`? It's the point at which we're trying to compute the numeric derivative. That is, `d-dx` needs to return not a number but a function (as the type indicates) that will consume this `x`: "Lambdas are relegated to relative obscurity until Java makes them popular by not having them."—James Iry, A Brief, Incomplete, and Mostly Wrong History of Programming Languages

```
fun d-dx(f):  
  lam(x):  
    (f(x + epsilon) - f(x)) / epsilon  
  end  
end
```

If we want to be a little more explicit we can annotate the inner function:

```
fun d-dx(f):  
  lam(x :: Number) -> Number:  
    (f(x + epsilon) - f(x)) / epsilon  
  end  
end
```

This is a special case of a concept useful in many programming contexts, which we explore in more detail elsewhere: Staging.

Sure enough, this definition now works. We can, for instance, test it as follows (note the use of `num-floor` to avoid numeric precision issues from making our tests appear to fail):

```
d-dx-sq = d-dx(sq)  
  
check:  
  ins = [list: 0, 1, 10, 100]  
  for map(n from ins):  
    num-floor(d-dx-sq(n))  
  end  
  is  
  for map(n from ins):  
    num-floor(dbl(n))  
  end  
end
```

Now we can return to the original example that launched this investigation: what the sloppy and mysterious notation of math is really trying to say is,

```
d-dx(lam(x): x * x end) = lam(x): 2 * x end
```

or, in the notation of A Notation for Functions,

$$\begin{aligned} \text{\&begin\{equation*}\}\{\frac{d}{dx}\}} [x \rightarrow x^2] = [x \rightarrow 2x]\end{aligned}$$

Pity math textbooks for not wanting to tell us the truth!

8.1.2 A Helpful Shorthand for Anonymous Functions Pyret offers a shorter syntax for writing anonymous functions. Though, stylistically, we generally avoid it so that our programs don't become a jumble of special characters, sometimes it's particularly convenient, as we will see below. This syntax is

```
{(a): b}
```

where **a** is zero or more arguments and **b** is the body. For instance, we can write `lam(x): x * x end` as

```
{(x): x * x}
```

where we can see the benefit of brevity. In particular, note that there is no need for `end`, because the braces take the place of showing where the expression begins and ends. Similarly, we could have written `d-dx` as

```
fun d-dx-short(f):
  {(x): (f(x + epsilon) - f(x)) / epsilon}
end
```

but many readers would say this makes the function harder to read, because the prominent `lam` makes clear that `d-dx` returns an (anonymous) function, whereas this syntax obscures it. Therefore, we will usually only use this shorthand syntax for “one-liners”.

8.1.3 Streams From Functions People typically think of a function as serving one purpose: to parameterize an expression. While that is both true and the most common use of a function, it does not justify having a function of no arguments, because that clearly parameterizes over nothing at all. Yet functions of no argument also have a use, because functions actually serve two purposes: to parameterize, and to suspend evaluation of the body until the function is applied. In fact, these two uses are orthogonal, in that one can employ one feature without the other. Below, we will focus on delay without abstraction (the other shows up in other computer science settings).

Let's consider the humble list. A list can be only finitely long. However, there are many lists (or sequences) in nature that have no natural upper bound: from mathematical objects (the sequence of natural numbers) to natural ones (the sequence of hits to a Web site). Rather than try to squeeze these unbounded lists into bounded ones, let's look at how we might represent and program over these unbounded lists.

First, let's write a program to compute the sequence of natural numbers:

```
fun nats-from(n):
  link(n, nats-from(n + 1))
end
```

Do Now!

Does this program have a problem?

While this represents our intent, it doesn't work: running it—e.g., `nats-from(0)`—creates an infinite loop evaluating `nats-from` for every subsequent natural number. In other words, we want to write something very like the above, but that doesn't recur until we want it to, i.e., on demand. In other words, we want the rest of the list to be lazy.

This is where our insight into functions comes in. A function, as we have just noted, delays evaluation of its body until it is applied. Therefore, a function would, in principle, defer the invocation of `nats-from(n + 1)` until it's needed.

Except, this creates a type problem: the second argument to `link` needs to be a list, and cannot be a function. Indeed, because it must be a list, and every value that has been constructed must be finite, every list is finite and eventually terminates in `empty`. Therefore, we need a new data structure to represent the links in these lazy lists (also known as streams):

```
<stream-type-def> ::=
```

```

data Stream<T>:
| lz-link(h :: T, t :: ( -> Stream<T>))
end

```

where the annotation `(-> Stream<T>)` means a function from no arguments (hence the lack of anything before `->`), also known as a thunk. Note that the way we have defined streams they must be infinite, since we have provided no way to terminate them.

Let's construct the simplest example we can, a stream of constant values:

```
ones = lz-link(1, lam(): ones end)
```

Pyret will actually complain about this definition. Note that the list equivalent of this also will not work:

```
ones = link(1, ones)
```

because `ones` is not defined at the point of definition, so when Pyret evaluates `link(1, ones)`, it complains that `ones` is not defined. However, it is being overly conservative with our former definition: the use of `ones` is “under a `lam`”, and hence won't be needed until after the definition of `ones` is done, at which point `ones` will be defined. We can indicate this to Pyret by using the keyword `rec`:

```
rec ones = lz-link(1, lam(): ones end)
```

Note that in Pyret, every `fun` implicitly has a `rec` beneath it, which is why we can create recursive functions with aplomb.

Exercise

Earlier we said that we can't write

```
ones = link(1, ones)
```

What if we tried to write

```
rec ones = link(1, ones)
```

instead? Does this work and, if so, what value is `ones` bound to? If it doesn't work, does it fail to work for the same reason as the definition without the `rec`?

Henceforth, we will use the shorthand [A Helpful Shorthand for Anonymous Functions] instead. Therefore, we can rewrite the above definition as:

```
rec ones = lz-link(1, {(): ones})
```

Notice that `{(): ...}` defines an anonymous function of no arguments. You can't leave out the `()`! If you do, Pyret will get confused about what your program means.

Because functions are automatically recursive, when we write a function to create a stream, we don't need to use `rec`. Consider this example:

```

fun nats-from(n :: Number):
  lz-link(n, {(): nats-from(n + 1)})
end

```

with which we can define the natural numbers:

```
nats = nats-from(0)
```

Note that the definition of `nats` is not recursive itself—the recursion is inside `nats-from`—so we don't need to use `rec` to define `nats`.

Do Now!

Earlier, we said that every list is finite and hence eventually terminates. How does this remark apply to streams, such as the definition of `ones` or `nats` above?

The description of `ones` is still a finite one; it simply represents the potential for an infinite number of values. Note that:

1. A similar reasoning doesn't apply to lists because the rest of the list has already been constructed; in contrast, placing a function there creates the potential for a potentially unbounded amount of computation to still be forthcoming.
2. That said, even with streams, in any given computation, we will create only a finite prefix of the stream. However, we don't have to prematurely decide how many; each client and use is welcome to extract less or more, as needed.

Now we've created multiple streams, but we still don't have an easy way to "see" one. First we'll define the traditional list-like selectors. Getting the first element works exactly as with lists:

```
fun lz-first<T>(s :: Stream<T>) -> T: s.h end
```

In contrast, when trying to access the rest of the stream, all we get out of the data structure is a thunk. To access the actual rest, we need to force the thunk, which of course means applying it to no arguments:

```
fun lz-rest<T>(s :: Stream<T>) -> Stream<T>: s.t() end
```

This is useful for examining individual values of the stream. It is also useful to extract a finite prefix of it (of a given size) as a (regular) list, which would be especially handy for testing. Let's write that function:

```
fun take<T>(n :: Number, s :: Stream<T>) -> List<T>:
  if n == 0:
    empty
  else:
    link(lz-first(s), take(n - 1, lz-rest(s)))
  end
end
```

If you pay close attention, you'll find that this body is not defined by cases over the structure of the (stream) input—instead, it's defined by the cases of the definition of a natural number (zero or a successor). We'll return to this below (<lz-map2-def>).

Now that we have this, we can use it for testing. Note that usually we use our data to test our functions; here, we're using this function to test our data:

```
check:
  take(10, ones) is map(lam(_): 1 end, range(0, 10))
  take(10, nats) is range(0, 10)
  take(10, nats-from(1)) is map(_ + 1, range(0, 10))
end
```

The notation $(_ + 1)$ defines a Pyret function of one argument that adds 1 to the given argument.

Let's define one more function: the equivalent of `map` over streams. For reasons that will soon become obvious, we'll define a version that takes two lists and applies the first argument to them pointwise:

```
<lz-map2-def> ::=

fun lz-map2<A, B, C>(
  f :: (A, B -> C),
  s1 :: Stream<A>,
  s2 :: Stream<B>) -> Stream<C>:
  lz-link(
    f(lz-first(s1), lz-first(s2)),
    {(): lz-map2(f, lz-rest(s1), lz-rest(s2))})
end
```

Now we can see our earlier remark about the structure of the function driven home especially clearly. Whereas a traditional `map` over lists would have two cases, here we have only one case because the data definition (<stream-type-def>) has only one case! What is the consequence of this? In a traditional `map`, one case looks like the above, but the other case corresponds to the `empty` input, for which it produces the same output. Here, because the stream never terminates, mapping over it doesn't either, and the structure of the function reflects this. This raises a much

subtler problem: if the function's body doesn't have base- and inductive-cases, how can we perform an inductive proof over it? The short answer is we can't: we must instead use coinduction.

Why did we define `lz-map2` instead of `lz-map`? Because it enables us to write the following:

```
rec fibs =
  lz-link(0,
    {(): lz-link(1,
      {(): lz-map2({{a :: Number, b :: Number}: a + b},
        fibs,
        lz-rest(fibs))}))}
```

from which, of course, we can extract as many Fibonacci numbers as we want!

```
check:
  take(10, fibs) is [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
end
```

Exercise

Define the equivalent of `map` and `filter` for streams.

Streams and, more generally, infinite data structures that unfold on demand are extremely valuable in programming. Consider, for instance, the possible moves in a game. In some games, this can be infinite; even if it is finite, for interesting games the combinatorics mean that the tree is too large to feasibly store in memory. Therefore, the programmer of the computer's intelligence must unfold the game tree on demand. Programming it by using the encoding we have described above means the program describes the entire tree, lazily, and the tree unfolds automatically on demand, relieving the programmer of the burden of implementing such a strategy.

In some languages, such as Haskell, lazy evaluation is built in by default. In such a language, there is no need to use thunks. However, lazy evaluation places other burdens on the language, which you can learn about in a programming-languages class.

8.1.4 Combining Forces: Streams of Derivatives When we defined `d-dx`, we set `epsilon` to an arbitrary, high value. We could instead think of `epsilon` as itself a stream that produces successively finer values; then, for instance, when the difference in the value of the derivative becomes small enough, we can decide we have a sufficient approximation to the derivative.

The first step is, therefore, to make `epsilon` some kind of parameter rather than a global constant. That leaves open what kind of parameter it should be (number or stream?) as well as when it should be supplied.

It makes most sense to consume this parameter after we have decided what function we want to differentiate and at what value we want its derivative; after all, the stream of `epsilon` values may depend on both. Thus, we get:

```
fun d-dx(f :: (Number -> Number)) ->
  (Number -> (Number -> Number)):
  lam(x :: Number) -> (Number -> Number):
  lam(epsilon :: Number) -> Number:
    (f(x + epsilon) - f(x)) / epsilon
  end
end
end
```

with which we can return to our `square` example:

```
d-dx-square = d-dx(square)
```

Note that at this point we have simply redefined `d-dx` without any reference to streams: we have merely made a constant into a parameter.

Now let's define the stream of negative powers of ten:

```
tenths = block:
  fun by-ten(d):
```

```

new-denom = d / 10
lz-link(new-denom, lam(): by-ten(new-denom) end)
end
by-ten(1)
end

```

so that

```

check:
take(3, tenths) is [list: 1/10, 1/100, 1/1000]
end

```

For concreteness, let's pick an abscissa at which to compute the numeric derivative of `square`—say 10:

```
d-dx-square-at-10 = d-dx-square(10)
```

Recall, from the types, that this is now a function of type `(Number -> Number)`: given a value for `epsilon`, it computes the derivative using that value. We know, analytically, that the value of this derivative should be 20. We can now (lazily) map `tenths` to provide increasingly better approximations for `epsilon` and see what happens:

```
lz-map(d-dx-square-at-10, tenths)
```

Sure enough, the values we obtain are 20.1, 20.01, 20.001, and so on: progressively better numerical approximations to 20.

Exercise

Extend the above program to take a tolerance, and draw as many values from the `epsilon` stream as necessary until the difference between successive approximations of the derivative fall within this tolerance.

contents ← prev up next →

8.2 Queues from Lists

8.2 Queues from Lists Suppose you have a list. When you take its first element, you get the element that was most recently linked to create it. The next element is the second most recent one that was linked, and so on. That is, the last one in is the first one out. This is called a LIFO, short for “last-in-first-out”, data structure. A list is LIFO; we sometimes also refer to this as a stack.

But there are many settings where you want the first-in to be the first-out. When you stand at a supermarket line, try to purchase concert tickets, submit a job request, or any number of other tasks, you want to be rewarded, not punished, for being there first. That is, you want a FIFO instead. This is called a queue.

The game we’re playing here is that we want one datatype but our language has given us another (in this case, lists), and we have to figure out how to encode one in the other. We’ve see elsewhere how to encode sets with lists [Representing Sets as Lists]. Here let’s see how we can encode queues with lists.

With sets, we allowed the set type to be an alias for lists; that is, the two were the same. Another option we have when encoding is to create a completely new type that does nothing more than wrap a value of the encoding type. We’ll use that principle here to illustrate how that might work.

8.2.1 Using a Wrapper Datatype Concretely, here’s how we’ll represent queues. For all the code that follows, it’s helpful to use the Pyret type-checker to make sure we’re composing code correctly:

```
data Queue<T>:  
  | queue(l :: List<T>)  
end
```

With this encoding, we can start define a few helper functions: e.g., a way to construct an empty queue and to check for emptiness:

```
fun mk-mtq<T>() -> Queue<T>:  
  queue(empty)  
end  
  
fun is-mtq<T>(q :: Queue<T>) -> Boolean:  
  is-empty(q.l)  
end
```

Adding an element to a queue is usually called “enqueueing”. It has this type:

```
enqueue :: <T> Queue<T>, T -> Queue<T>
```

Here’s the corresponding implementation:

```
fun enqueue(q, e):  
  queue(link(e, q.l))  
end
```

Do Now!

Did we have a choice?

Yes, we did! We could have made the new element the first element or the last element. Be careful here: we mean the first or last element of the list that represents the queue, not of the queue itself. There, FIFO gives us no choice. We just happened to choose one representation. The other would be equally valid; we would just need to implement all the other operations consistently. Let’s stick to this one for now.

Now we come to a problem. What does it mean to “dequeue”? We need to get back the one element, but we also need to get back the rest. Let’s first write this as two functions, very analogous to first and rest on lists:

```
qpeek :: <T> Queue<T> -> T
qrest :: <T> Queue<T> -> Queue<T>
```

Let’s write out a few examples to make sure we know how these should work:

```
q_ = mk-mtq()
q3 = enqueue(q_, 3)
m43 = enqueue(q3, 4)
m543 = enqueue(m43, 5)

check:
  qpeek(q3) is 3
  qpeek(m43) is 3
  qpeek(m543) is 3
end

check:
  qrest(q3) is mk-mtq()
  qrest(m43) is enqueue(mk-mtq(), 4)
  qrest(m543) is enqueue(enqueue(mk-mtq(), 4), 5)
end
```

Now let’s implement these:

```
fun qpeek(q):
  if is-mtq(q):
    raise("can't peek an empty queue")
  else:
    q.l.get(q.l.length() - 1)
  end
end

fun qrest(q):
  fun safe-rest(l :: List<T>) -> List<T>:
    cases (List) l:
      | empty => raise("can't dequeue an empty queue")
      | link(f, r) => r
    end
  end
  queue(safe-rest(q.l.reverse()).reverse())
end
```

8.2.2 Combining Answers However, it would be nice if we could obtain both the oldest element and the rest of the queue at once, if we want them both. That means the single function would need to return two values; since a function can return only one value at a time, it would need to use a data structure to hold both of them. Furthermore, note that both `qpeek` and `qrest` above have the possibility of not having any more elements! We might as well reflect that too in the type. Thus we end up with a type that looks like

```
data Dequeued<T>:
  | none-left
  | elt-and-q(e :: T, q :: Queue<T>)
end
```

Exercise

Write out the function to use this return type.

Observe that this also follows our principle of making exceptional behavior manifest in the return type: The Option Type, and especially in Summary.

Exercise

Write out the function using this return type.

8.2.3 Using a Picker Does `Dequeued` look familiar? Of course it should! It's basically the same as the pickers used for sets in Pyret: Picking Elements from Sets. If we make queues provide the same operations, we can reuse the `Pick` library already built into the language, and reuse any code that is written expecting the picker interface.

To do so, first we need to import the picker library:

```
include pick
```

Then we can write:

```
dequeue :: <T> Queue<T> -> Pick<T, Queue<T>>
```

Here are some examples showing how it would work:

```
check:
  dequeue(q_) is pick-none
  dequeue(q3) is pick-some(3, mk-mtq())
  dequeue(m43) is pick-some(3, enqueue(mk-mtq(), 4))
  dequeue(m543) is pick-some(3, enqueue(enqueue(mk-mtq(), 4), 5))
end
```

And here's the corresponding code:

```
fun dequeue<T>(q):
  rev = q.l.reverse()
  cases (List) rev:
    | empty => pick-none
    | link(f, r) =>
      pick-some(f, queue(r.reverse()))
  end
end
```

In terms of big-O complexity, this is a dreadfully inefficient implementation, causing two reversals on every `qrest` or `dequeue`. To see how to do better, and to conduct a more sophisticated analysis, see An Example: Queues from Lists.

One thing to note is that by providing only a picker interface, we're slightly changing the meaning of queues. The picker interface in Pyret is designed for sets, which don't have a notion of order. But queues are, of course, very much an ordered datatype; order is why they exist. So by providing only a picker interface, we don't offer the very guarantee that queues are designed for. Therefore, we should provide a picker in addition to an ordered interface, rather than in place of one.

At this point we're done with the essential content, but here are two more parts that you may find interesting.

8.2.4 Using Tuples Earlier, we created the `Dequeued` datatype to represent the return value from the `dequeue`. Indeed, it is often useful to create datatypes of this sort to document functions and make sure the types can be meaningfully interpreted even when their values flow around the code some distance from where they were created.

Sometimes, however, we want to create a compound datum in a special circumstance: it represents the return value of a function, and that return value will not live for very long, i.e., it will be taken apart as soon as it has returned and only the constituent parts will be used thereafter. In such situations, it can feel like a burden to create a new datatype for such a fleeting purpose. For such cases, Pyret has a built-in generic datatype called the tuple.

Here are some examples of tuples, which illustrate their syntax; note that each position (separated by `,`) takes an expression, not only a constant value:

```
{1; 2}  
{3; 4; 5}  
{1 + 2; 3}  
{6}  
{}
```

We can also pull values out of tuples as follows:

```
{a; b} = {1; 2}
```

Evaluate **a** and **b** and see what they are bound to.

```
{c; d; e} = {1 + 2; 6 - 2; 5}
```

Similarly, see what **c**, **d**, and **e** are bound to.

Exercise

What happens if we use too few or too many variables? Try out the following in Pyret and see what happens:

```
{p; q} = {1}  
{p} = {1; 2}  
{p} = 1
```

Do Now!

What happens if instead we write this?

```
p = {1; 2}
```

This binds **p** to the entire tuple.

Exercise

How might we pull apart the constituents of **p**?

Now that we have tuples, we can write dequeue as:

```
fun dequeue-tuple<T>(q :: Queue<T>) -> {T; Queue<T>}:  
  rev = q.l.reverse()  
  cases (List) rev:  
    | empty => raise("can't dequeue an empty queue")  
    | link(f, r) =>  
      {f; queue(r.reverse())}  
  end  
end  
  
check:  
  dequeue-tuple(q3) is {3; mk-mtq()}  
  dequeue-tuple(m43) is {3; enqueue(mk-mtq(), 4)}  
  dequeue-tuple(m543) is {3; enqueue(enqueue(mk-mtq(), 4), 5)}  
end
```

And here's how we can use it more generally:

```
fun q2l<T>(q :: Queue<T>) -> List<T>:  
  if is-mtq(q):  
    empty  
  else:  
    {e; rq} = dequeue-tuple(q)  
    link(e, q2l(rq))  
  end  
end
```

```

check:
  q21(mk-mtq()) is empty
  q21(q3) is [list: 3]
  q21(m43) is [list: 3, 4]
  q21(m543) is [list: 3, 4, 5]
end

```

You should feel free to use tuples in your programs provided you follow the rules above for when tuples are applicable. In general, tuples can cause a reduction in readability, and increase the likelihood of errors (because tuples from one source aren't distinguishable from those from another source). Use them with caution!

8.2.5 A Picker Method Second, and this is truly optional: you may have noticed earlier that `Sets` had a built-in `pick` method. We have a function, but not method, that picks. Now we'll see how we can write this as a method:

```

data Queue<T>:
  | queue(l :: List<T>) with:
    method pick(self):
      rev = self.l.reverse()
      cases (List) rev:
        | empty => pick-none
        | link(f, r) =>
          pick-some(f, queue(r.reverse()))
      end
    end
  end
end

```

This is a drop-in replacement for our previous definition of `Queue`, because we've added a method but left the general datatype structure intact, so all our existing code will still work. In addition, we can rewrite `q21` in terms of the picker interface:

```

fun q21m<T>(c :: Queue<T>) -> List<T>:
  cases (Pick) c.pick():
    | pick-none => empty
    | pick-some(e, r) => link(e, q21m(r))
  end
end

check:
  q21m(m543)
end

```

We can also write generic programs over data that support the `Pick` interface. For instance, here's a function that will convert anything satisfying that interface into a list:

```

fun pick21<T>(c) -> List<T>:
  cases (Pick) c.pick():
    | pick-none => empty
    | pick-some(e, r) => link(e, pick21(r))
  end
end

```

For instance, it works on both sets and our new `Queues`:

```

import sets as S      # put this at the top of the file

check:
  pick21([S.set: 3, 4, 5]).sort() is [list: 3, 4, 5]
  pick21(m543) is [list: 3, 4, 5]
end

```

Exercise

Do you see why we invoked `sort` in the test above?

The only weakness here is that for this last part (making the function generic), we have to transition out of the type-checker, because `pick21` cannot be typed by the current Pyret type checker. It requires a feature that the type checker does not (yet) have.

contents ← prev up next →

8.3 Examples, Testing, and Program Checking

8.3 Examples, Testing, and Program Checking Back in Documenting Functions with Examples, we began to develop your habit of writing concrete examples of functions. In Task Plans, we showed you how to develop examples of intermediate values to help you plan the code for you to write. As these examples show, there are many ways to write down examples. We could write them on a board, on paper, or even as comments in a computer document. These are all reasonable and indeed, often, the best way to begin working on a problem. However, if we can write our examples in a precise form that a computer can understand, we achieve two things:

- When we're done writing our purported solution, we can have the computer check whether we got it right.
- In the process of writing down our expectation, we often find it hard to express with the precision that a computer expects. Sometimes this is because we're still formulating the details and haven't yet pinned them down, but at other times it's because we don't yet understand the problem. In such situations, the force of precision actually does us good, because it helps us understand the weakness of our understanding.

8.3.1 From Examples to Tests Until now, we have written examples in `where:` blocks for two purposes: to help us figure out what a function needs to do, and to provide guidance to someone reading our code as to what behavior they can expect when using our function. For the smaller programs that we have written until now, `where`-based examples have been sufficient. As our programs get more complicated, however, a small set of related illustrative examples won't suffice. We need to think about being much more thorough in the sets of inputs that we consider.

Consider for example a function `count-uses` that counts how many times a specific string appears in a list (this could be used to tally votes, to compute the frequency of using a discount code, and so on). What input scenarios might we need to check before using our function to run an actual election or a business?

- The result for a string that is in the list once
- The result for a string that is in the list multiple times
- The result for a string that is at the end of a longer list (to make sure we are checking all of the elements)
- The result for a string that isn't in the list
- The result for a string that is in the list but with different capitalization
- The result for a string that is a typo-away from a word in the list

Notice that here we are considering many more situations, including fairly nuanced ones that affect how robust our code would be under realistic situations. Once we start considering situations like these, we are shifting from examples to illustrate our code to tests to thoroughly test our code.

In Pyret, we use `where` blocks inside function definitions for examples. We use a `check` block outside the function definition for tests. For example:

```
fun count-uses(of-string :: String, in-list :: List<String>) -> Number:  
  ...  
  where:  
    count-uses("pepper", [list:]) is 0  
    count-uses("pepper", [list: "onion"]) is 0  
    count-uses("pepper", [list: "pepper", "onion"]) is 1  
    count-uses("pepper", [list: "pepper", "pepper", "onion"]) is 2  
end
```

```

check:
  count-uses("ppper", [list: "pepper"]) is 0
  count-uses("ONION", [list: "pepper", "onion"]) is 1
  count-uses("tomato",
    [list: "pepper", "onion", "onion", "pepper", "tomato",
     "tomato", "onion", "tomato"])
  is 3
  ...
end

```

As a guiding rule, we put illustrative cases that would help someone else reading our code into the `where` block, while we put the nitty-gritty checks that our code handles the wider range of usage scenarios (including error cases) into the `check`. Sometimes, the line between these two isn't clear: for example, one could easily argue that the second test (the function handles different capitalization) belongs in `where` instead. The third test about using a really long list would remain in `check`, however, as longer inputs are generally not instructive to a reader of your code.

Putting tests in a block that lives outside the function has another advantage at the level of professional programming: it allows your tests to live in a separate file from your code. This has two key benefits. First, it makes it easier for someone to read the essential parts of your code (if they are building on your work). Second, it makes it easier to control when tests are run. When your `check` blocks are in the same file as your code, all the tests will be checked when you run your code. When they are in a different file, an organization can choose when to run the tests. During development, tests are run frequently to make sure no errors have been introduced. Once code is tested and ready to be deployed or used, tests are not run along with the program (unless there has been a modification or someone has discovered an error with the code). This is standard practice in software projects.

It is also worth noting that the collection of tests grows throughout the development process, moreso than do the collection of examples. As you are developing code, every time you find a bug in your code, add a test for it in your `check` block so you don't accidentally introduce that same error again later. Whereas we develop examples up front as we figure out what we want our program to do, we augment our tests as we discover what our program actually does (and perhaps should not do). In practice, developers write an initial set of checks on the scenarios they thought of before and while writing the code, then expand those tests as they try out more scenarios and gain users who report scenarios where the code does not work.

Nearly all programming languages come with some constructs or packages in which you can write tests in separate files. Pyret is unique in supporting the distinction between examples and tests (both for learning and for readability of code by others). Many programming tools that support professionals expect you to put all tests in separate folders and files (offering no support for examples). In this book, we emphasize the difference between these two uses of input-output pairs in programming because we find them extremely useful both professionally and pedagogically.

8.3.2 More Refined Comparisons Sometimes, a direct comparison via `is` isn't enough for testing. We have already seen this in the case of `raises` tests (Computing Genetic Parents from an Ancestry Table). As another example, when doing some computations, especially involving math with approximations, the exact match of `is` isn't feasible. For example, consider these tests for `distance-to-origin`:

```

check:
  distance-to-origin(point(1, 1)) is ???
end

```

What can we check here? Typing this into the REPL, we can find that the answer prints as `1.4142135623730951`. That's an approximation of the real answer, which Pyret cannot represent exactly. But it's hard to know that this precise answer, to this decimal place, and no more, is the one we should expect up front, and thinking through the answers is supposed to be the first thing we do!

Since we know we're getting an approximation, we can really only check that the answer is roughly correct, not exactly correct. If we can check that the answer to `distance-to-origin(point(1, 1))` is around, say, 1.41, and can do the same for some similar cases, that's probably good enough for many applications, and for our purposes here. If we were calculating orbital dynamics, we might demand higher precision, but note that we'd still need to pick a cutoff! Testing for inexact results is a necessary task.

Let's first define what we mean by "around" with one of the most precise ways we can, a function:

```

fun around(actual :: Number, expected :: Number) -> Boolean:
    doc: "Return whether actual is within 0.01 of expected"
    num-abs(actual - expected) < 0.01
where:
    around(5, 5.01) is true
    around(5.01, 5) is true
    around(5.02, 5) is false
    around(num-sqrt(2), 1.41) is true
end

```

The `is` form now helps us out. There is special syntax for supplying a user-defined function to use to compare the two values, instead of just checking if they are equal:

```

check:
    5 is%(around) 5.01
    num-sqrt(2) is%(around) 1.41
    distance-to-origin(point(1, 1)) is%(around) 1.41
end

```

Adding `%(something)` after `is` changes the behavior of `is`. Normally, it would compare the left and right values for equality. If something is provided with `%`, however, it instead passes the left and right values to the provided function (in this example `around`). If the provided function produces `true`, the test passes, if it produces `false`, the test fails. This gives us the control we need to test functions with predictable approximate results.

Exercise

Extend the definition of `distance-to-origin` to include polar points.

Exercise

This might save you a Google search: polar conversions. Use the design recipe to write `x-component` and `y-component`, which return the `x` and `y` Cartesian parts of the point (which you would need, for example, if you were plotting them on a graph). Read about `num-sin` and other functions you'll need at the Pyret number documentation.

Exercise

Write a data definition called `Pay` for pay types that includes both hourly employees, whose pay type includes an hourly rate, and salaried employees, whose pay type includes a total salary for the year. Use the design recipe to write a function called `expected-weekly-wages` that takes a `Pay`, and returns the expected weekly salary: the expected weekly salary for an hourly employee assumes they work 40 hours, and the expected weekly salary for a salaried employee is 1/52 of their salary.

8.3.3 When Tests Fail Suppose we've written the function `sqrt`, which computes the square root of a given number. We've written some tests for this function. We run the program, and find that a test fails. There are two obvious reasons why this can happen.

Do Now!

What are the two obvious reasons?

The two reasons are, of course, the two “sides” of the test: the problem could be with the values we've written or with the function we've written. For instance, if we've written

`sqrt(4) is 1.75`

then the fault clearly lies with the values (because $\sqrt{1.75^2}$ is clearly not $\sqrt{4}$). On the other hand, if it fails the test

`sqrt(4) is 2`

then the odds are that we've made an error in the definition of `sqrt` instead, and that's what we need to fix.

Note that there is no way for the computer to tell what went wrong. When it reports a test failure, all it's saying is that there is an inconsistency between the program and the tests. The computer is not passing judgment on which one is "correct", because it can't do that. That is a matter for human judgment. For this reason, we've been doing research on peer review of tests, so students can help one another review their tests before they begin writing programs.

Actually...not so fast. There's one more possibility we didn't consider: the third, not-so-obvious reason why a test might fail. Return to this test:

```
sqrt(4) is 2
```

Clearly the inputs and outputs are correct, but it could be that the definition of `sqrt` is also correct, and yet the test fails.

Do Now!

Do you see why?

Depending on how we've programmed `sqrt`, it might return the root -2 instead of 2 . Now -2 is a perfectly good answer, too. That is, neither the function nor the particular set of test values we specified is inherently wrong; it's just that the function happens to be a relation, i.e., it maps one input to multiple outputs (that is, $\sqrt{4} = \pm 2$). The question now is how to write the test properly.

8.3.4 Oracles for Testing In other words, sometimes what we want to express is not a concrete input-output pair, but rather check that the output has the right relationship to the input. Concretely, what might this be in the case of `sqrt`? We hinted at this earlier when we said that 1.75 clearly can't be right, because squaring it does not yield 4 . That gives us the general insight: that a number is a valid root (note the use of "a" instead of "the") if squaring it yields the original number. That is, we might write a function like this:

```
fun is-sqrt(n):
    n-root = sqrt(n)
    n == (n-root * n-root)
end
```

and then our test looks like

```
check:
  is-sqrt(4) is true
end
```

Unfortunately, this has an awkward failure case. If `sqrt` does not produce a number that is in fact a root, we aren't told what the actual value is; instead, `is-sqrt` returns false, and the test failure just says that `false` (what `is-sqrt` returns) is not `true` (what the test expects)—which is both absolutely true and utterly useless.

Fortunately, Pyret has a better way of expressing the same check. Instead of `is`, we can write `satisfies`, and then the value on the left must satisfy the predicate on the right. Concretely, this looks like:

```
fun check-sqrt(n):
  lam(n-root):
    n == (n-root * n-root)
  end
end
```

which lets us write:

```
check:
  sqrt(4) satisfies check-sqrt(4)
end
```

Now, if there's a failure, we learn of the actual value produced by `sqrt(4)` that failed to satisfy the predicate.

9.1 From Pyret to Python

9.1 From Pyret to Python Through our work in Pyret to this point, we've covered several core programming skills: how to work with tables, how to design good examples, the basics of creating datatypes, and how to work with the fundamental computational building blocks of functions, conditionals, and repetition (through `filter` and `map`, as well as recursion). You've got a solid initial toolkit, as well as a wide world of other possible programs ahead of you!

But we're going to shift gears for a little while and show you how to work in Python instead. Why?

Seeing how the same concepts play out in multiple languages can help you distinguish core computational ideas from the notations and idioms of specific languages. If you plan to write programs as part of your professional work, you'll inevitably have to work in different languages at different times: we're giving you a chance to practice that skill in a controlled and gentle setting.

Why do we call this gentle? Because the notations in Pyret were designed partly with this transition in mind. You'll find many similarities between Pyret and Python at a notational level, yet also some interesting differences that highlight some philosophical differences that underlie languages. The next set of programs that we want to write (specifically, data-rich programs where the data must be updated and maintained over time) fit nicely with certain features of Python that you haven't seen in Pyret. A future release will contain material that contrasts the strengths and weaknesses of the two languages.

We highlight the basic notational differences between Pyret and Python by redoing some of our earlier code examples in Python.

9.1.1 Expressions, Functions, and Types Back in Functions Practice: Cost of pens, we introduced the notation for functions and types using an example of computing the cost of an order of pens. An order consisted of a number of pens and a message to be printed on the pens. Each pen cost 25 cents, plus 2 cents per character for the message. Here was the original Pyret code:

```
fun pen-cost(num-pens :: Number, message :: String) -> Number:  
    doc: ```total cost for pens, each 25 cents  
        plus 2 cents per message character```  
    num-pens * (0.25 + (string-length(message) * 0.02))  
end
```

Here's the corresponding Python code:

```
def pen_cost(num_pens: int, message: str) -> float:  
    """total cost for pens, each at 25 cents plus  
    2 cents per message character"""  
    return num_pens * (0.25 + (len(message) * 0.02))
```

Do Now!

What notational differences do you see between the two versions?

Here's a summary of the differences:

- Python uses `def` instead of `fun`.
- Python uses underscores in names (like `pen_cost`) instead of hyphens as in Pyret.
- The type names are written differently: Python uses `str` and `int` instead of `String` and `Number`. In addition, Python uses only a single colon before the type whereas Pyret uses a double colon.

- Python has different types for different kinds of numbers: `int` is for integers, while `float` is for decimals. Pyret just used a single type (`Number`) for all numbers.
- Python doesn't label the documentation string (as Pyret does with `doc:`).
- There is no `end` annotation in Python. Instead, Python uses indentation to locate the end of an if/else statement, function, or other multi-line construct finishes.
- Python labels the outputs of functions with `return`.

These are minor differences in notation, which you will get used to as you write more programs in Python.

There are differences beyond the notational ones. One that arises with this sample program arises around how the language uses types. In Pyret, if you put a type annotation on a parameter then pass it a value of a different type, you'll get an error message. Python ignores the type annotations (unless you bring in additional tools for checking types). Python types are like notes for programmers, but they aren't enforced when programs run.

Exercise

Convert the following `moon-weight` function from Functions Practice: Moon Weight into Python:

```
fun moon-weight(earth-weight :: Number) -> Number:
    doc: "Compute weight on moon from weight on earth"
    earth-weight * 1/6
end
```

9.1.2 Returning Values from Functions In Pyret, a function body consisted of optional statements to name intermediate values, followed by a single expression. The value of that single expression is the result of calling the function. In Pyret, every function produces a result, so there is no need to label where the result comes from.

As we will see, Python is different: not all “functions” return results (note the name change from `fun` to `def`). In mathematics, functions have results by definition. Programmers sometimes distinguish between the terms “function” and “procedure”: both refer to parameterized computations, but only the former returns a result to the surrounding computation. Some programmers and languages do, however, use the term “function” more loosely to cover both kinds of parameterized computations. Moreover, the result isn't necessarily the last expression of the `def`. In Python, the keyword `return` explicitly labels the expression whose value serves as the result of the function.

Do Now!

Put these two definitions in a Python file.

```
def add1v1(x: int) -> int:
    return x + 1

def add1v2(x: int) -> int:
    x + 1
```

At the Python prompt, call each function in turn. What do you notice about the result from using each function?

Hopefully, you noticed that using `add1v1` displays an answer after the prompt, while using `add1v2` does not. This difference has consequences for composing functions.

Do Now!

Try evaluating the following two expressions at the Python prompt: what happens in each case?

```
3 * add1v1(4)

3 * add1v2(4)
```

This example illustrates why `return` is essential in Python: without it, no value is returned, which means you can't use the result of a function within another expression. So what use is `add1v2` then? Hold that question; we'll return to it in Modifying Variables.

9.1.3 Examples and Test Cases In Pyret, we included examples with every function using `where:` blocks. We also had the ability to write `check:` blocks for more extensive tests. As a reminder, here was the `pen-cost` code including a `where:` block:

```
fun pen-cost(num-pens :: Number, message :: String) -> Number:  
    doc: ```total cost for pens, each 25 cents  
        plus 2 cents per message character```  
    num-pens * (0.25 + (string-length(message) * 0.02))  
where:  
    pen-cost(1, "hi") is 0.29  
    pen-cost(10, "smile") is 3.50  
end
```

Python does not have a notion of `where:` blocks, or a distinction between examples and tests. There are a couple of different testing packages for Python; here we will use `pytest`, a standard lightweight framework that resembles the form of testing that we did in Pyret. How you set up `pytest` and your test file contents will vary according to your Python IDE. We assume instructors will provide separate instructions that align with their tool choices. To use `pytest`, we put both examples and tests in a separate function. Here's an example of this for the `pen_cost` function:

```
import pytest  
  
def pen_cost(num_pens: int, message: str) -> float:  
    """total cost for pens, each at 25 cents plus  
    2 cents per message character"""  
    return num_pens * (0.25 + (len(message) * 0.02))  
  
def test_pens():  
    assert pen_cost(1, "hi") == 0.29  
    assert pen_cost(10, "smile") == 3.50
```

Things to note about this code:

- We've imported `pytest`, the lightweight Python testing library.
- The examples have moved into a function (here `test_pens`) that takes no inputs. Note that the names of functions that contain test cases must have names that start with `test_` in order for `pytest` to find them.
- In Python, individual tests have the form
`assert EXPRESSION == EXPECTED_ANS`
rather than the `is` form from Pyret.

Do Now!

Add one more test to the Python code, corresponding to the Pyret test

```
pen-cost(3, "wow") is 0.93
```

Make sure to run the test.

Do Now!

Did you actually try to run the test?

Whoa! Something weird happened: the test failed. Stop and think about that: the same test that worked in Pyret failed in Python. How can that be?

9.1.4 An Aside on Numbers It turns out that different programming languages make different decisions about how to represent and manage real (non-integer) numbers. Sometimes, differences in these representations lead to subtle quantitative differences in computed values. As a simple example, let's look at two seemingly simple real numbers $1/2$ and $1/3$. Here's what we get when we type these two numbers at a Pyret prompt:

0.5

1/3

0.3

If we type these same two numbers in a Python console, we instead get:

1/2

0.5

1/3

0.3333333333333333

Notice that the answers look different for $1/3$. As you may (or may not!) recall from an earlier math class, $1/3$ is an example of a non-terminating, repeating decimal. In plain terms, if we tried to write out the exact value of $1/3$ in decimal form, we would need to write an infinite sequence of 3s. Mathematicians denote this by putting a horizontal bar over the 3. This is the notation we see in Pyret. Python, in contrast, writes out a partial sequence of 3s.

Underneath this distinction lies some interesting details about representing numbers in computers. Computers don't have infinite space to store numbers (or anything else, for that matter): when a program needs to work with a non-terminating decimal, the underlying language can either:

- approximate the number (by chopping off the infinite sequence of digits at some point), then work only with the approximated value going forward, or
- store additional information about the number that may enable doing more precise computation with it later (though there are always some numbers that cannot be represented exactly in finite space).

Python takes the first approach. As a result, computations with the approximated values sometimes yield approximated results. This is what happens with our new `pen_cost` test case. While mathematically, the computation should result in 0.93, the approximations yield 0.9299999999999999 instead.

So how do we write tests in this situation? We need to tell Python that the answer should be "close" to 0.93, within the error range of approximations. Here's what that looks like:

```
assert pen_cost(3, "wow") == pytest.approx(0.93)
```

We wrapped the exact answer we wanted in `pytest.approx`, to indicate that we'll accept any answer that is nearly the value we specified. You can control the number of decimal points of precision if you want to, but the default of $\pm 2.3e-06$ often suffices.

9.1.5 Conditionals Continuing with our original `pen_cost` example, here's the Python version of the function that computed shipping costs on an order:

```
def add_shipping(order_amt: float) -> float:  
    """increase order price by costs for shipping"""  
    if order_amt == 0:  
        return 0  
    elif order_amt <= 10:  
        return order_amt + 4  
    elif (order_amt > 10) and (order_amt < 30):  
        return order_amt + 8  
    else:  
        return order_amt + 12
```

The main difference to notice here is that `else if` is written as the single-word `elif` in Python. We use `return` to mark the function's results in each branch of the conditional. Otherwise, the conditional constructs are quite similar across the two languages.

You may have noticed that Python does not require an explicit `end` annotation on `if`-expressions or functions. Instead, Python looks at the indentation of your code to determine when a construct has ended. For example, in the code sample for `pen_cost` and `test_pens`, Python determines that the `pen_cost` function has ended because it

detects a new definition (for `test_pens`) at the left edge of the program text. The same principle holds for ending conditionals.

We'll return to this point about indentation, and see more examples, as we work more with Python.

9.1.6 Creating and Processing Lists As an example of lists, let's assume we've been playing a game that involves making words out of a collection of letters. In Pyret, we could have written a sample word list as follows:

```
words = [list: "banana", "bean", "falafel", "leaf"]
```

In Python, this definition would look like:

```
words = ["banana", "bean", "falafel", "leaf"]
```

The only difference here is that Python does not use the `list:` label that is needed in Pyret.

9.1.6.1 Filters, Maps, and Friends When we first learned about lists in Pyret, we started with common built-in functions such as `filter`, `map`, `member` and `length`. We also saw the use of `lambda` to help us use some of these functions concisely. These same functions, including `lambda`, also exist in Python. Here are some samples:

```
words = ["banana", "bean", "falafel", "leaf"]

# filter and member
words_with_b = list(filter(lambda wd: "b" in wd, words))
# filter and length
short_words = list(filter(lambda wd: len(wd) < 5, words))
# map and length
word_lengths = list(map(len, words))
```

Note that you have to wrap calls to `filter` (and `map`) with a use of `list()`. Internally, Python has these functions return a type of data that we haven't yet discussed (and don't need). Using `list` converts the returned data into a list. If you omit the `list`, you won't be able to chain certain functions together. For example, if we tried to compute the length of the result of a `map` without first converting to a `list`, we'd get an error:

```
len(map(len, b))

TypeError: object of type 'map' has no len()
```

Don't worry if this error message makes no sense at the moment (we haven't yet learned what an "object" is). The point is that if you see an error like this while using the result of `filter` or `map`, you likely forgot to wrap the result in `list`.

Exercise

Practice Python's list functions by writing expressions for the following problems. Use only the list functions we have shown you so far.

- Given a list of numbers, convert it to a list of strings "`pos`", "`neg`", "`zero`", based on the sign of each number.
- Given a list of strings, is the length of any string equal to 5?
- Given a list of numbers, produce a list of the even numbers between 10 and 20 from that list.

We're intentionally focusing on computations that use Python's built-in functions for processing lists, rather than showing you how to write your own (as we did with recursion in Pyret). While you can write recursive functions to process lists in Pyret, a different style of program is more conventional for that purpose. We'll look at that in the chapter on Modifying Variables.

9.1.7 Data with Components An analog to a Pyret data definition (without variants) is called a dataclass in Python. Those experienced with Python may wonder why we are using dataclasses instead of dictionaries or raw classes. Compared to dictionaries, dataclasses allow the use of type hints and capture that our data has a fixed collection of fields. Compared to raw classes, dataclasses generate a lot of boilerplate code that makes them much

lighterweight than raw classes. Here's an example of a todo-list datatype in Pyret and its corresponding Python code:

```
# a todo item in Pyret
data ToDoItemData:
    | todoItem(descr :: String,
               due :: Date,
               tags :: List[String])
end

-----
# the same todo item in Python

# to allow use of dataclasses
from dataclasses import dataclass
# to allow dates as a type (in the ToDoItem)
from datetime import date

@dataclass
class ToDoItem:
    descr: str
    due: date
    tags: list

# a sample list of ToDoItem
myTD = [ToDoItem("buy milk", date(2020, 7, 27), ["shopping", "home"]),
        ToDoItem("grade hwk", date(2020, 7, 27), ["teaching"]),
        ToDoItem("meet students", date(2020, 7, 26), ["research"])]
    ]
```

Things to note:

- There is a single name for the type and the constructor, rather than separate names as we had in Pyret.
- There are no commas between field names (but each has to be on its own line in Python)
- There is no way to specify the type of the contents of the list in Python (at least, not without using more advance packages for writing types)
- The `@dataclass` annotation is needed before `class`.
- Dataclasses don't support creating datatypes with multiple variants, like we did frequently in Pyret. Doing that needs more advanced concepts than we will cover in this book.

9.1.7.1 Accessing Fields within Dataclasses In Pyret, we extracted a field from structured data by using a dot (period) to “dig into” the datum and access the field. The same notation works in Python:

```
travel = ToDoItem("buy tickets", date(2020, 7, 30), ["vacation"])

travel.descr

"buy tickets"
```

9.1.8 Traversing Lists

9.1.8.1 Introducing For Loops In Pyret, we wrote recursive functions to compute summary values over lists. As a reminder, here's a Pyret function that sums the numbers in a list:

```
fun sum-list(numlist :: List[Number]) -> Number:
    cases (List) numlist:
        | empty => 0
        | link(fst, rst) => fst + sum-list(rst)
```

```
    end
end
```

In Python, it is unusual to break a list into its first and rest components and process the rest recursively. Instead, we use a construct called a `for` to visit each element of a list in turn. Here's the form of `for`, using a concrete (example) list of odd numbers:

```
for num in [5, 1, 7, 3]:
    // do something with num
```

The name `num` here is of our choosing, just as with the names of parameters to a function in Pyret. When a `for` loop evaluates, each item in the list is referred to as `num` in turn. Thus, this `for` example is equivalent to writing the following:

```
// do something with 5
// do something with 1
// do something with 7
// do something with 3
```

The `for` construct saves us from writing the common code multiple times, and also handles the fact that the lists we are processing can be of arbitrary length (so we can't predict how many times to write the common code).

Let's now use `for` to compute the running sum of a list. We'll start by figuring out the repeated computation with our concrete list again. At first, let's express the repeated computation just in prose. In Pyret, our repeated computation was along the lines of "add the first item to the sum of the rest of the items". We've already said that we cannot easily access the "rest of the items" in Python, so we need to rephrase this. Here's an alternative:

```
// set a running total to 0
// add 5 to the running total
// add 1 to the running total
// add 7 to the running total
// add 3 to the running total
```

Note that this framing refers not to the "rest of the computation", but rather to the computation that has happened so far (the "running total"). If you happened to work through the chapter on [my-running-sum: Examples and Code](#), this framing might be familiar.

Let's convert this prose sketch to code by replacing each line of the sketch with concrete code. We do this by setting up a variable named `run_total` and updating its value for each element.

```
run_total = 0
run_total = run_total + 5
run_total = run_total + 1
run_total = run_total + 7
run_total = run_total + 3
```

This idea that you can give a new value to an existing variable name is something we haven't seen before. In fact, when we first saw how to name values (in The Program Directory), we explicitly said that Pyret doesn't let you do this (at least, not with the constructs that we showed you). Python does. We'll explore the consequences of this ability in more depth shortly (in Modifying Variables). For now, let's just use that ability so we can learn the pattern for traversing lists. First, let's collapse the repeated lines of code into a single use of `for`:

```
run_total = 0
for num in [5, 1, 7, 3]:
    run_total = run_total + num
```

This code works fine for a specific list, but our Pyret version took the list to sum as a parameter to a function. To achieve this in Python, we wrap the `for` in a function as we have done for other examples earlier in this chapter. This is the final version.

```
def sum_list(numlist : list) -> float:
    """sum a list of numbers"""
    run_total = 0
```

```

for num in numlist:
    run_total = run_total + num
return(run_total)

```

Do Now!

Write a set of tests for `sum_list` (the Python version).

Now that the Python version is done, let's compare it to the original Pyret version:

```

fun sum-list(numlist :: List[Number]) -> Number:
    cases (List) numlist:
        | empty => 0
        | link(fst, rst) => fst + sum-list(rst)
    end
end

```

Here are some things to notice about the two pieces of code:

- The Python version needs a variable (here `run_total`) to hold the result of the computation as we build it up while traversing (working through) the list.
- The initial value of that variable is the answer we returned in the `empty` case in Pyret.
- The computation in the `link` case of the Pyret function is used to update that variable in the body of the `for`.
- After the `for` has finished processing all items in the list, the Python version returns the value in the variable as the result of the function.

9.1.8.1.1 An Aside on Order of Processing List Elements There's another subtlety here if we consider how the two programs run: the Python version sums the elements from left to right, whereas the Pyret version sums them right to left. Concretely, the sequence of values of `run_total` are computed as:

```

run_total = 0
run_total = 0 + 5
run_total = 5 + 1
run_total = 6 + 7
run_total = 13 + 3

```

In contrast, the Pyret version unrolls as:

```

sum_list([list: 5, 1, 7, 3])
5 + sum_list([list: 1, 7, 3])
5 + 1 + sum_list([list: 7, 3])
5 + 1 + 7 + sum_list([list: 3])
5 + 1 + 7 + 3 + sum_list([list:])
5 + 1 + 7 + 3 + 0
5 + 1 + 7 + 3
5 + 1 + 10
5 + 11
16

```

As a reminder, the Pyret version did this because the `+` in the `link` case can only reduce to an answer once the sum of the rest of the list has been computed. Even though we as humans see the chain of `+` operations in each line of the Pyret unrolling, Pyret sees only the expression `fst + sum-list(rst)`, which requires the function call to finish before the `+` executes.

In the case of summing a list, we don't notice the difference between the two versions because the sum is the same whether we compute it left-to-right or right-to-left. In other functions we write, this difference may start to matter.

9.1.8.2 Using For Loops in Functions that Produce Lists Let's practice using `for` loops on another function that traverses lists, this time one that produces a list. Specifically, let's write a program that takes a list of strings and produces a list of words within that list that contain the letter "z".

As in our `sum_list` function, we will need a variable to store the resulting list as we build it up. The following code calls this `zlist`. The code also shows how to use `in` to check whether a character is in a string (it also works for checking whether an item is in a list) and how to add an element to the end of a list (`append`).

```
def all_z_words(wordlist : list) -> list:  
    """produce list of words from the input that contain z"""\n    zlist = [] // start with an empty list\n    for wd in wordlist:\n        if "z" in wd:\n            zlist = [wd] + zlist\n    return(zlist)
```

This code follows the structure of `sum_list`, in that we update the value of `zlist` using an expression similar to what we would have used in Pyret. For those with prior Python experience who would have used `zlist.append` here, hold that thought. We will get there in Mutable Lists.

Exercise

Write tests for `all_z_words`.

Exercise

Write a second version of `all_z_words` using `filter`. Be sure to write tests for it!

Exercise

Contrast these two versions and the corresponding tests. Did you notice anything interesting?

9.1.8.3 Summary: The List-Processing Template for Python Just as we had a template for writing list-processing functions in Pyret, there is a corresponding template in Python based on `for` loops. As a reminder, that pattern is as follow:

```
def func(lst: list):  
    result = ... # what to return if the input list is empty  
    for item in lst:  
        # combine item with the result so far  
        result = ... item ... result  
    return result
```

Keep this template in mind as you learn to write functions over lists in Python.

contents ← prev up next →

9.2 Dictionaries

9.2 Dictionaries So far, we have seen several ways to process sequential data such as lists. In each of Pyret and Python, we can use `filter` and `map` to perform certain operations that yield lists. In Pyret, we used recursion to aggregate list data into a single value. In Python, we used for-loops for this task. While we could use recursion or for-loops for `filter` and `map` tasks as well, using these named operators makes it easier for someone else to quickly read your code and understand what kind of operation it is performing.

This observation raises a question though: are there other common code patterns that get written with recursion or for-loops that would benefit from specialized handling?

As an example, imagine that we had a dataclass for airline flights. Each flight has its origin and destination cities, the flight code (including the airline name and flight number), and the number of seats on the flight. Imagine also that we have functions to look up the destination and seating capacity of individual flights:

```
@dataclass
class Flight:
    from_city: str
    to_city: str
    code: str
    seats: int

schedule = [Flight('NYC','PVD','CSA-342',50),
            Flight('PVD','ORD','CSA-590',50),
            Flight('NYC','ORD','CSA-723',120),
            Flight('ORD','DEN','CSA-145',175),
            Flight('BOS','ORD','CSA-647',80)]

def destination1(for_code: str, flights: list):
    '''get the destination of the flight with the given code'''
    for fl in flights:
        if fl.code == for_code:
            return fl.to_city

def capacity1(for_code: str, flights: list):
    '''get the seating capacity of the flight with the given code'''
    for fl in flights:
        if fl.code == for_code:
            return fl.seats
```

Do Now!

Look at the similarity between `destination1` and `capacity1`. How might we share the common code between these two functions?

Both `destination1` and `capacity1` traverse the list of flights looking for the one with the given flight-code, then extract a piece of information from that flight. The for-loop isn't doing anything other than looking for the desired flight data. This suggests that a `find_flight` helper could be useful here:

```
def find_flight(for_code: str, flights: list):
    '''return the flight with the given code'''
    for fl in flights:
```

```

if fl.code == for_code:
    return fl

def destination2(for_code: str, flights: list):
    return find_flight(for_code, flights).to_city

def capacity2(for_code: str, flights: list):
    return find_flight(for_code, flights).seats

```

Searching for a single element from a list based on a specific piece of information is common in many programs. This is so common, in fact, that languages provide special data structures and operations just to help with this task. In Python, this data structure is called a dictionary (hashmap, hashtable, and associative arrays are names for similar data structures in other languages, though there are key nuances that distinguish all these variations).

9.2.1 Creating and Using a Dictionary A dictionary maps unique values (called keys) to corresponding pieces of data for each key (called values). Here is our flight example written instead as a dictionary instead of a list:

```

sched_dict = {'CSA-342': Flight('NYC', 'PVD', 'CSA-342', 50),
              'CSA-590': Flight('PVD', 'ORD', 'CSA-590', 50),
              'CSA-723': Flight('NYC', 'ORD', 'CSA-723', 120),
              'CSA-145': Flight('ORD', 'DEN', 'CSA-145', 175),
              'CSA-647': Flight('BOS', 'ORD', 'CSA-647', 80)
}

```

The general form of a dictionary is:

```
{key1: value1,
 key2: value2,
 key3: value3,
 ...}
```

Dictionaries are designed to enable easy lookup of values give a key. To get the

Flight

associated with key 'CSA-145', we can write simply:

```
sched_dict['CSA-145']
```

To get the number of seats on flight 'CSA-145', we can simply write:

```
sched_dict['CSA-145'].seats
```

In other words, the dictionary data structure removes the need to traverse a list to find the `Flight` with a specific key. The dictionary lookup operation does that work for us. Actually, dictionaries are even more nuanced: depending on how they are designed, dictionaries can retrieve the value for a key without traversing all the values (or even any other value). In general, you can assume that dictionary-based lookup is significantly faster than a list-based one. How this works is a more advanced topic; some of this content is explained in [SECREF].

One limitation of dictionaries is that they allow only one value per key. Let's consider a different example, this time one that uses rooms in a building as keys and occupants as values:

```

office_dict = {410: 'Farhan',
               411: 'Pauline',
               412: 'Marisol',
               413: 'Saleh'}

```

What if someone new moves into office 412? In Python, we can the value for that key as follows:

```
office_dict[412] = 'Zeynep'
```

Now, any use of `office_dict[412]` will evaluate to 'Zeynep' instead of 'Marisol'.

9.2.2 Searching Through the Values in a Dictionary What if we wanted to find all of the flights with more than 100 seats? For this, we have to search through all of the key-value pairs and check their balances. This again sounds like we need a for-loop. What does that look like on a dictionary though?

Turns out, it looks much like writing a for loop on a list (at least in Python). Here's a program that creates a list of the flights with more than 100 seats:

```
above_100 = []

# the room variable takes on each key in the dictionary
for flight_code in sched_dict:
    if sched_dict[flight_code].seats > 100:
        above_100.append(sched_dict[flight_code])
```

Here, the for-loop iterates over the keys. Within the loop, we use each key to retrieve its corresponding Flight, perform the balance check on the Flight, then put the Flight in our running list if it meets our criterion.

Exercise

Create a dictionary that maps names of classrooms or meeting rooms to the numbers of seats that they have. Write expressions to:

1. Look up how many seats are in a specific room
2. Change the capacity of a specific room to have 10 more seats than it did initially
3. Find all rooms that can seat at least 50 students

9.2.3 Dictionaries with More Complex Values

Do Now!

A track-and-field tournament needs to manage the names of the players on the individual teams that will be competing. For example, “Team Red” has “Shaoming” and “Lijin”, “Team Green” contains “Obi” and “Chinara”, and “Team Blue” has “Mateo” and “Sophia”. Come up with a way to organize the data that will allow the organizers to easily access the names of the players on each team, keeping in mind that there could be many more teams than just the three listed here.

This feels like a dictionary situation, in that we have a meaningful key (the team name) with which we want to access values (the names of the players). However, we have already said that dictionaries allow only one value per key. Consider the following code:

```
players = {}
players["Team Red"] = "Shaoming"
players["Team Red"] = "Lijin"
```

Do Now!

What would be in the dictionary after running this code? If you aren't sure, try it out!

How do we store multiple player names under the same key? The insight here is that the collection of players, not an individual player, is what we want to associate with the team name. We should therefore store a list of players under each key, as follows:

```
players = []
players["Team Red"] = ["Shaoming", "Lijin"]
players["Team Green"] = ["Obi", "Chinara"]
players["Team Blue"] = ["Mateo", "Sophia"]
```

The values in a dictionary aren't limited to being basic values. They can be arbitrarily complex, including lists, tables, or even other dictionaries (and more!). There is still only one value per key, which is the requirement of a dictionary.

9.2.4 Dictionaries versus Dataclasses Previously, we learned about dataclasses as a way to create compound data in Python. Here again is the `ToDoItem` dataclass that we introduced earlier, as well as an example datum for that class:

```
class ToDoItem:  
    descr: str  
    due: date  
    tags: list  
  
milk = ToDoItem("buy milk", date(2020, 7, 27), ["shopping", "home"])
```

One could view the field names in the dataclass as akin to keys in a dictionary. If we did so, we could also capture the `milk` datum via a dictionary as follows:

```
milk_dict = {"descr": "buy milk",  
             "due": date(2020, 7, 27),  
             "tags": ["shopping", "home"]}  
}
```

Do Now!

Create a dictionary to capture the compound datum
`ToDoItem("grade hwk", date(2020, 7, 27), ["teaching"])`

Do Now!

Create a to-do list named `myTD_L` that contains a list of dictionaries, rather than a list of dataclasses.

Putting these two approaches side-by-side, here's the contrast:

```
myTD_L = [ToDoItem("buy milk", date(2020, 7, 27), ["shopping", "home"]),  
          ToDoItem("grade hwk", date(2020, 7, 27), ["teaching"]),  
          ToDoItem("meet students", date(2020, 7, 26), ["research"])]
```

```
myTD_D = [milk_dict,  
          {"descr": "grade hwk",  
           "due": date(2020, 7, 27),  
           "tags": ["teaching"]},  
          {"descr": "meet students",  
           "due": date(2020, 7, 26),  
           "tags": ["research"]}]
```

Do Now!

What do you see as the benefits and drawbacks of each of dataclasses and dictionaries to represent compound data?

Dataclasses have a fixed number of fields, while dictionaries allow arbitrary numbers of keys. Dataclass fields can be annotated with types (which most languages will check when you make new data); dictionaries can use fixed types for each of keys and values, though this gets restrictive when using dictionaries to capture dataclasses with fields of different types. Dataclasses give you a function name for creating new data, whereas with dictionaries you'd have to create such a function on your own.

Overall, dataclasses come with more linguistic support for error checking: you can't supply data for the wrong number of fields or field values of the wrong type. Dictionaries are more flexible: you can support optional fields more easily, including adding new fields/keys as a program runs. Each of these makes more sense in some programming situations.

Do Now!

Write a function `ToDoItem_D` that takes a description, due date, and list of tags and returns a dictionary with keys for each field of a to-do item.

Summary Python programmers tend to make substantial use of dictionaries. In this chapter, we've seen dictionaries used in two different settings:

- one in which the keys uniquely identify different entities or individuals among a larger set; the values represent some consistent type of information about each individual. The dictionary overall captures information about a large population of individuals, each with their own key.
- one in which the keys name fields of compound data; the values associated with each field can have different types from the values for other fields. This setting corresponds to the use of dataclasses, in which a dictionary captures information about one individual; some other structure (such as a list or another dictionary) would be needed to hold the dictionaries for each individual.

As a general rule, it is better to use dataclasses for the second setting when you have a fixed set of fields. The use of dictionaries for dataclasses is somewhat associated with programming practices in the Python community (less so in other languages). The first setting, however, is a common use of dictionaries in nearly all languages, especially since dictionaries are usually built to provide fast access to the data associated with a specific key.

contents ← prev up next →

9.3 Arrays

9.3 Arrays We ended the last chapter with a question about how fast one can access a specific element of a list. Specifically, if you have a list called `finishers` of Runners (our example from last time) and you write:

```
finishers[9]
```

How long does it take to locate the Runner in 10th place (remember, indices start at 0)?

It depends on how the list is laid out in memory.

9.3.1 Two Memory Layouts for Ordered Items When we say "list", we usually mean simply: a collection of items with order. How might a collection of ordered items be arranged in memory? Here are two examples, using a list of course names:

```
courses = ["CS111", "ENGN90", "VISA100"]
```

In the first version, the elements are laid out in consecutive memory locations (this is roughly how we've shown lists up to now):

Prog	Directory	Memory
courses -->	loc 1001	loc 1001 --> [loc 1002, loc 1003, loc 1004]
		loc 1002 --> "CS111"
		loc 1003 --> "ENGN90"
		loc 1004 --> "VISA100"

loc 1001 -->	loc 1001 --> [loc 1002, loc 1003, loc 1004]
	loc 1002 --> "CS111"
	loc 1003 --> "ENGN90"
	loc 1004 --> "VISA100"

In the second version, each element is captured as a datatype containing the element and the next list location. When we were in Pyret, this datatype was called `link`.

Prog	Directory	Memory
courses -->	loc 1001	loc 1001 --> link("CS111", loc 1002)
		loc 1002 --> link("ENGN90", loc 1003)
		loc 1003 --> link("VISA100", loc 1004)
		loc 1004 --> empty

loc 1001 -->	loc 1001 --> link("CS111", loc 1002)
	loc 1002 --> link("ENGN90", loc 1003)
	loc 1003 --> link("VISA100", loc 1004)
	loc 1004 --> empty

What are the tradeoffs between the two versions? In the first, we can access items by index in constant time, as we could for hashmaps, but changing the contents (adding or deleting) requires moving things around in memory. In the second, the size of the collection can grow or shrink arbitrarily, but it takes time proportional to the index to look up a specific value. Each organization has its place in some programs.

In data structures terms, the first organization is called an array. The second is called a linked list. Pyret implements linked lists, with arrays being a separate data type (with a different notation from lists). Python implements lists as arrays. When you approach a new programming language, you need to look up whether its lists are linked lists or arrays if you care about the run-time performance of the underlying operations.

Going back to our Runners discussion from the last chapter, we can simply use Python lists (arrays) rather than a hashtable, and be able to access the names of Runners who finished in particular positions. But let's instead ask a different question.

How would we report the top finishers in each age category? In particular, we want to write a function such as the following:

```
def top_5_range(runners: list, lo: int, high: int) -> list:  
    """get list of top 5 finishers with ages in  
    the range given by lo to high, inclusive  
    """
```

Think about how you would write this code.

Here's our solution:

```
def top_5_range(runners: list, lo: int, high: int) -> list:  
    """get list of top 5 finishers with ages in  
    the range given by lo to high, inclusive  
    """  
  
    # count of runners seen who are in age range  
    in_range: int = 0  
    # the list of finishers  
    result: list = []  
  
    for r in runners:  
        if lo <= r.age and r.age <= high:  
            in_range += 1  
            result.append(r)  
        if in_range == 5:  
            return result  
    print("Fewer than five in category")  
    return result
```

Here, rather than return only when we get to the end of the list, we want to return once we have five runners in the list. So we set up an additional variable (`in_range`) to help us track progress of the computation. Once we have gotten to 5 runners, we return the list. If we never get to 5 runners, we print a warning to the user then return the results that we do have.

Couldn't we have just looked at the length of the list, rather than maintain the `in_range` variable? Yes, we could have, though this version sets up a contrast to our next example.

9.3.2 Iterating Partly through an Ordered Datum What if instead we just wanted to print out the top 5 finishers, rather than gather a list? While in general it is usually better to separate computing and displaying data, in practice we do sometimes merge them, or do other operations (like write some data to file) which won't return anything. How do we modify the code to print the names rather than build up a list of the runners?

The challenge here is how to stop the computation. When we are building up a list, we stop a computation using `return`. But if our code isn't returning, or otherwise needs to stop a loop before it reaches the end of the data, what do we do?

We use a command called `break`, which says to terminate the loop and continue the rest of the computation. Here, the `break` is in place of the inner `return` statement:

```
def print_top_5_range(runners: list, lo: int, high: int):  
    """print top 5 finishers with ages in  
    the range given by lo to high, inclusive  
    """  
  
    # count of runners seen who are in age range
```

```
in_range: int = 0

for r in runners:
    if lo <= r.age and r.age <= high:
        in_range += 1
        print(r.name)
    if in_range == 5:
        break
print("End of results")
```

If Python reaches the `break` statement, it terminates the for loop and goes to the next statement, which is the `print` at the end of the function.

contents ← prev up next →

10.1 Introduction to Pandas

10.1 Introduction to Pandas Now it's time to transfer what we learned about tables in Pyret over to Python. Pandas is a popular package, and you'll find many tutorial and help sites for it online. In general, Python usually provides many ways to approach a given task. As such, there are many ways to do common operations in Pandas. We have chosen to present a certain collection of ways that align with the concepts as we covered them in Pyret.

To work in Pandas, you'll need to include the following line at the top of your file:

```
import pandas as pd
```

10.1.1 Pandas Table Basics

10.1.1.1 Core Datatypes: DataFrame and Series Pandas uses the term DataFrame for a table with rows and columns. DataFrames are built out of two more basic types:

- An array is a sequence of values that can be accessed by position (e.g., 0, 1, ... up to one less than the length of the array). Like lists, arrays capture a linear (ordered) collection of values. Unlike lists, arrays are created with a limit on the number of elements that they contain. In practice, lists are more commonly used when elements are frequently added or removed whereas arrays are more commonly used when elements frequently get accessed by their position. Nearly every programming language offers both lists and arrays; a detailed contrast is beyond the scope of this book (this information would be covered in a data structures class).
- A Series is an array in which the positions optionally have labels in addition to the position numbers.

In Pandas, a row is a Series in which an array of the cell values is labeled with the column headers (this is similar to the 'Row' datatype in Pyret). A DataFrame is a series of these rows.

10.1.1.2 Creating and Loading DataFrames DataFrames can be created manually or loaded in from a file, as we did in Pyret. Here's a simple example of creating one by hand:

```
data = {  
    'day': ['mon', 'tues', 'wed', 'thurs', 'fri'],  
    'max temp': [58, 62, 55, 59, 64]  
}  
temp = pd.DataFrame(data)
```

`data` is a dictionary that maps column names to values. Calling `pd.DataFrame` creates a DataFrame from the dictionary. (There are other ways to create DataFrames manually which you can find by searching online.)

To load a DataFrame from a CSV file, you need either the path to the file on your computer or the url where you can get the CSV file online. Here's an example of the url version. In this example, we have the following CSV contents and we want to change the header names when loading the file:

The following `read_csv` command says that the CSV file is at `url`, that there are headers in the first row (numbered 0), and that we want to use the values in `names` as the column labels (this will ignore whatever might be in the header row in the CSV file).

```
events_url = "https://raw.githubusercontent.com/data-centric-computing/dcic-public/main/materials/datasets/  
events = pd.read_csv(events_url, header=0,  
                     names=['name', 'email', 'numtix', 'discount', 'delivery'])
```

If we wanted to use the headers in the CSV file as the column headers, we would leave out the `names=[...]` part. If the CSV had no header row, we would write `header=None` instead of `header=0`. (There are many more configuration options in the Pandas documentation, but you won't need them for the examples in this book.)

Conceptually, the loaded DataFrame is as follows, with the labels shown in blue and the indices (positions) show in yellow:

indices/positions	0	1	2	3	4
labels	<code>name</code>	<code>email</code>	<code>numtix</code>	<code>discount</code>	<code>delivery</code>
0	Josie Zhao	jo@mail.com	2	BIRTHDAY	email
1	Sam Ochibe	s@sweb.com	1		pickup
2	Bart Simple	bart@simpson.org	5	STUDENT	yes
3	Ernie O'Malley	ernie.mail.com	0	none	email
4	Alvina Velasquez	alvie@schooledu	3	student	email
5	Zander	zandaman	10		email
6	Shweta Chowpatti	snc@this.org	three		pickup

Since we did not specify labels for the rows, Pandas has used numeric labels by default. At the moment, the positions and the labels are the same for each row, but we will see that this is not always the case.

(If you look at the actual loaded table, some of the blank cells in the discount column will contain `NaN`, which is the standard Python value for “missing information”. We will deal with that information shortly.

10.1.1.3 Using Labels and Indices to Access Cells Rows, columns, and cells can be accessed using either their (numeric) positions or their labels. Here are some examples:

```
events['numtix']           # extract the numtix column as a series
events['numtix'][2]         # get the value in the numtix column, row 2
events.loc[6]               # extract row with label 6 from the DataFrame
events.loc[6]['numtix']     # get the value in row with label 6, numtix column
events.iloc[6]              # extract the row with index/position 6
```

Notice that we used different notation for accessing a cell depending on whether we accessed the row first or the column first. This is because we are showing you how to access data through either position indices or labels. Using `.loc` tells Pandas that you are using a label to access a row. If you want to use the position instead, you need to use `iloc` (the `i` stands for “integer”). If you are using a programmer-supplied label instead, you can just use the label directly.

In a DataFrame, both rows and columns always have position indices and may have labels. The `.loc` notation works on either rows or columns, we just happened to illustrate the notation on the rows since we had already created labels on the columns when we loaded `events`.

10.1.2 Filtering Rows Back in Pyret, we filtered rows from a table by writing a function from `Row` to `Boolean`. The `filter-with` function applied that function to every row in the table, returning a new table with those rows for which the predicate were true.

In Pandas, we select rows by providing an array of Booleans that has the same length as the number of rows in the DataFrame. Filtering keeps those rows for which the corresponding array entry is `True`. For example, here's our DataFrame diagram from before, this time with an array to the right indicating that we want to keep rows 0, 2, and 6.

indices/positions		0	1	2	3	4	
	labels	name	email	numtix	discount	delivery	Keep
0	0	Josie Zhao	jo@mail.com	2	BIRTHDAY	email	True
1	1	Sam Ochibe	s@sweb.com	1		pickup	False
2	2	Bart Simple	bart@simpson.org	5	STUDENT	yes	True
3	3	Ernie O'Malley	ernie.mail.com	0	none	email	False
4	4	Alvina Velasquez	alvie@schooledu	3	student	email	False
5	5	Zander	zandaman	10		email	False
6	6	Shweta Chowpatti	snc@this.org	3		pickup	True

The “keep” array is not part of the DataFrame. Here is the corresponding array expressed in code, followed by the notation to use the array to filter the DataFrame:

```
# which rows we want
keep = [True, False, True, False, False, False, True]
```

Once we have the array of booleans, we use it to extract a collection of rows using similar notation that we previously used to extract a column. Just as we wrote `events['numtix']` to select the 'numtix' column, we can write `events[keep]` to select a collection of rows. The DataFrame that results from filtering (along with the True cells of the keep array for illustration) appears as follows:

indices/positions		0	1	2	3	4	
	labels	name	email	numtix	discount	delivery	Keep
0	0	Josie Zhao	jo@mail.com	2	BIRTHDAY	email	True
1	2	Bart Simple	bart@simpson.org	5	STUDENT	yes	True
2	6	Shweta Chowpatti	snc@this.org	3		pickup	True

How does Pandas know whether we want to select rows or columns? It depends on what we provide in the square brackets: if we provide a single label, we get the column or row with that label; if we provide an array of booleans, we get the rows for which the corresponding row (by position) is `True`.

Do Now!

Look at the returned DataFrame. Do you notice anything interesting?

Look at the row labels and indices: the labels have been retained from the original DataFrame (0, 2, and 6), while the indices are a sequence of consecutive numbers starting from 0. Having both ways to reference rows—one based on raw order and the other based on programmer-provided labels—provides a lot of flexibility as we use filter to isolate parts of tables that we want to work on.

Do Now!

Does filtering rows this way in Python keep the original `events` DataFrame intact? Try it out!

Arrays of booleans that are used for filtering out other arrays are called masks. Here, we have shown a simple mask that we constructed by hand. If we had a long DataFrame, however, we would not want to construct a mask for it by hand. Fortunately, we don't have to. Python provides notations that let us construct masks via expressions over a series.

Imagine that we wanted to filter the `events` table down to those rows with delivery method 'email'. To create a mask for this, we first select the delivery column as a series:

```
events['delivery']
```

Next, we use the series in a boolean expression that states the constraint that we want on each element of the series:

```
events['delivery'] == 'email'
```

Wait, what's going on here? `events['deliver']` is a Series (a labeled array of strings). 'email' is a string. What does it even mean to ask whether two values of different types be considered equal, especially when one has many component values and the other does not?

In this case, the `==` doesn't mean "are these equal"? Instead, Python applies `== 'email'` to every element of the `events['delivery']` Series, constructing a new Series of the results. This idea of applying an operation to all elements of an array is known as "lifting". It is one of the shortcuts that Python provides to help experienced programmers do simple common tasks quickly and easily.

Now that we have a Series of booleans (for which events will be picked up by email), we can use it to select those rows from the `events` DataFrame:

```
events[events['delivery'] == 'email']
```

The inner use of `events` is for creating the mask, while the outer one is for filtering the table with that mask.

As a warning: if you search online for information on how to filter or process DataFrame, you might find code samples that do this using for loops. While that approach works, it isn't considered good Pandas (or general programming) practice. Most modern languages provide built-in constructs for iterating over lists and other sequence-style data. These operations have more descriptive names than generic loops (which makes them easier for other programmers to read), and are often engineered to run more efficiently under the hood. As a general rule, only default to basic loops if there is no built-in operator to do the computation that you have in mind.

10.1.3 Cleaning and Normalizing Data The same operator-lifting idea that we just saw when creating masks from DataFrames also comes into play for normalizing data. Recall that when we worked with the `events` table in Pyret, we converted all of the discount codes to lowercase. Here's the code that does this in Pandas:

```
events['discount'] = events['discount'].str.lower()
```

Do Now!

Look at the above code. Break it down and try to articulate what each part does. Do any parts seem new or different from things we've done so far in Pandas?

On the right side of the `=`, we are extracting the Series of discount codes (`events['discount']`), then using the lowercase operation on strings `str.lower()` to convert each one, building up a Series of the results. Normally, given a string (such as `'BIRTHDAY'`), we could get a lowercase version of it by writing just `'BIRTHDAY.lower()'`. What's the extra `str` doing in there?

This is a nuance about lifting. Python can evaluate `'BIRTHDAY.lower()` because `lower()` is defined directly on strings. `lower()` is not, however, directly defined on Series. To bridge the gap between having Series data and wanting to use a string operation on it, we insert `str` before `lower()`. Effectively, this tells Python where to find the `lower()` operation (in the collection of operations defined on strings).

The left side of the above code looks like:

```
events['discount'] = ...
```

This tells Pandas to replace the current contents of the `'discount'` series with the series on the right side of the `=`. It is similar to `transform-column` from Pyret, but with a fundamental difference: in Pyret, `transform-column` left the old table intact and produced a new table with the new column values. Instead, in Pandas the old column gets replaced, thus destroying the original table. There are many nuances to having operations destroy and replace data; the chapter on State, Change, and Testing studies them in detail.

10.1.3.1 Clearing out unknown values Now let's try a different cleaning and normalization problem: we want the discount column to contain only known discount codes or empty strings. The `none` entry in line 3 of the table should be converted to an empty string, and we should make sure that all of the `NaN` and seemingly empty entries in the discount cells are also converted to empty strings (as opposed to strings of multiple spaces).

Do Now!

Plan out how you might do this task using mask expressions. Even if you don't know all the specific notation for the operations you need, you can still work out a plan for completing this task.

If you planned out the tasks, you might have a todo list like the following:

1. create a mask of rows with known discount codes

2. invert that mask (swap the false and true values)
3. filter the DataFrame to rows without a known discount code
4. replace all the discount column values in that DataFrame with an empty string

We have seen how to do parts of steps 1 and 3, but neither of steps 2 and 4. Let's work through the steps one by one:

Here's the code for step 1, which creates a mask for the rows with known discount codes:

```
codes = ['birthday', 'student']      # a list of valid codes
events['discount'].isin(codes)       # which rows have valid codes
```

Here, we use a lifted `isin` operator on lists to compute the mask.

For step 2, we have to swap the true and false values. We can do this by using the negation operator `~` on the mask from step 1:

```
~events['discount'].isin(codes)
```

For step 3, we want to filter `events` with this mask. Just to keep the code easier to read, we'll give the mask a name and then perform the filter:

```
mask = ~events['discount'].isin(codes)    # rows with INVALID codes
events[mask]
```

Finally, we use `=` to set the discount column of the filtered DataFrame to the empty string:

```
events[mask]['discount'] = ''
```

Whoops – this seems to have generated an error message that says something about a “SettingWithCopyWarning”. This is a subtlety that we will return to later, after we learn more about what happens when we update data contents. For now, we'll use this alternate form that avoids the error:

```
events.loc[mask, 'discount'] = ''
```

Putting it all together, the entire program looks like:

```
codes = ['birthday', 'student']
mask = ~events['discount'].isin(codes)
events[mask]['discount'] = ''
```

Summarizing, the code pattern for updating values for a column in some rows of a DataFrame is as follows:

- make a boolean series mask for which rows to update
- use the mask to select just the rows where the mask is true
- use `.loc` with the mask and column name to select the series of cells to update
- use `=` to give those cells their new value

Exercise

Follow the above pattern to transform all delivery values of 'yes' to 'pickup'.

10.1.3.2 Repairing Values and Column Types The source file for the `events` table contained an error in which someone entered the string 'three' in place of the number 3 for the number of tickets in the last row. We can repair errors like this manually:

```
events.loc[6]['numtix'] = 3
```

Do Now!

Make this repair and ask your Python environment to show you the corrected table.

Now that the 'numtix' column contains only numbers, we can total the number of tickets that were sold:

```
events['numtix'].sum()
```

Do Now!

What did you get? Why?

Because Python environments print strings without quotation marks, the numtix column appears to contain numbers. The failure of `sum` shows that this is indeed not the case. We can inspect the types that Python has determined for the numtix values using the `type` operation:

```
type(events['numtix'].loc[0]) # prints str  
type(events['numtix'].loc[6]) # prints int for the corrected value
```

What happened here? During the original call to `read_csv`, Python detected both numeric and string data in the numtix column. It therefore read in all the values as strings. Our manual repair that replaced the string '`'three'`' with the number `3` fixed the value and type for one row, but the remaining values in that column have still been read in as integers.

Fortunately, Python provides an operation to change the type of data within a series. The following code converts the values in the `events['numtix']` series to integers, updating the series within the DataFrame in the process.

```
events['numtix'] = events['numtix'].astype('int')  
  
events['numtix'].sum() # now this works
```

10.1.4 Computing New Columns Let's extend the events table with the total cost of tickets, while also accounting for a discount. We'll start by building a column for the ticket price without any discounts. This is a straightforward application of lifting as we've seen it so far:

```
ticket_price = 10  
events['total'] = events['numtix'] * ticket_price
```

Do Now!

Use masks, operator lifting, filtering, and series updating to give a 10% discount to everyone with the "birthday" discount code.

We do this by creating a mask for the "birthday" discount, then updating just that part of the DataFrame.

```
bday_mask = events['discount'] == 'birthday'  
events.loc[bday_mask, 'total'] = events['total'] * 0.90
```

Notice that the notation for computing new columns and updating existing ones is the same (unlike in Pyret, where we had different operations `build-column` and `transform-column`). In Pandas, a new column is created if the given column name doesn't already exist in the DataFrame; otherwise, the existing column with the given name gets updated.

10.1.5 Aggregating and Grouping Columns Pandas has built-in operations for doing standard mathematical computations over series. For example, to total the number of tickets sold or to compute the average number of tickets per order, we can write

```
events['numtix'].sum() # compute total number of tickets sold  
events['numtix'].mean() # compute average number of tickets per sale
```

These are the same built-in operations that apply to Python lists.

Imagine now that we wanted a finer-grained look at total ticket sales. Rather than just the total sold overall, we'd like the total sold per discount category.

Do Now!

How might you compute this?

We could imagine constructing a list of the discount codes, filtering the ticket sales table to each code, then using `sum` on each filtered table. This feels like a lot of work, however. Producing summaries of one column (e.g., ```numtix''`) around the values in another (e.g., ```discount''`) is a common technique in data analysis. Spreadsheets typically provide a feature called a "pivot table" that supports such a view of data.

In Pandas, we can do a computation like this using an operation called `groupby`. Here's are two examples. The first reports how many sales (rows) were made with each discount code, while the second summarize the total number of tickets sold by discount code:

```
events.groupby('delivery').count()
events.groupby('discount')['numtix'].sum()
```

`groupby` takes the name of the column whose values will be used to cluster rows. It returns a special type of data (called `GroupBy`). From there, we can select a column and perform an operation on it. The column selection and operation are performed on each collection of rows in the `GroupBy`. The results of the second expression in the above code are reported in a new DataFrame:

discount

	14
birthday	2
student	8

In this DataFrame, `discount` labels a column. The first row has the empty string in the discount column, with 14 tickets purchased without discount codes. There were 2 tickets purchased with a birthday discount and 8 with a student discount.

The Pandas documentation provides a large collection of operations that can be used on `GroupBy` data; these cover computations such as counting, mean, finding largest and smallest values, and performing various other statistical operations.

10.1.6 Wide Versus Tall Data Let's try grouping data on a different dataset. Here's a table showing sales data across several regions during each month of the year:

month	division	northwest	northeast	central	southeast	southwest
Jan	medical	125	91	78	83	93
Feb	medical	89	70	50	74	117
Mar	medical	120	123	85	121	118
Apr	medical	95	70	76	53	94
May	medical	124	77	88	50	124
Jun	medical	76	90	51	98	125
Jul	medical	122	87	120	77	122
Aug	medical	83	119	122	102	106
Sep	medical	121	94	117	54	103
Oct	medical	77	67	79	65	101
Nov	medical	80	92	115	100	98
Dec	medical	124	120	114	87	105

Copy the following code to load this table for yourself.

```
import pandas as pd

sales_url = "https://raw.githubusercontent.com/data-centric-computing/dcic-public/main/materials/datasets/wide.csv"
col_names = ['month','division','northwest','northeast','central','southeast','southwest']
sales = pd.read_csv(sales_url, header=0, names=col_names)
```

Do Now!

Here are several questions that we might want to ask from this dataset. For each one, develop a plan that indicates which Pandas operations you would use to answer it. If a question seems

hard to answer with the operations you have, explain what's difficult about answering that question.

1. In which month did the northwest region have the lowest sales?
2. What were the total sales per month across all regions?
3. Which region had the highest sales in April?
4. Which region had the highest sales for the entire year?

For question 1, we can sort the table by northwest sales in decreasing order, then see which month is listed in the first row.

```
s = sales.sort_values('northwest', ascending=True)
s.iloc[0]['month']
```

Do Now!

What value would we have gotten had we used `loc` instead of `iloc` in the above code?

Do Now!

Did sorting the `sales` table change the row order permanently? Check by having Python show you the value of `sales` after you run `sort_values`.

For question 2, we could build a new column that stores the sales data across each row:

```
# we use parens around the right side to break the expression across
# multiple lines, rather than extend past the window edge
sales['total'] = (sales['northwest'] + sales['northeast'] +
                   sales['central'] + sales['southeast'] +
                   sales['southwest'])
```

Do Now!

Did computing the `total` column change the row order permanently? Check by having Python show you the value of `sales` after you run the code.

(If you want to remove the new `total` column, you can do this with `sales = sales.drop(columns='total')`.)

Question 3 is more challenging because we want to sort on the regions, which are in columns rather than rows. Question 4 is even more challenging because we want to produce sums of columns, then compare regions. Both of these feel a bit like problems we might know how to solve if the rows corresponded to regions rather than months, but that isn't how our data are organized. And even if we did flip the table around (we could, the technical term for this is `transpose`), problem 4 would still feel a bit complicated by the time we computed annual sales per region and sorted them.

What if instead our table had looked like the following? Would questions 3 and 4 get any easier?

month	division	region	sales
Jan	medical	northwest	125
Jan	medical	northeast	91
Jan	medical	central	78
Jan	medical	southeast	83
Jan	medical	southwest	93
Feb	medical	northwest	89
Feb	medical	northeast	70
Feb	medical	central	50
Feb	medical	southeast	74
Feb	medical	southwest	117
...

With the data organized this way, question 3 can be answered with a combination of row selection and `sort_values`. Question 4 becomes easy to answer with a `groupby`. Even the code for Question 2 gets cleaner.

The contrast between these two tables highlights that how our data are organized can determine how easy or hard it is to process them with the standard operations provided by table-processing packages such as Pandas (what we're discussing here applies to other languages that support tables, such as Pyret and R).

In general, the operations in table-processing packages were designed to assume that there is one core observation per row (about which we might have many smaller details or attributes), and that we will want to aggregate and display data across rows, not across columns. Our original treated each month as an observation, with the regions being details. For questions 1 and 2, which focused on months, the built-in operations sufficed to process the table. But for questions 3 and 4, which focused on regions or combinations of regions and months, it helps to have each month and region data be in its own row.

Tables like the original `sales` data are called wide tables, whereas the second form are termed tall tables. At the extremes, wide tables have every variable in its own column whereas tall tables have only one column for a single value of interest, with a separate row for each variable that contributed to that value. Wide tables tend to be easier for people to read; as we have seen with our sales data, tall tables can be easier to process in code, depending on how our questions align with our variables.

Converting Between Wide and Tall Data Table-processing packages generally provide built-in operators for converting between wide and tall data formats. The following Pandas expression converts the (original) wide-format `sales` table into a tall-format table, retaining the month of the year and the product division as a label on every datapoint:

```
sales.melt(id_vars=['month', 'division'])
```

This basic `melt` expression uses default column names of `variable` and `value` for the new columns. We can customize those names as part of the `melt` call if we wish:

```
sales_tall = sales.melt(id_vars=['month', 'division'], var_name='region', value_name='sales')
```

Let's put the wide and tall tables side by side to visualize what `melt` is doing.

month	division	northwest	northeast	central	southeast	southwest		month	division	region
Jan	medical	125	91	78	83	93		Jan	medical	northwest
Feb	medical	89	70	50	74	117		Jan	medical	northeast
Mar	medical	120	123	85	121	118		Jan	medical	central
Apr	medical	95	70	76	53	94		Jan	medical	southeast
May	medical	124	77	88	50	124		Jan	medical	southwest
Jun	medical	76	90	51	98	125		Feb	medical	northwest
Jul	medical	122	87	120	77	122		Feb	medical	northeast
Aug	medical	83	119	122	102	106		Feb	medical	central
Sep	medical	121	94	117	54	103		Feb	medical	southeast
Oct	medical	77	67	79	65	101		Feb	medical	southwest
Nov	medical	80	92	115	100	98
Dec	medical	124	120	114	87	105				

The columns named in `id_vars` remain in the original table. For each column not named in `id_vars`, a row is created with the `id_vars` columns, the melted-column name, and the melted-column value for the `id_vars`. The above figure color codes how cells from the wide table are arranged in the melted tall table.

With the tall table in hand, we can proceed to answer questions 3 and 4, as well as to redo our solution to question 2:

```
# Question 2: total sales per month across regions
sales_tall.groupby('region').sum()
```

```
# Question 3: which region had the highest sales in April
apr_by_region = sales_tall[sales_tall['month'] == 'Apr']
```

```

apr_by_region.sort_values('sales', ascending=False).iloc[0]['region']

# Question 4: which region had the highest sales for the year
tot_sales_region = sales_tall.groupby('region').sum()
tot_sales_region.sort_values('sales', ascending=False).reset_index().iloc[0]['region']

```

The solution to question 4 uses a new Pandas operator called `reset_index`, which is needed if you want to manipulate the output of a group-by as a regular DataFrame.

10.1.7 Plotting Data

Let's continue with the sales data as we explore plotting in Pandas.

Let's say we now want to take a seasonal view, rather than a monthly view, and look at sales within seasons.

Let's say we wanted to see how summer sales varied over the years. This is a good situation in which to use a line plot. To create this, we first need to load `matplotlib`, the Python graphic library:

```

import matplotlib.pyplot as plt
from Pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()

```

Next, to generate the line plots, we call the `plt.plot` function on the series of numbers that we want to form the points on the plot. We can also specify the values on the axes, as shown in the following examples.

```

# create a new canvas in which to make the plot
plt.figure()

# plot month column (x-axis) vs northeast sales (y-axis)
plt.plot(sales['month'], sales['northeast'])

# add central sales to the same plot
plt.plot(sales['central'])

# add labels to the y-axis and the chart overall
plt.ylabel('Monthly Sales')
plt.title('Comparing Regional Sales')

# show the plot
plt.show()

```

Pandas will put both line plots in the same display window. In general, each time you call `plt.figure()`, you create a new window in which subsequent plot commands will appear (at least until you ask for a plot that does not nicely overlay with the previous plot type).

The `matplotlib` package offers many kinds of charts and customizations to graph layouts. A more comprehensive look is beyond the scope of this book; see the `matplotlib` website for tutorials and many examples of more sophisticated plots.

10.1.8 Takeaways

This chapter has been designed to give you an overview of Pandas while pointing out key concepts in programming for data science. It is by no means a comprehensive Pandas tutorial or reference guide: for those, see the Pandas website.

Conceptually, we hope you will take away three high-level ideas from this chapter:

- There are two notions for how to access specific cells in tables and DataFrames: by numeric position (e.g., first row, second column) or by labeled index (e.g., `numtix`). Both have their roles in professional-grade data analysis programming. Filter-like operations that extract rows from tables maintain labeled indices, but renumber the positional ones (so that every DataFrame has a sequence of consecutively-numbered rows).
- Professional-grade programming languages sometimes “lift” operations from single values to collections of values (e.g., using `+` to add elements within similarly-sized series). Lifting can be a powerful and timesaving tool for programmers, but they can also lead to type confusions for both novices and experienced programmers. You should be aware that this feature exists as you learn new languages and packages.

- Different table organizations (for the same data) are better in different situations. Wide and tall tables are two general shapes, each with their own affordances. You should be aware that table-processing packages provide a variety of tools to help you automatically reformat tables. If the computation you are trying to do feels too complicated, stop and consider whether the problem would be easier with a different organization of the same data.

contents [←](#) [prev](#) [up](#) [next](#) [→](#)

10.2 Reshaping Tables

10.2 Reshaping Tables

10.2.1 Binning Rows

10.2.2 Wide versus Tall Datasets

contents ← prev up next →

11.1 State, Change, and Testing

11.1 State, Change, and Testing We will now study a new kind of data and the programming style that accompanies it. This will give us both great power and great responsibility. We will develop this idea in both Pyret and Python, both because the core concept arises in both (indeed in nearly all) languages and because their contrast is instructive.

11.1.1 Example: Bank Accounts Imagine that we want to represent bank accounts, where each account has a (unique) id number and a balance:

Python

Pyret

```
@dataclass
class Account:
    id: int
    balance: float

data Account:
    account(id :: Number,
            balance :: Number)
end
```

Let's now make an account:

Python

```
acct1 = Account(8404, 500)
acct1 = account(8404, 500)
```

Now let's say we learn that the account has just earned another 200. We could always reflect the resulting account as follows:

Python

```
Account(acct1.id, acct1.balance + 200)
```

Pyret

```
account(acct1.id, acct1.balance + 200)
```

However, this creates a new account; if we look at the current `balance` of `acct1`, by writing `acct1.balance`, it is still 500. If this were our account, we would be quite sad!

Rather, we want to change the balance in the existing account. This requires a programming feature that we have not encountered until now: data that can be changed. Such data are called mutable, and we explore them below. In contrast, until now we have worked with immutable data: data that cannot be altered.

First, we have to declare that the data can be changed. In Python, this is automatically true, always, so nothing changes. In Pyret, however, fields cannot be changed—they are immutable—by default. We have to explicitly say they can be changed:

Python

Pyret

```
@dataclass
class Account:
    id: int
    balance: float

data Account:
    account(id :: Number,
            ref balance :: Number)
end
```

This Pyret definition says that `id` cannot be changed, while `balance` can. This ensures that no programmer can accidentally change the bank account number. In Python, every programmer has to make sure they don't accidentally change it. (If we did want `id` to be mutable in Pyret, we would add a `ref` in front of it, too.)

With this definition, making accounts looks the same (unsurprisingly in Python, since nothing has changed):

Python

Pyret

```
acct1 = Account(8404, 500)
acct1 = account(8404, 500)
```

When we view the account in Pyret, we see something special:

```
>>> acct1
account(8404, [500])
```

The yellow-and-black “caution tape” indicator is a reminder that the value can change, so what is shown on screen may not be the current value.

Accessing an immutable field in Pyret remains the same:

Python

Pyret

```
acct1.id
acct1.id
```

However, accessing a mutable field looks different in Pyret:

Python

Pyret

```
acct1.balance
acct1!balance
```

The `!` is there to remind that what you are getting is the current value of `balance`, and it may be different later. Python does not offer a similar syntactic warning, but then again, recall that every field is always mutable.

So now let's see how to change that account balance. For simplicity, let's first see how to set the account balance to zero. We use slightly different syntaxes for it in the two languages:

Python

Pyret

```
acct1.balance = 0
```

```
acct1!{balance: 0}
```

In Pyret, again, we use `!` in the syntax for changing the field: read it as “change the value now!”

Do Now!

You now know all the parts you need to figure out how to set `balance` to be 200 more than its previous value. Can you figure out how to write that?

Here's how we combine the pieces—accessing the value and then setting it:

Python

```
acct1.balance = acct1.balance + 200
```

Pyret

```
acct1!{balance: acct1!balance + 200}
```

While Pyret's syntax is a little more onerous for changing the value of one field, it proves to be lighter-weight if we want to change multiple fields. In Python we'd have to write `acct1.` for each of them, whereas in Pyret we need only the one `acct1!`. So there is a trade-off between the two syntaxes.

We hadn't written any tests above. Suppose we had: already we might notice something a bit odd. Say we had written

Python

Pyret

```
def test_balance():
    assert acct1.balance == 500

check:
    acct1!balance is 500
end
```

This would pass before we performed the update, but fails after the update is performed. In Python, tests are run when we call the testing functions, which we typically do after loading the full file (either by running them at the prompt or by putting our tests in a separate file).

In Pyret, tests are run as if they were written at the very bottom of definitions. Therefore, even if the program looked like this in Pyret:

```
acct1 = account(8404, 500)
```

```
check:
    acct1!balance is 500
end
```

```
acct1!{balance: acct1!balance + 200}
```

the test fails. Alternatively, we can write

```
acct1 = account(8404, 500)
```

```
check:
    acct1!balance is 700
end
```

```
acct1!{balance: acct1!balance + 200}
```

and it passes, but not if we comment out the update.

In both languages, then, we see a new phenomenon: tests that are only sometimes true. This phenomenon is called state. There is a “state” (a collection of values for the defined names) in which the balance is 500, and another where it is 700. This is not merely limited to testing! Testing is just a reflection of what is going on in the program

as it runs. From now on, every programming instruction will run in some state, and its actions will depend on the other values in that state. If those values change, the same instruction—i.e., the same piece of program text—may produce different answers. This makes programming much harder, and we will have to get used to the subtleties that come along with it.

11.1.2 Testing Functions that Mutate Data Our example of adding funds to an account corresponds to making a deposit into a bank account. Let's turn our balance-updating expression into a function (named `deposit`) that takes the deposit amount as input. Then, we'll look at how to write tests for that function. First, the function definition:

Python

```
def deposit(ac: Account, amt: float):
    '''add amt to the account's balance'''
    ac.balance = ac.balance + amt
```

Pyret

```
fun deposit(ac :: Account, amt :: Number):
    doc: "add amt to the account's balance"
    ac!{balance: ac!balance + amt}
end
```

How do we test this?

In Python, this function does not return anything. In Pyret, the update operation does return the value being updated, but in a larger function we can't always assume that it will be the value returned. Therefore, we have to set up our test to assume otherwise.

In general, tests for functions that contain mutation need to have three to four parts:

1. Setup: set up the necessary values to provide the function.
2. Call: call the function.
3. Check: check that the function had the desired behavior.
4. Teardown: restore data to their expected state.

Python

Pyret

```
def test_deposit():
    # Setup
    a1 = Account(8200, 150)

    # Call
    deposit(a1, 100)

    # Check
    assert a1.balance == 250

check:
    # Setup
    a1 = account(8200, 150)

    # Call
    deposit(a1, 100)

    # Check
    a1!balance is 250
end
```

In this case we don't need to perform a Teardown step because we created data purely for testing the function. But if, for instance, we had run the test over a dataset whose values matter, we would need to restore the changes.

Similarly, the Setup phase needs to make sure that all data have the right values. Until now, once created, data did not change. But now, data may have been changed by some other mutations, and this may cause tests to fail. Therefore, the Setup phase requires not only creating necessary data but also setting the values of previously-created data to be what the test expects. (Again, note that in Python it is difficult to know which fields might have been changed, whereas in Pyret, we only have to reset the value of mutable fields.)

Exercise

Write tests for the following function that adds interest to an account balance:

Python

```
def add_interest(ac: Account):
    '''increases the account value by 2 percent'''
    ac.balance = ac.balance * 1.02
```

Pyret

```
fun add-interest(ac :: Account):
    doc: "increases the account value by 2 percent"
    ac!{balance: ac!balance * 1.02}
end
```

11.1.3 Aliasing Now let's suppose our bank allows accounts to be shared by multiple customers. We should thus separate information about customers from that of the account:

Python

```
@Dataclass
class Customer:
    name: str
    acct: Account

data Customer:
    cust(name :: String,
          acct :: Account)
end
```

Specifically, suppose we have two accounts (`acct1` and `acct2`), where `acct1` is owned jointly by Elena and Jorge:

Python

Pyret

```
acct1 = Account(8404, 500)
acct2 = Account(8405, 350)
elena = Customer("Elena", acct1)
jorge = Customer("Jorge", acct1)

acct1 = account(8404, 500)
acct2 = account(8405, 350)
elena = cust("Elena", acct1)
jorge = cust("Jorge", acct1)
```

Now let's say Elena earns an additional 150. We want to update the account to reflect this. How might we do it? First we have to access the account itself: `elena.acct` (in both languages). Then we would update it using the syntax above:

Python

Pyret

```
a = elena.acct
a.balance = a.balance + 150

a = elena.acct
a!{balance: a!balance + 150}
```

Sure enough, Elena's account will now have the value of 850 (the original 500, the bonus of 200, and now the extra 150):

Python

```
assert elena.acct.balance == 850
```

Pyret

```
check:
  elena.acct!balance is 850
end
```

Observe that in Pyret we use `.` to get the account but `!` to get the balance: a reminder that Elena's account will never change (the way we have defined the data structure), but that account's balance may and, indeed, does. Between the designs of Python and Pyret, there's a trade-off between convenience and precision.

The key question now is: what is Jorge's balance? Put differently, will this test pass or fail?

Python

```
assert jorge.acct.balance == 850
```

Pyret

```
check:
  jorge.acct!balance is 850
end
```

Or even more simply: what is the value of this program?

Python

Pyret

```
jorge.acct.balance
jorge.acct!balance
```

There are two very reasonable answers here:

1. Going by our prose, Jorge's account should also have 850, because that's what it means to "share" an account.
2. Going by the visible code, Jorge's account should still have 700, because the update was made through `elenा.acct`, not `jorge.acct`.

Do Now!

Run the above code and see what you get.

What you find is that the above test passes: Jorge's account also has 850. We say that `elenা.acct` and `jorge.acct` are aliases: they are two different "names" for the exact same datum.

This is not the first time we have had shared data. However, until now, it hasn't mattered that the data were aliased. But now that we have mutation, aliases matter: the balance in `jorge.acct` has changed even though we never made an explicit change using that name. It is as if `elenা.acct` exhibited spooky action at a distance.

Again, there is a linguistic difference here. Because all fields are mutable in Python, you have to always be on the alert for this. Because only `ref` fields are mutable in Pyret, you can be sure that fields accessed through `.` will never change in value over time or even if there are aliases, but those accessed through `!` might change over time (and via aliases).

11.1.4 Data Mutation and the Directory Now that we have the ability to mutate the contents of data, we will need to show and then revise our notion of directories. The directories are essentially the same between Pyret and Python, with one exception: we have different naming conventions in the two languages. For instance, we write `Account(8404, 500)` in Python versus `account(8404, 500)` in Pyret. It would be annoying to write every one of these twice, with the only difference being the capitalization. Therefore, where the only difference is the naming, we will ignore this difference and show only one version (in this case, the Python version); you should assume that the exact same thing is true for Pyret, other than the capitalization.

As a reminder, here are our initial definitions once again:

```
acct1 = Account(8404, 500)
acct2 = Account(8405, 325)
elena = Customer("Elena", acct1)
jorge = Customer("Jorge", acct1)
```

Do Now!

Review the following proposal for the directory contents after running the initial definitions. Is this what you expect to see?

Directory

- `acct1`
→
`Account(8404, 500)`
- `acct2`
→
`Account(8404, 500)`
- `elena`
→
`Customer("Elena", acct1)`
- `jorge`
→
`Customer("Jorge", acct1)`

There's a problem with this version, namely the use of `acct1` in the values associated with `elena` and `jorge`. Remember, the values in the directory can't refer to names in the directory: both Pyret and Python replace names with their values when evaluating expressions. Here is the corresponding version of the directory that uses the value of `acct1`:

Directory

- `acct1`
→
`Account(8404, 500)`
- `acct2`
→
`Account(8405, 325)`
- `elena`
→
`Customer("Elena", Account(8404, 500))`

- `jorge`
 -
 - `Customer("Jorge", Account(8404, 500))`

Observe that this is also what you would see if you were to evaluate the corresponding variable names.

Now, let's add funds to Elena's account:

Python

```
elena.acct.balance = elena.acct.balance + 150
```

Pyret

```
elena.acct!{balance: elena.acct!balance + 150}
```

Do Now!

Show how the directory changes if you run the above code.

If we follow the code precisely, we might expect the following directory, in which only the balance in Elena's version of the account changes.

Directory

- `acct1`
 -
 - `Account(8404, 500)`
- `acct2`
 -
 - `Account(8405, 325)`
- `elena`
 -
 - `Customer("Elena", Account(8404, 650))`
- `jorge`
 -
 - `Customer("Jorge", Account(8404, 500))`

We know from running the code, however, that the account is aliased, so that the balances accessible from each of `acct`, `elena.acct`, and `jorge.acct` all reflect the update. This suggests that the actual directory should look something like

Directory

- `acct1`
 -
 - `Account(8404, 650)`
- `acct2`
 -
 - `Account(8405, 325)`
- `elena`
 -
 - `Customer("Elena", Account(8404, 650))`

- `jorge`

→

```
Customer("Jorge", Account(8404, 650))
```

But this is also weird. The directory represents the information that Pyret or Python maintain about your defined names and their values. What in the directory indicates that those three balances should change, but not the balance of `acct2`? Put differently, what reflects the aliasing? Nothing!

The directory as we have used it up until now works fine for programs without mutation. But once we have both mutation and aliasing, this simple idea of mapping names to values breaks down because it doesn't capture the aliases. We need a refined representation of the connections between names and values that does capture aliasing.

11.1.4.1 Introducing the Heap Our original presentation of the directory reflected the aliases that referred to a single `Account` through repeated use of the name `acct1`. We only lost that sharing when we replaced `acct1` with its value while setting up the data for Elena and Jorge. The rule that names can't appear in the values is still important, especially in the presence of mutation (we'll return to this later in Modifying Variables in Memory). But the idea of having a single term that can be reused to reflect sharing is a good one. Indeed, it reflects what happens inside your computer.

Every time you use a constructor to create data, your programming environment stores it in the memory of your computer. Memory consists of a (large) number of slots. Your newly-created datum goes into one of these slots. Each slot is labeled with an address. Just as a street address refers to a specific building, a memory address refers to a specific slot where a datum is stored. Memory slots are physical entities, not conceptual ones. A computer with a 500GB hard drive has about 500 billion slots in which it can store data. Not all of that memory is available to your programming environment: your Web browser, applications, operating system, and so on all get stored in the memory. Your programming environment does get a portion of memory to use for storing its data. That portion is called the heap.

When you write a statement like

```
acct1 = Account(8404, 500)
```

your programming environment puts the new `Account` into a physical slot in the heap, then associates the address of that slot with the variable name in the directory. The name in the directory doesn't map to the value itself, but rather to the address that holds the value. The address bridges between the physical storage location and the conceptual name you want to associate with the new datum. In other words, our directory really looks like:

Directory

- `acct1`

→ 1001

Heap

- 1001: `Account(8404, 500)`

Our revised version has two separate areas: the directory (mapping names to addresses) and the heap (showing the values stored at the addresses). We will use four-digit numbers for addresses, prefixed with an @ symbol (reserving numbers with fewer digits for data values). The specific number for the initial address (here 1001) is arbitrary. Subsequent storage of structured data values will use the addresses in order. Let's write out the directory and heap contents for our initial definitions of accounts in this new format, and see how it supports the aliasing that we intended.

First, we create both `acct1` and `acct2` in order as follows. Note that the `Account` associated with name `acct2` goes in address 1002.

Directory

- `acct1`

→ 1001

- acct2

→ 1002

Heap

- 1001: Account(8404, 500)
- 1002: Account(8404, 500)

When we run

Python

Pyret

```
elena = Customer("Elena", acct1)
elena = customer("Elena", acct1)
```

what happens? As before, we look up what the name `acct1` refers to in the directory and substitute the result for the name in the `Customer` data. Now, `acct1` evaluates to an address, 1001. Therefore, the `Customer` value in the heap contains an address:

Directory

- acct1
- 1001
- elena
- 1002

Heap

- 1001: Account(8404, 500)
- 1002: Customer("Elena", 1001)

Similarly, when we run

Python

Pyret

```
jorge = Customer("Jorge", acct1)
jorge = customer("Jorge", acct1)
```

the directory and heap look like this:

Directory

- acct1
- 1001
- acct2
- 1002
- elena
- 1003
- jorge
- 1004

Heap

- 1001: Account(8404, 500)

- 1002: `Account(8405, 3250)`
- 1003: `Customer("Elena", 1001)`
- 1004: `Customer("Jorge", 1001)`

Do Now!

Fun fact in the Web version of the book: Did you try hovering over the addresses? Try it now!

With the heap articulated separately from the directory, we now see the relationship between the `acct` fields for the two customers and the name `acct1`: they refer to the same address, which in turn means they refer to the same value. In contrast, the name `acct2`, which was not aliased in the original code, refers to an address that is not referenced anywhere else. This is the heart of aliasing: that's why changes made through one name also affect values viewed through another.

Do Now!

Write three distinct expressions each of which uses a different name in the directory to return the balance in account `acct1`.

Do Now!

Would the following statement work to update the balance in Elena and Jorge's shared account?

Python

```
elena.acct.balance = jorge.acct.balance - 50
```

Pyret

```
elena.acct!{balance: jorge.acct!balance - 50}
```

Does this seem like a good or bad way to do this computation? Why?

Do Now!

Extend the most recent directory and heap contents to reflect running the following statement:

```
acct3 = acct1
```

Did you change the heap in the previous exercise? Should you have?

Three rules guide how the directory and heap are affected by running programs:

1. If the code construct a new piece of structured data, put the new piece of structured data at the next address in the heap.
2. If the code associates a name with a piece of structured data, the directory should map the name to the address of the datum in the heap.
3. If the code modifies a field within structured data, modify the data in the heap.

In the example above, we did not alter the heap in any way; only the directory should be modified to reflect that `acct3` and `acct1` are now aliases.

11.1.4.2 Basic Data and the Heap The above rules don't indicate what happens when we have basic data, such as numbers or strings, associated with names in the directory. Do those values also get addresses in the heap?

They do not. As our example with shared accounts illustrated, we need the heap so that updates to fields of shared data affect all aliases (names that refer to) those data. Basic data don't have fields, so there is no need to put them in the heap. Here's a concrete example:

```
x = 4
prof = "Dr. Kumar"
```

The corresponding directory and heap contents would be as follows:

Directory

- `x`
→
4
- `prof`
→ "Dr. Kumar"

Notice that this particular program puts nothing in the heap: according to our rules above, only structured data only go into the heap. Now assume our program also had a dataclass (Python) or datatype (Pyret) for `Offices`, with a professor's name and room number. Here's another example showing a combination of basic and structured data:

```
x = 4
prof = "Dr. Kumar"
office1 = Office("Dr. Lakshmi", 311)
office2 = Office(prof, 310 + x)
```

Directory

- `x`
→
4
- `prof`
→ "Dr. Kumar"
- `office1`
→ 1005
- `office2`
→ 1006

Heap

- 1005: `Office("Dr. Lakshmi", 311)`
- 1006: `Office("Dr. Kumar", 314)`

Though specific language implementations can vary, this shows that it is sufficient to think of basic data as residing in the directory, not the heap. The whole point of structured data is that they have both their own identity and multiple components. The heap gives access to both concepts. Basic data can't be broken down (by definition). As such, there is nothing lost by putting them only in the directory.

But what about strings? We've referred to them as basic data until now, but don't they have "components", namely the characters that make up the string? Yes, that is technically accurate. However, we are treating strings as basic data because we aren't using operations that modify that sequence of characters. This is a subtle point, one that usually comes up later in computer science. This book will leave strings in the directory, but if you are writing programs that modify the internal characters, put them in the heap instead.

[contents](#) ← [prev](#) [up](#) [next](#) →

11.2 Understanding Equality

11.2 Understanding Equality

11.2.1 Equality of Data Now that we have the ability to mutate data, it's worth asking what it means for two pieces of data to be equal. We'll motivate this through a concrete example. Following the naming convention of Data Mutation and the Directory, we will write every name only once, using the upper-case name from Python, but everything we write will equally be true for Pyret.

First, consider these three statements:

```
a1 = Account(8603, 500)
a2 = Account(8603, 500)
a3 = Account(8603, 250)
```

Do Now!

Which of the above `Accounts` do you consider “equal”?

The third `Account` has a different balance than the first two, so it can't be considered equal to either of the first two. The first two have the same contents, so arguably they can be considered equal.

Now, let's consider the directory and heap that would result from running these three statements:

Directory

- a1
→ 1120
- a2
→ 1121
- a3
→ 1122

Heap

- 1120:
`Account(8603, 500)`
- 1121:
`Account(8603, 500)`
- 1122:
`Account(8603, 250)`

From the perspective of the heap, each account ends up at its own address. Those different addresses are a way in which the two values are not the same: they have the same contents, but not the same address. Is that relevant? To explore this, let's associate another name (`a4`) with the same address as `a2`, then change the balance in `a2`. For now we will show just the Python version:

```

a1 = Account(8603, 500)
a2 = Account(8603, 500)
a3 = Account(8603, 250)
a4 = a2
# checkpoint 1
a2.balance = 800
# checkpoint 2

```

What does memory look like before and after checkpoint 1? Before the checkpoint:

Directory

- a1
→ 1130
- a2
→ 1131
- a3
→ 1132
- a4
→ 1131

Heap

- 1130:
Account(8603, 500)
- 1131:
Account(8603, 500)
- 1132:
Account(8603, 250)

a1 and a2 refer to two different Accounts with the same contents. After checkpoint 1, those contents are different because we modified the contents of the balance field in a2:

Directory

- a1
→ 1130
- a2
→ 1131
- a3
→ 1132
- a4
→ 1131

Heap

- 1130:
Account(8603, 500)
- 1131:
Account(8603, 800)

- 1132:

```
Account(8603, 250)
```

In contrast, `a2` and `a4` are aliases for the same `Account`. Therefore, their values change in lockstep: asking to display the value of either one would now show an account with a balance of 800.

Do Now!

What do you think now? Are the first two accounts equal?

11.2.2 Different Equality Operations This sequence of examples points out that we seem to be raising three possible notions of equality:

1. Whether two values have the same contents. This is formally called structural equality; you can think of it as a “print equality”, namely, when displayed, do the two values look the same.
2. Whether two values live at the same address, i.e., there is actually only one value in memory. This is formally called reference equality. Usually, we would refer to the two values by different names (so there is the possibility that they are different), and reference equality checks whether the names are aliases. Observe that a given value always prints the same way, so any two names that have reference equality also have structural equality, but not vice versa.

Which notion of equality is “correct”? It turns out that they are valuable in different contexts. For this reason, programming languages generally provide multiple equality operations, letting the programmer indicate which kind of equality they mean in their context.

Unfortunately, the names of equality operations, and their exact meaning, vary across languages. Therefore, we will examine each of Pyret and Python separately.

11.2.2.1 Equality in Python The `==` operator that you learned in Pyret and we carried into Python checks for structural equality, independent of addresses:

```
a1 == a2
```

True

```
a2 == a4
```

True

However, note that this will no longer be true at checkpoint 2:

```
a1 == a2
```

False

```
a2 == a4
```

True

If we instead want to check for aliasing, we instead use an operation called `is` (not to be confused with Pyret’s `is`, which is used for writing tests):

```
a1 is a2
```

False

```
a2 is a4
```

True

This explains why `a2 == a4` was true both before and after the mutation, but `a1 == a2` was no longer true after it. The latter seems to violate a very basic meaning of “equality”; the problem here is caused by the introduction of mutation.

As we go forward, you'll get more practice with when to use each kind of equality. The `==` operator is more accepting, so it is usually the right default. If you actually need to know whether two expressions evaluate to the same address, you should instead use `is`.

11.2.2.2 Equality in Pyret Equality in Pyret is somewhat more detailed, because the language wants you to think harder about what is happening in your programs.

Recall that we are using the datatype in Example: Bank Accounts and have written the following definitions:

```
a1 = account(8603, 500)
a2 = account(8603, 500)
a3 = account(8603, 250)
a4 = a2
# checkpoint 1
a2!{balance: 800}
# checkpoint 2
```

In Python, we saw that `a1 == a2` before the mutation. However, in Pyret, this produces `false!` Why?

The reason is because structural equality is actually complicated; there are two different questions we could be asking:

1. Are these two values structurally equal right now?
2. Will these two values be structurally equal always?

Pyret makes a distinction between these two.

By default, Pyret tends towards safer programming practices. Therefore, the standard (structural) equality predicate, `==`, will only return `true` if the two values will always be equal. Thus:

```
a2 == a4
```

```
true
```

Because the two values are actually aliases, no matter how one changes, the “other” will always change in the same way. Therefore, they will always “print the same”. We can confirm that they are aliases by using Pyret’s reference equality operator, `<=>`:

```
a1 <=> a2
```

```
false
```

```
a2 <=> a4
```

```
true
```

In contrast, that guarantee does not apply to `a1` and `a2`; and indeed, at checkpoint 2, we see that they are no longer equal. Hence

```
a1 == a2
```

```
false
```

However, there is a time when `a1` and `a2` do print the same, namely before checkpoint 1. Therefore, Pyret provides another equality operator that checks whether values are equal at the moment, `=~`. If we ask this before checkpoint 1, we get:

```
a1 =~ a2
```

```
true
```

But if we ask the same question at checkpoint 2, we get:

```
a1 =~ a2
```

```
false
```

These operators and their funny symbols may be hard to remember, but Pyret also gives them useful (if longer) names, and they can be used as ordinary functions:

Symbol

Function

Type

Meaning

`==`

`equal-always`

Structural

If it returns `true`, they will always be equal, irrespective of any future mutations.

`=~`

`equal-now`

Structural

If it returns `true` they are currently equal, but that may change after future mutations.

`<=>`

`identical`

Reference

Returns `true` if the two arguments are aliases, `false` otherwise.

Thus, before checkpoint 1:

```
equal-now(a1, a2)
true
equal-now(a2, a4)
true
equal-always(a1, a2)
false
equal-always(a2, a4)
```

```
true
  identical(a1, a2)
false
  identical(a2, a4)
true
```

After checkpoint 2, we no longer need to check any of the `equal-always` or `identical` relationships again, because by definition they cannot change. But we should check `equal-now` again. Sure enough:

```
equal-now(a1, a2)
false
  equal-now(a2, a4)
true
```

Therefore, in Pyret, the `==` operator is the same as `equal-always`. When data contain mutable fields, this will always produce `false`, because even if the values are structurally equal now, it's possible that a future mutation will change that. This is to remind you to be careful in the presence of mutation. In situations where we really care only about equality at that instant, we can use `=~`, i.e., `equal-now`.

The examples above might suggest that only aliased values are `equal-always`. This is not true! If our data are immutable (which is the default in the language), then if two values are structurally equal now, they must remain structurally equal forever. For such data, `equal-always` will return `true` even when they are not aliases. This is a reminder that we get stronger guarantees about immutable data.

It is worth noting that upto this point we have used `equal-always`—in the form of both `==` and Pyret's `is` in testing—without really bothering to understand very much about how it works, and yet have always gotten predictable answers. This suggests that there is something natural about working with immutable data. In contrast, with mutable data, something has to give. Pyret made a conscious design choice to reflect this in the distinction between `equal-always` and `equal-now`. Python made a different choice, which results in “equality” having a perhaps surprising meaning. (Python has no notion of `equal-always`, only `equal-now` or `=~`, which is written as `==`, and `identical` or `<=>`, which is written as `is`.)

contents ← prev up next →

11.3 Arrays and Lists in Memory

11.3 Arrays and Lists in Memory In Understanding Equality, we drew memory diagrams to show how values appear in the heap. At that time, we looked only at structured data. Now we will look at aggregate data: lists in Python and arrays in Pyret.

Both Python lists and Pyret arrays are stored in memory with a starting value that indicates how many values there are, followed by the actual elements in subsequent addresses. For instance, suppose we write the following value:

Python

```
s1 = ['bread', 'coffee', 'eggs']
```

Pyret

```
s1 = [array: 'bread', 'coffee', 'eggs']
```

Here's what memory might look like:

Directory

- s1
→ 1001

Heap

- 1001: (length:3)
- 1002: "bread"
- 1003: "coffee"
- 1004: "eggs"

If these terms mean anything to you: Python's default list implementation is array-based, not a linked list. So at 1001 there's an indication of how many entries follow, and the next memory locations have those values. There are no directory entries for the individual elements, but they can be reached by referring to s1 followed by an offset: for instance,

Python

Pyret

```
s1[2]
```

```
s1.get-now(2)
```

Internally, this turns into: “obtain the address of s1, then add 1 and the offset 2 to it”. This produces the address $1001 + 1 + 2 = 1004$. Looking up the value stored at address 1004, in both languages, produces 'eggs'. You may find it odd that the offsets begin at 0. While it is indeed confusing—the “first” value is at offset 0, the “third” value at offset 2, and so on—this is a convention both Python and Pyret chose to be consistent with most other programming languages.

Similarly, suppose we try to change the shopping list's content to replace the second value with 'tea'. Recall that the second value is at offset 1:

Python

Pyret

```
s1[1] = 'tea'  
s1.set-now(1, 'tea')
```

Again, we obtain the address of `s1`, which is 1001; add 1 and the offset (1) to it; this gives us the address 1003. Now, we modify the value at 1003 to be the new value:

Directory

- `s1`
→ 1001

Heap

- 1001: (length:3)
- 1002: "bread"
- 1003: "tea"
- 1004: "eggs"

If we now ask the language for the list as a whole, we see the change: in Python,

```
s1  
['bread', 'tea', 'eggs']
```

and in Pyret,

```
s1  
[array: "bread", "tea", "eggs"]
```

Observe that what we have learned about aliasing applies here, too. Suppose Shaunae and Jonella share a shopping list, where `s1` is Shaunae's and Jonella writes:

Python

Pyret

```
j1 = s1  
j1 = s1
```

Now `j1` and `s1` are aliases for the same data in the heap:

Directory

- `s1`
→ 1001
- `j1`
→ 1001

Heap

- 1001: (length:3)
- 1002: "bread"
- 1003: "tea"
- 1004: "eggs"

Thus, modifying the list `j1` has the same impact as modifying it via `s1`:

Python

Pyret

```
jl[0] = 'butter'  
jl.set-now(0, 'butter')  
means the memory looks like
```

Directory

- **sl**
→ 1001
- **jl**
→ 1001

Heap

- 1001: (length:3)
- 1002: "butter"
- 1003: "tea"
- 1004: "eggs"

Thus, if we ask for the value of **jl**, we will see in Python:

```
sl  
['butter', 'tea', 'eggs']  
and in Pyret:  
  
sl  
[array: "butter", "tea", "eggs"]  
even though we did not make a modification through the name sl.  
contents ← prev up next →
```

11.4 Cyclic Data

11.4 Cyclic Data

11.4.1 Creating Cyclic Data Earlier [Aliasing], we introduced the idea of aliased bank accounts, where multiple customers can operate the same account. Sometimes, a bank wants to keep track of all the customers who have access to a given account. For instance, when the account balance runs low, it would want to notify all the customers who have access to it.

Therefore, each account needs to maintain a list of its customers. Because the set of owners can change over time, we make that field mutable in Pyret:

Python

Pyret

```
@dataclass
class Account:
    id: int
    balance: int
    owners: list # of Customer

@dataclass
class Customer:
    name: str
    acct: Account

data Account:
    account(id :: Number,
        ref balance :: Number,
        ref owners :: List<Customer>)
end

data Customer:
    cust(name :: String,
        acct :: Account)
end
```

If you look closely, you'll see that `Account` refers to `Customer` (specifically, a list of them) and in turn `Customer` refers to `Account`. This could get interesting.

Previously, we could create an account with one customer as follows:

Python

```
elena = Customer("Elena", Account(8404, 500))
```

Pyret

```
elena = cust("Elena", account(8404, 500))
```

How do we do that now? Every `Account` requires a list of its `Customers`. We need to write

Python

```
elena = Customer("Elena", Account(8404, 500, [_____]))
```

Pyret

```
elena = cust("Elena", account(8404, 500, [list: _____]))
```

But what goes in _____? It needs to refer to the very customer account that we are presently creating.

Another way to think about writing this is:

Python

```
acct1 = Account(8404, 500, [_____])
elena = Customer("Elena", acct1)
```

Pyret

```
acct1 = account(8404, 500, [list: _____])
elena = cust("Elena", acct1)
```

This hasn't solved our fundamental problem—we still need to fill in _____—but at least we now have names to refer to entities. We would like to be able to write

Python

```
acct1 = Account(8404, 500, [elena])
elena = Customer("Elena", acct1)
```

Pyret

```
acct1 = account(8404, 500, [list: elena])
elena = cust("Elena", acct1)
```

But when we try to run this, both Python and Pyret will give us an error. That is because they try to evaluate the right-hand-side of the first line to create an account, whose heap address will be bound in the directory to `acct1`. To do so, they must evaluate that account-creation expression. In doing so, they look up the name `elena`. However, `elena` has not yet been bound in the directory. Therefore, they produce an error.

Observe that we can't just reverse the order of these two bindings. If we try that, we end up with the same problem: we try to create a customer that refers to `acct1`. But we haven't yet defined `acct1`, producing the same error.

The problem is we are trying to create cyclic data. The two data refer to one another. We could already sense that this might happen from the data definitions, and now we must confront it. The problem is that we have to create some value first, and we can't produce either one correctly since each one depends on the other.

As you might guess, we have to compromise. We have to construct one of the data first, and when we do so, it might not be entirely accurate. Then we create the other, and then modify the first one to have the correct contents.

In our case, the Pyret version makes clear what order to use in creating the data. Nothing in a `Customer` is mutable (nor needs to be), whereas the list of account owners is (and should be, because the set of customers can grow). Therefore, we can write:

Python

```
acct1 = Account(8404, 500, [])
```

Pyret

```
acct1 = account(8404, 500, empty)
```

Note that at this point, this is actually accurate! There are no owners of this account.

Now we create Elena's account:

Python

```
elena = Customer("Elena", acct1)
```

Pyret

```
elena = cust("Elena", acct1)
```

At this point, our memory looks like this: For simplicity, we will show the list of owners inside the account instead of putting it in its own memory location(s).

Directory

- **acct1**
→ 1001
- **elena**
→ 1002

Heap

- 1001: Account(8404, 500, [])
- 1002: Customer("Elena", 1001)

Now things are slightly inaccurate: the account at 1001 does have an owner, which is not yet reflected. So we have to update it to reflect that:

Python

Pyret

```
acct1.owners = [elena]  
acct1!{owners: [list: elena]}
```

We can legitimately do this now because **elena** is bound in the dictionary. Furthermore, it is bound to something useful: Elena's customer information. So now the values are properly set up: Elena's customer information refers to the account, and the account refers to Elena's customer information:

Directory

- **acct1**
→ 1001
- **elena**
→ 1002

Heap

- 1001: Account(8404, 500, [1002])
- 1002: Customer("Elena", 1001)

This is the cycle in “cyclic”: 1001 depends on 1002 and 1002 depends on 1001.

Observe that if we introduce another customer, Jorge, who shares the same account, we can update the account to reflect that also:

Python

```
jorge = Customer("Jorge", acct1)
```

Pyret

```
jorge = cust("Jorge", acct1)
```

Again, the information in **acct1** is inaccurate because it does not reflect the new owner. We can modify it in a similar way:

Python

```
acct1.owners = acct1.owners + [jorge]
```

Pyret

```
acct1!{owners: acct1!owners + [list: jorge]}
```

So now our memory would look like this:

Directory

- acct1
 - 1001
- elena
 - 1002
- jorge
 - 1003

Heap

- 1001: Account(8404, 500, [1002, 1003])
- 1002: Customer("Elena", 1001)
- 1003: Customer("Jorge", 1001)

Do Now!

We wrote slightly different code when adding Jorge's account than when adding Elena's account.
Is one better than the other?

The code for Elena's addition ignored whatever owners there previously were. That is, it would only work correctly in a setting where there were no other owners. The code for Jorge's addition takes into account all the previous owners. Therefore, Elena's code was perfectly fine for illustrating the simple first case, but Jorge's code is more general in that it will work in all settings (including when the prior list of owners is empty).

Exercise

Write a function that takes care of adding a customer to an account.

11.4.2 Testing Cyclic Data When you want to write a test involving circular data, you can't write out the circular data manually. For example, imagine that we wanted to write out `acct1` from earlier:

Python

```
assert(acct1 = Account(8404, 500, [Customer("Elena", Account(8404, 500, ...))]))
```

Pyret

```
check:  
  acct1 is  
    account(8404, 500, [list: cust("Elena", account(8404, 500, ...))])  
end
```

However, because of the circularity, we can't finish writing down the data. We can't just leave part of it unspecified with `...`.

This leaves us with two choices:

You have two options: write tests in terms of the names of data, or write tests on the components of the data.

Here's an example that illustrates both. After setting up the account, we might want to check that the owner of the new account is the new customer:

```
assert(new_acct.owner is new_cust)
```

Here, rather than write out the `Customer` explicitly, we use the name of the existing item in the directory. This doesn't require you to write ellipses. We also focused on just the `owner` component, as a part of the `Account` value that we expected to change.

11.4.3 Cycles in Practice Cyclic data show up in many settings in real programs. Whenever two data are interrelated, and we have good reason to want to get from either one to the other, they have the potential to have references to each other, which can lead to cycles. Sometimes the connection can be to provide updates, as above; other times it can simply be for navigational convenience.

Consider the Document Object Model (DOM), which is the data structure that represents every Web page in a Web browser. Programmers usually think of the DOM hierarchically, as a tree, because every node refers to all the nodes that constitute it: e.g., a page has references to each of its paragraphs, a list has references to each list item, and so forth. However, every one of these elements also has a reference to its parent. This way, a program can conveniently traverse “downward” or “upward”.

Programming with cyclic data introduces complications. If we traverse the data naïvely, we would go into an infinite loop. Rather, we have to keep track of the data we have previously visited, and make sure we don’t visit them again (see [The Size of a DAG](#)). Indeed, cyclic data are graphs [Graphs], so issues in processing graphs become relevant here.

One interesting question that programming languages face is, how do you print cyclic data? For instance, what happens if the programmer writes

```
acct1
```

? To print the account we must print its owners; to print each owner, we must print their account; to print that account...

Do Now!

Try it out in both Python and Pyret!

Different programming languages handle this problem in different ways. Some languages will go into an infinite loop trying to print cyclic data. Both Python and Pyret handle this more intelligently. Determining even whether a datum is cyclic is an interesting question, which we take up in [Detecting Cycles](#).

contents ← prev up next →

12.1 Modifying Variables

12.1 Modifying Variables

12.1.1 Modifying Variables in Memory Now that we have introduced the idea of the heap, let's revisit our use of a variable to compute the sum of elements in a list. Here again is our code from earlier:

```
run_total = 0
for num in [5, 1, 7, 3]:
    run_total = run_total + num
```

Let's see how the directory and heap update as we run this code. In Basic Data and the Heap, we pointed out that basic data (such as numbers, strings, and booleans) don't get put in the heap because they have no internal structure. Those values are stored in the directory itself. Therefore, the initial value for `run_total` is stored within the directory.

Directory

- `run_total`
 -
 - 0

The `for` loop also sets up a directory entry, this time for the variable `num` that is used to refer to the list elements. When the loop starts, `num` takes on the first value in the list. Thus, the directory appears as:

Directory

- `run_total`
 -
 - 0
- `num`
 -
 - 5

Inside the `for` loop, we compute a new value for `run_total`. The use of `=` tells Python to modify the value of `run_total`.

Do Now!

Does this modification get made in the directory or the heap?

Since basic data values are stored only in the directory, this update modifies the contents of the directory. The heap isn't involved:

Directory

- `run_total`
 -
 - 5

- num

→

5

This process continues: Python advances num to the next list element

Directory

- run_total

→

5

- num

→

1

then modifies the value of run_total

Directory

- run_total

→

6

- num

→

1

This process continues until all of the list elements have been processed. When the for-loop ends, the directory contents are:

Directory

- run_total

→

16

- num

→

3

There are two takeaways from this example:

- When we use = to update the value associated with a variable, the variable's entry in the directory changes to reflect the new value.
- For loops introduce a name into the directory (the one the programmer chose to refer to the individual list elements). As the loop progresses, Python updates the value associated with that name to refer to each successive element.

Exercise

Draw the sequence of directory contents for the following program:

```
score = 0
score = score + 4
score = 10
```

Exercise

Draw the sequence of directory contents for the following program:

```
count_long = 0
for word in ["here", "are", "some", "words"]:
    if len(word) > 4:
        count_long = count_long + 1
```

12.1.2 Variable Updates and Aliasing In State, Change, and Testing, we saw how a statement of the form `elena.acct.balance = 500` resulted in a change to `jorge.acct.balance`. Does this same effect occur if we update the value of a variable directly, rather than a field? Consider the following example:

```
y = 5
x = y
```

Do Now!

What do the directory and heap look like after running this code?

Since `x` and `y` are assigned basic values, there are no values in the heap:

Directory

- `y`
→
5
- `x`
→
5

Do Now!

If we now evaluate `y = 3`, does the value of `x` change?

It does not. The value associated with `y` in the directory changes, but there is no connection between `x` and `y` in the directory. The statement `x = y` says "get the value of `y` and associate it with `x` in the directory". Immediately after this statement, `y` and `x` refer to the same value, but this relationship is neither tracked nor maintained. If we associate either variable with a new value, as we do with `y = 3`, the directory entry for that variable—and only the directory entry for that variable—are changed to reflect the new value. Thus, the directory after we evaluate `y = 3` appears as follows:

Directory

- `y`
→
3
- `x`
→
5

This example highlights that aliasing occurs only when two variables refer to the same piece of data with components, not when variables refer to basic data. This is because data with components are stored in the heap, with heap address stored in the directory. Note, though, that uses of `varname = ...` still affect the directory, even when the values are data with components.

Do Now!

After running the following code, what is the value of `ac2.balance`?

```
ac1 = Account(8623, 600)
ac2 = ac1
ac1 = Account(8721, 350)
```

Draw the directory and heap contents for this program and check your prediction.

All three of these lines results in changes in the directory; the first two result in changes in the heap, but only because we made new pieces of data. `ac1` and `ac2` are aliases immediately after running the second line, but the third line breaks that relationship.

Do Now!

After running the following code, what is the value of `ac1.balance`?

```
savings = 475
ac3 = Account(8722, savings)
savings = 500
```

Draw the directory and heap contents for this program and check your prediction.

Since the value of `savings` is stored in `ac3.balance`, and not the name `savings` itself, updating the value of `savings` on the third line does not affect `ac3.balance`.

12.1.3 Updating Variables versus Updating Data Fields We've now seen two different forms of updates in programs: updates to fields of structured data in State, Change, and Testing, and updates to the values associated with names when computing over lists with `for` loops. At a quick glance, these two forms of update look similar:

```
acct1.balance = acct1.balance - 50
run_total = run_total + fst
```

Both use the `=` operator and compute a new value on the right side. The left sides, however, are subtly different: one is a field within structured data, while the other is a name in the directory. This difference turns out to be significant: the first form changes a value stored in the heap but leaves the directory unchanged, while the second updates the directory but leaves the heap unchanged.

At this point, you might not appreciate why this difference is significant. But for now, let's summarize how each of these forms impacts each of the directory and the heap.

Strategy: Rules for updating the directory and the heap

Summarizing, the rules for how the directory and memory update are as follows:

- We add to the heap when a data constructor is used
- We update the heap when a field of existing data is reassigned
- We add to the directory when a name is used for the first time (this includes parameters and internal variables when a function is called)
- We update the directory when a name that is already in the directory is subsequently assigned a new value)

Do Now!

After running the following code, what is the value of `ac3.balance`?

```
ac2 = Account(8728, 200)
ac3 = ac2
print(ac3.balance)
ac2.balance = 500
print(ac3.balance)
ac2 = Account(8734, 350)
ac2.balance = 700
print(ac3.balance)
```

Draw the directory and heap contents for this program and check your prediction.

This example combines updates to variables and updates to fields. On the third line, `ac2` and `ac3` refer to the same address in the heap (which contains the `Account` with id 8728. Immediately after updating `ac2.balance` on the fourth line, the balance in both `ac2` and `ac3` is 500. Line six, however, creates a new `Account` in the heap and updates the directory to have `ac2` refer to that new `Account`. From that point on, `ac2` and `ac3` refer to different accounts, so the update to the balance in `ac2` on the seventh line does not affect `ac3`.

This example illustrates the subtleties and impacts of different uses of `=`. Programs behave differently depending on whether the left side of the `=` is a variable name or a field reference, and on whether the right side is basic data or data with components. We will continue to work with these various combinations to build your understanding of when and how to use each one.

12.1.4 Updating Parameters in Function Calls When we first learned about the directory in [REFSEC], we showed how function calls created their own local directory segments to store any names that got introduced while running the function. Now that we have the ability to update the values associated with variables, we should revisit this topic to understand what happens when these updates occur within functions.

Consider the following two functions:

```
def add10(num: int):
    num = num + 10

def deposit10(ac: Account)
    ac.balance = ac.balance + 10
```

Let's use these two functions in a program:

```
x = 15
a = Account(8435, 500)
add10(x)
deposit10(a)
```

Do Now!

What are the values of `x` and `a` when the program has finished?

Let's draw out the directory and heap for this program.

We need a way to distinguish local directories from the global one – easiest for now might be to add a form for local-env-with-heap that uses the label "Local Directory (fun name)".

After the first two lines but before the function calls, we have the following:

Directory

- `x`
→ 15
- `a`
→ 1014

Heap

- 1014:
`Account(8435, 500)`

Calling `add10` creates a local directory containing the name of the parameter:

Directory

- `num`
→ 15

Heap

- 1014:

`Account(8435, 500)`

Wait – why is the heap listed alongside the local directory? Only the directory gets localized during function calls. The same heap is used at all times.

The body of `add10` now updates the value of `num` in the directory to 25. This does not affect the value of `x` in the top-level directory, for the same reasons we explained in [REFSEC] regarding the lack of aliasing between variables that refer to basic data. Thus, once the function finishes and the local directory is deleted, the value associated with `x` is unchanged.

Now, let's evaluate the call `deposit10(a)`. As with `add10`, we create a local directory and create an entry for the parameter. What gets associated with that parameter in the directory, however?

Directory

- `ac`
→ 1014

Heap

- 1014:
`Account(8435, 500)`

Do Now!

Why didn't we create a new `Account` datum when we made the function call?

Remember our rule for when we create new data in the heap: we only create heap data when we explicitly use a constructor. The function call does not involve creating a new `Account`. Whatever is associated with the name `a` gets associated with the parameter name `ac`. In other words, we have created an alias between `a` and `ac`.

In the body of `deposit10`, we update the balance of `ac`, which is also the balance of `a` due to the aliasing. Since there is no local heap, when the function call is over, the new balance persists in `a`.

All we've done here is put together pieces that we've already seen, just in a new context. We're passing parameters and updating either the (local) directory or the heap according to how we have used `=`. But this example highlights a detail that initially confuses many people when they start writing functions that update variables.

Strategy: Updating Values within Functions

If you want a function to update a value and have that update persist after the function completes, you must put that value inside a piece of data. You cannot have it be basic data associated with a variable name.

12.1.5 Updating Top-Level Variables within Function Calls Let's return to our banking example to illustrate a situation where the ability to update variables is extremely useful. Consider our current process for creating new accounts in the bank by looking at the following example:

```
ac5 = Account(8702, 435)
ac6 = Account(8703, 280)
ac7 = Account(8704, 375)
```

Notice that each time we create an `Account` we have to take care to increase the id number? What if we made a typo or accidentally forgot to do this?

```
ac5 = Account(8702, 435)
ac6 = Account(8703, 280)
ac7 = Account(8703, 375)
```

Now we'd have multiple accounts with the same ID number, when we really need these numbers to be unique across all accounts. To avoid such problems, we should instead have a function for creating accounts that takes the initial balance as input and uses a guaranteed-unique ID number.

How might we write such a function? The challenge is to be able to generate unique ID numbers each time. What if we used a variable to store the next available ID number, updating it each time we created a new account? That function might look at follows:

```
nextID = 8000 # stores the next available ID number

def create_acct(init_bal: float) -> Account:
    new_acct = Account(nextID, init_bal)
    nextID = nextID + 1
    return(new_acct)
```

Let's run this program, creating new accounts as follows:

```
ac5 = create_acct(435)
ac6 = create_acct(280)
ac7 = create_acct(375)
```

Do Now!

Copy this code into Python and run it. Check that each of `ac5`, `ac6`, and `ac7` have unique ID numbers.

What happened? All three of these have the same ID of 8000. It looks like our update to `nextID` just didn't work. Actually, it did work, but to understand how, we have to look at what happened in the directory.

Do Now!

Draw the memory diagram for this example.

After we set up `nextID` and define the function, our memory diagram appears as:

Directory

- `nextID`
→ 8000

Now, let's evaluate `ac5 = create_acct(435)`. We call `create_acct`, which yields the following local directory after creating the `Account` but before updating `nextID`.

Directory

- `init_bal`
→ 435
- `new_acct`
→ 1015

Heap

- 1015:
`Account(8000, 435)`

Do Now!

What do you think happens when we run `nextID = nextID + 1`?

Let's run this carefully. Python first evaluates the right side of the `= (nextID + 1)`. `nextID` is not in the local directory, so Python retrieves its value (8000) from the top-level directory. Thus, this computation becomes `nextID = 8001`.

The question here is how Python treats `nextID = 8001`: we currently have both the local directory for the function call and the top-level directory. Which one should get the new value of `nextID`? Since the local directory is active, Python sets the value of `nextID` there.

Directory

- `init_bal`
→ 435
- `new_acct`
→ 1015
- `nextID`
→ 8001

Heap

- 1015:
`Account(8000, 435)`

Let's repeat that: Python computed `nextID + 1` using the `nextID` value from the top-level directory since there was no value for `nextID` in the local directory. But the setting of the value of `nextID` could and did occur in the local directory. Thus, when `create_acct` finishes, the value of `nextID` in the top-level directory is unchanged. As a result, all of the accounts get the same value.

The computation we are trying to do—updating the top-level variable—is just fine. The problem is that Python (reasonably) defaults to the local directory. To make this work, we need to tell Python that we want to make updates to `next_id` in the top-level directory. Here's the version of `create_acct` that does that:

```
def create_acct(init_bal: float) -> Account:
    global nextID
    new_acct = Account(nextID, init_bal)
    nextID = nextID + 1
    return(new_acct)
```

The `global` keyword tells Python to make updates to the given variable in the top-level directory, not the local directory. Once we make this modification, each account we create will get a unique ID number.

Responsible Computing: Keeping IDs Unpredictable

While this general pattern of generating unique IDs works, in practice we shouldn't use consecutive numbers. Consecutive numbers are guessable: if there is an account 8000 there must be an account 8001, and so on. Guessable account numbers could make it easier for someone who keeps trying to guess valid IDs to use to log into websites or otherwise access information.

Instead, we would use a computation that is less predictable than “add 1” when storing the `nextID` value. For now, the pattern we have shown you is fine. If you were building a real system, however, you'd want to make that computation a bit more sophisticated.

12.1.6 The Many Roles of Variables At this point, we have used the single coding construct of a variable in the directory for multiple purposes. It's worth stepping back and calling those out explicitly. In general, variables serve one of the following purposes:

1. Tracking progress of a computation (e.g., the running value of a result in a `for`-loop)
2. Maintaining information across multiple calls to a single function (e.g., the `next-id` variable)
3. Naming a local or intermediate value in a computation

Each of these uses involves a different programming pattern. The first creates a variable locally within a function. The second two create top-level variables and require using `global` in functions that modify the contents. The third is different from the second, however, in that the third is only meant to be used by a single function. Ideally, there would be a way to not expose the variable to all functions in the third case. Indeed, many programming languages (including Pyret) make it easy to do that. This is harder to achieve with introductory-level concepts in Python, however. The fourth is more about local names rather than variables, in that our code never updates the value after the variable is created.

We call out these three roles precisely because they invoke different code patterns, despite using the same fine-grained concept (assigning a new value to a variable). When you look at a new programming problem, you can ask yourself whether the problem involves one of these purposes, and use that to guide your choice of pattern to use.

contents ← prev up next →

12.2 Mutable Lists

12.2 Mutable Lists Let's expand our study of updates yet again, this time looking at updating lists. We'll start with lists.

Imagine that Shaunae wants to use a program to maintain her shopping list. She creates an initial list with two items:

```
shaunae_list = ["bread", "coffee"]
```

Do Now!

Shaunae wants to add eggs to her list. Write a line of code to accomplish this.

There are two ways you could have done this:

```
# approach 1  
shaunae_list = shaunae_list + ["eggs"]
```

```
#approach 2  
shaunae_list.append("eggs")
```

What is the difference between these two approaches? The difference lies in the impact on the heap.

- The first version creates a new list containing "eggs", then puts the elements of the two lists together in a new list.
- The second version inserts "eggs" into the existing list in the heap.

Let's look at the directories for each version. Here's the final directory for the first version:

Directory

- `shaunae_list`
→ 1010

Heap

- 1005: `List(len:2)`
- 1006: "bread"
- 1007: "coffee"
- 1008: `List(len:1)`
- 1009: "eggs"
- 1010: `List(len:3)`
- 1011: "bread"
- 1012: "coffee"
- 1013: "eggs"

The original version of `shaunae_list` is in address 1005, the list with "eggs" is in 1008, and the combined list is in 1010.

In contrast, the final directory for the second version would look like:

Directory

- shaunae_list
→ 1010

Heap

- 1005: List(len:3)
- 1006: "bread"
- 1007: "coffee"
- 1008: "eggs"

Notice here that the length and contents of the original list are changed to include the newly-appended "eggs".

Do Now!

Which approach do you think is better? Why?

At first glance, the second approach might seem better because it doesn't create additional unnecessary lists. Both approaches result in the same contents in `shaunae_List`, so there seems little benefit to using the additional space.

Unless, of course, we want to still have access to the old version of `shaunae_list` later on. The old list is still in the heap (though our current program has no name through which to access that old list). What if we instead had written the program this way?

```
shaunae_list = ["bread", "coffee"]
prev_list = shaunae_list
shaunae_list = ["paint", "brushes"] + shaunae_list
```

Now, if Shaunae realizes she goofed and put her art supply shopping on the grocery list on that last update, she could "undo" the update by resetting her list variable to the previous list:

```
shaunae_list = prev_list
```

Undoing a modification (just like the undo feature in document-editing tools) is just one example of where it can help to hang on to older versions of data for a little while. The point here is not to give a sophisticated treatment of undoing computations, but more to motivate that there are situations in which creating a new list is preferable to updating the old one.

When might we want to update, rather than preserve, the existing list?

Remember our discussion of aliasing? We wanted two people, Elena and Jorge to share access to a common bank account. Might we ever want a shared shopping list? Sure, Shaunae and her roommate Jonella do share a shopping list, so that they can both add items while letting either one go to the store.

Do Now!

Set up a shared shopping list that is accessible through two names, `shaunae_list` and `jonella_list`. Then, add an item to the list via one of these names and check that the item appears under the other name.

You might have written something like the following:

```
shaunae_list = ["bread", "coffee"]
jonella_list = shaunae_list
jonella_list.append("eggs")
```

If you load this code at the prompt and look at both lists at the end, you'll see they have the same values.

In contrast, had we written the code as follows, only one of them would see the new item:

```
>>> jonella_list = ["apples"] + jonella_list
>>> jonella_list
["apples", "bread", "coffee", "eggs"]
>>> shaunae_list
```

```
["bread", "coffee", "eggs"]
```

Do Now!

Draw the memory diagram for the above program.

Exercise: Creating Lists of Accounts In [REFSEC], we wrote a function to create new accounts for the bank. That function returned each new account as it was created. That meant that every newly-created account had to be associated with a name in the directory (otherwise we would not be able to access it from the heap).

Maintaining either a list or a dictionary of all the created accounts makes much more sense. We'd need only a single name for the collection of accounts, but could still access individual accounts as needed. For example, we might want an `all_accts` list that looks something like the following:

```
all_accts = [Account(8623, 100),
             Account(8624, 300),
             Account(8625, 225),
             ...
           ]
```

Do Now!

Write a program that creates an empty `all_accts` list, then adds a new `Account` to it each time `create_acct` is called. You will need to modify `create_acct` in order to do this. Here is the existing code as a starting point.

```
next_id = 1

def create_acct(init_bal: float) -> Account:
    global next_id
    new_acct = Account(next_id, init_bal, [])
    next_id = next_id + 1
    return new_acct
```

Do Now!

Did you include a line like `global all_accts` in your code? Why or why not?

If you used `append` to update the `all_accts` list, then you would not need to include `global all_accts`. Recall that `global` is needed to tell Python to update a variable in the top-level directory rather than the local directory. If you use `all_accts.append`, however, you are modifying the heap instead of the directory. There is no need for `global` if your code is only modifying heap contents.

contents ← prev up next →

13 Predicting Growth

13 Predicting Growth

We will now commence the study of determining how long a computation takes. We'll begin with a little (true) story.

13.1 A Little (True) Story My student Debbie recently wrote tools to analyze data for a startup. The company collects information about product scans made on mobile phones, and Debbie's analytic tools classified these by product, by region, by time, and so on. As a good programmer, Debbie first wrote synthetic test cases, then developed her programs and tested them. She then obtained some actual test data from the company, broke them down into small chunks, computed the expected answers by hand, and tested her programs again against these real (but small) data sets. At the end of this she was ready to declare the programs ready.

At this point, however, she had only tested them for functional correctness. There was still a question of how quickly her analytical tools would produce answers. This presented two problems:

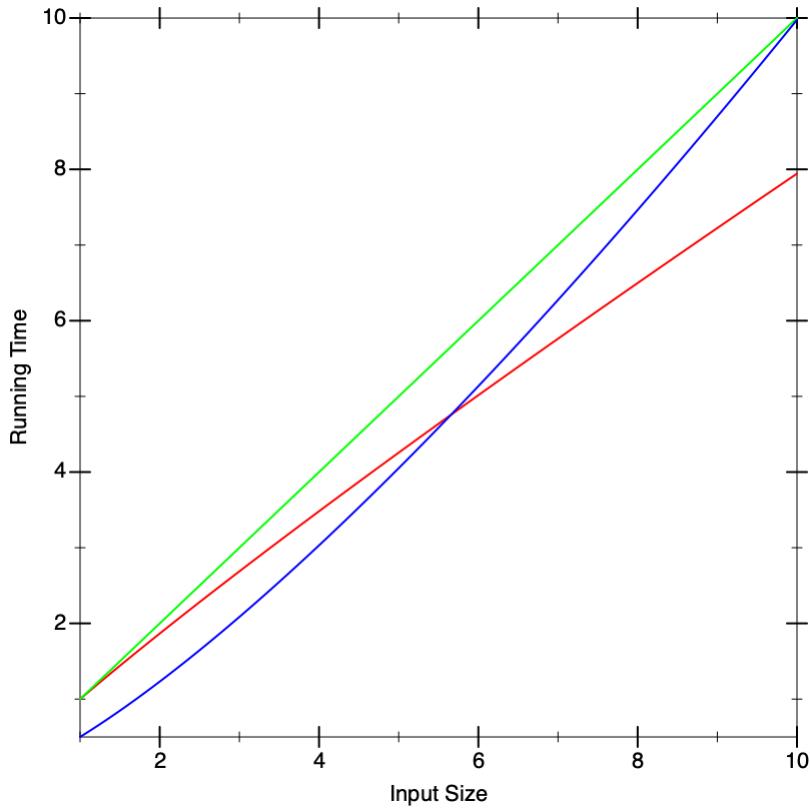
- The company was rightly reluctant to share the entire dataset with outsiders, and in turn we didn't want to be responsible for carefully guarding all their data.
- Even if we did get a sample of their data, as more users used their product, the amount of data they had was sure to grow.

We therefore got only a sampling of their full data, and from this had to make some prediction on how long it would take to run the analytics on subsets (e.g., those corresponding to just one region) or all of their data set, both today and as it grew over time.

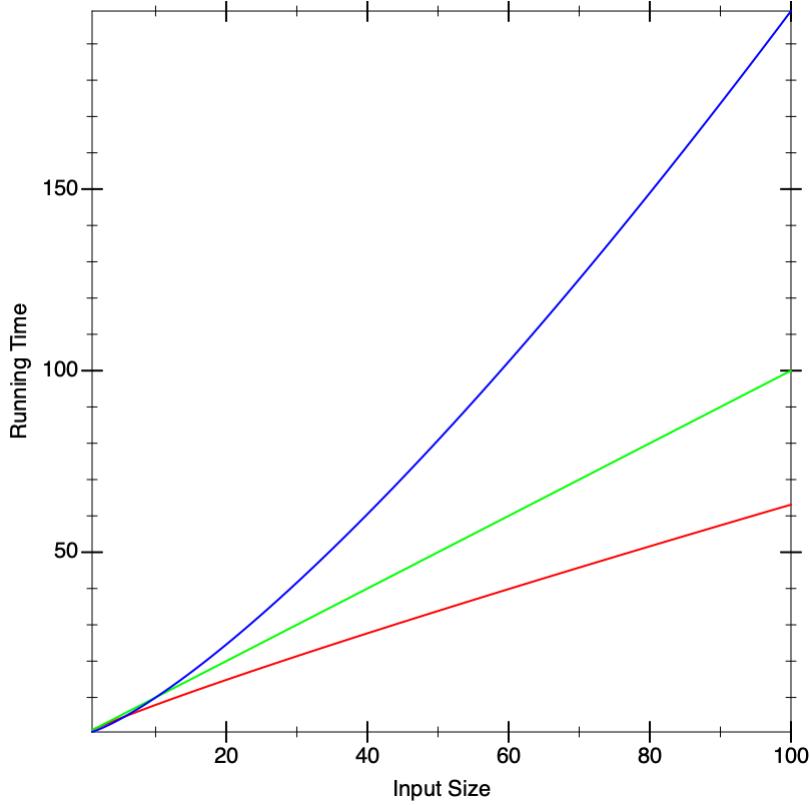
Debbie was given 100,000 data points. She broke them down into input sets of 10, 100, 1,000, 10,000, and 100,000 data points, ran her tools on each input size, and plotted the result.

From this graph we have a good bet at guessing how long the tool would take on a dataset of 50,000. It's much harder, however, to be sure how long it would take on datasets of size 1.5 million or 3 million or 10 million. These processes are respectively called interpolation and extrapolation. We've already explained why we couldn't get more data from the company. So what could we do?

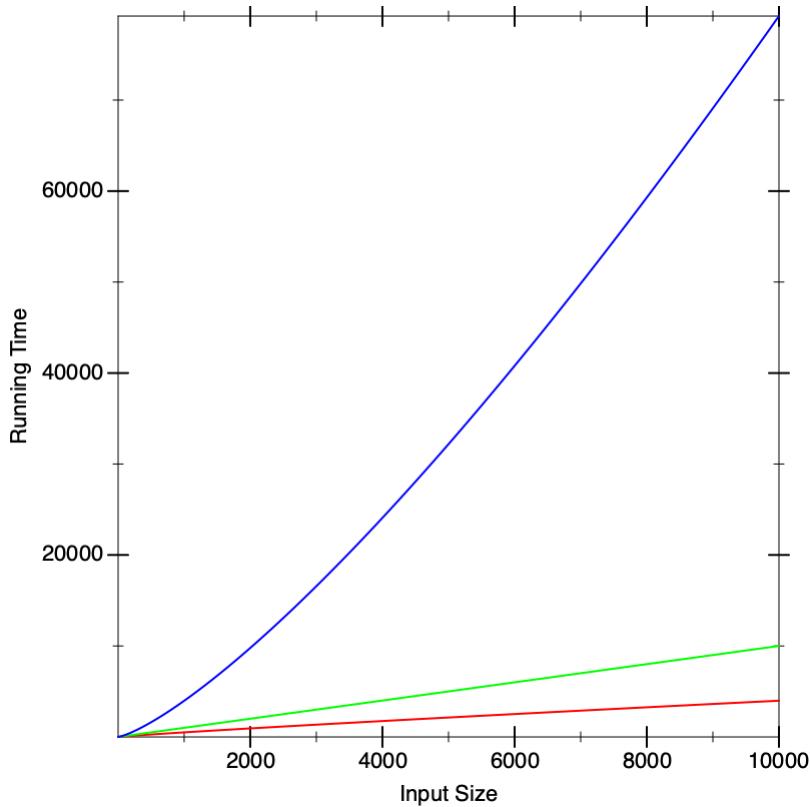
As another problem, suppose we have multiple implementations available. When we plot their running time, say the graphs look like this, with red, green, and blue each representing different implementations. On small inputs, suppose the running times look like this:



This doesn't seem to help us distinguish between the implementations. Now suppose we run the algorithms on larger inputs, and we get the following graphs:



Now we seem to have a clear winner (red), though it's not clear there is much to give between the other two (blue and green). But if we calculate on even larger inputs, we start to see dramatic differences:



In fact, the functions that resulted in these lines were the same in all three figures. What these pictures tell us is that it is dangerous to extrapolate too much from the performance on small inputs. If we could obtain closed-form descriptions of the performance of computations, it would be nice if we could compare them better. That is what we will do in the next section.

Responsible Computing: Choose Analysis Artifacts Wisely

As more and more decisions are guided by statistical analyses of data (performed by humans), it's critical to recognize that data can be a poor proxy for the actual phenomenon that we seek to understand. Here, Debbie had data about program behavior, which led to mis-interpretations regarding which program is best. But Debbie also had the programs themselves, from which the data were generated. Analyzing the programs, rather than the data, is a more direct approach to assessing the performance of a program.

While the rest of this chapter is about analyzing programs as written in code, this point carries over to non-programs as well. You might want to understand the effectiveness of a process for triaging patients at a hospital, for example. In that case, you have both the policy documents (rules which may or may not have been turned into a software program to support managing patients) and data on the effectiveness of using that process. Responsible computing tells us to analyze both the process and its behavioral data, against knowledge about best practices in patient care, to evaluate the effectiveness of systems.

13.2 The Analytical Idea With many physical processes, the best we can do is obtain as many data points as possible, extrapolate, and apply statistics to reason about the most likely outcome. Sometimes we can do that in computer science, too, but fortunately we computer scientists have an enormous advantage over most other sciences: instead of measuring a black-box process, we have full access to its internals, namely the source code. This enables us to apply analytical methods.“Analytical” means applying algebraic and other mathematical methods to make predictive statements about a process without running it. The answer we compute this way is complementary to what we obtain from the above experimental analysis, and in practice we will usually want to use a combination of the two to arrive a strong understanding of the program’s behavior.

The analytical idea is startlingly simple. We look at the source of the program and list the operations it performs. For each operation, we look up what it costs. We are going to focus on one kind of cost, namely running time. There

are many other other kinds of costs one can compute. We might naturally be interested in space (memory) consumed, which tells us how big a machine we need to buy. We might also care about power, this tells us the cost of our energy bills, or of bandwidth, which tells us what kind of Internet connection we will need. In general, then, we're interested in resource consumption. In short, don't make the mistake of equating "performance" with "speed": the costs that matter depend on the context in which the application runs. We add up these costs for all the operations. This gives us a total cost for the program.

Naturally, for most programs the answer will not be a constant number. Rather, it will depend on factors such as the size of the input. Therefore, our answer is likely to be an expression in terms of parameters (such as the input's size). In other words, our answer will be a function.

There are many functions that can describe the running-time of a function. Often we want an upper bound on the running time: i.e., the actual number of operations will always be no more than what the function predicts. This tells us the maximum resource we will need to allocate. Another function may present a lower bound, which tells us the least resource we need. Sometimes we want an average-case analysis. And so on. In this text we will focus on upper-bounds, but keep in mind that all these other analyses are also extremely valuable.

Exercise

It is incorrect to speak of "the" upper-bound function, because there isn't just one. Given one upper-bound function, can you construct another one?

13.3 A Cost Model for Pyret Running Time We begin by presenting a cost model for the running time of Pyret programs. We are interested in the cost of running a program, which is tantamount to studying the expressions of a program. Simply making a definition does not cost anything; the cost is incurred only when we use a definition.

We will use a very simple (but sufficiently accurate) cost model: every operation costs one unit of time in addition to the time needed to evaluate its sub-expressions. Thus it takes one unit of time to look up a variable or to allocate a constant. Applying primitive functions also costs one unit of time. Everything else is a compound expression with sub-expressions. The cost of a compound expression is one plus that of each of its sub-expressions. For instance, the running time cost of the expression `e1 + e2` (for some sub-expressions `e1` and `e2`) is the running time for `e1` + the running time for `e2 + 1`. Thus the expression `17 + 29` has a cost of 3 (one for each sub-expression and one for the addition); the expression `1 + (7 * (2 / 9))` costs 7.

As you can see, there are two big approximations here:

- First, we are using an abstract rather than concrete notion of time. This is unhelpful in terms of estimating the so-called "wall clock" running time of a program, but then again, that number depends on numerous factors—not just what kind of processor and how much memory you have, but even what other tasks are running on your computer at the same time. In contrast, abstract time units are more portable.
- Second, not every operation takes the same number of machine cycles, whereas we have charged all of them the same number of abstract time units. As long as the actual number of cycles each one takes is bounded by a constant factor of the number taken by another, this will not pose any mathematical problems for reasons we will soon understand [Comparing Functions].

Of course, it is instructive—after carefully setting up the experimental conditions—to make an analytical prediction of a program's behavior and then verify it against what the implementation actually does. If the analytical prediction is accurate, we can reconstruct the constant factors hidden in our calculations and thus obtain very precise wall-clock time bounds for the program.

There is one especially tricky kind of expression: `if` (and its fancier cousins, like `cases` and `ask`). How do we think about the cost of an `if`? It always evaluates the condition. After that, it evaluates only one of its branches. But we are interested in the worst case time, i.e., what is the longest it could take? For a conditional, it's the cost of the condition added to the cost of the maximum of the two branches.

13.4 The Size of the Input We gloss over the size of a number, treating it as constant. Observe that the value of a number is exponentially larger than its size: $\lfloor n \rfloor$ digits in base $\lfloor b \rfloor$ can represent $\lfloor b^n \rfloor$ numbers. Though irrelevant here, when numbers are central—e.g., when testing primality—the difference becomes critical! We will return to this briefly later [The Complexity of Numbers].

It can be subtle to define the size of the argument. Suppose a function consumes a list of numbers; it would be natural to define the size of its argument to be the length of the list, i.e., the number of `links` in the list. We could also define it to be twice as large, to account for both the `links` and the individual numbers (but as we'll see [Comparing Functions], constants usually don't matter). But suppose a function consumes a list of music albums, and each music album is itself a list of songs, each of which has information about singers and so on. Then how we measure the size depends on what part of the input the function being analyzed actually examines. If, say, it only returns the length of the list of albums, then it is indifferent to what each list element contains [Monomorphic Lists and Polymorphic Types], and only the length of the list of albums matters. If, however, the function returns a list of all the singers on every album, then it traverses all the way down to individual songs, and we have to account for all these data. In short, we care about the size of the data potentially accessed by the function.

13.5 The Tabular Method for Singly-Structurally-Recursive Functions Given sizes for the arguments, we simply examine the body of the function and add up the costs of the individual operations. Most interesting functions are, however, conditionally defined, and may even recur. Here we will assume there is only one structural recursive call. We will get to more general cases in a bit [Creating Recurrences].

When we have a function with only one recursive call, and it's structural, there's a handy technique we can use to handle conditionals. This idea is due to Prabhakar Ragde. We will set up a table. It won't surprise you to hear that the table will have as many rows as the `cond` has clauses. But instead of two columns, it has seven! This sounds daunting, but you'll soon see where they come from and why they're there.

For each row, fill in the columns as follows:

1. $|Q|$: the number of operations in the question
2. $\#Q$: the number of times the question will execute
3. Tot_Q : the total cost of the question (multiply the previous two)
4. $|A|$: the number of operations in the answer
5. $\#A$: the number of times the answer will execute
6. Tot_A : the total cost of the answer (multiply the previous two)
7. Total: add the two totals to obtain an answer for the clause

Finally, the total cost of the `cond` expression is obtained by summing the Total column in the individual rows.

In the process of computing these costs, we may come across recursive calls in an answer expression. So long as there is only one recursive call in the entire answer, ignore it.

Exercise

Once you've read the material on Creating Recurrences, come back to this and justify why it is okay to just skip the recursive call. Explain in the context of the overall tabular method.

Exercise

Excluding the treatment of recursion, justify (a) that these columns are individually accurate (e.g., the use of additions and multiplications is appropriate), and (b) sufficient (i.e., combined, they account for all operations that will be performed by that `cond` clause).

It's easiest to understand this by applying it to a few examples. First, let's consider the `len` function, noting before we proceed that it does meet the criterion of having a single recursive call where the argument is structural:

```
fun len(l):
  cases (List) l:
    | empty => 0
    | link(f, r) => 1 + len(r)
  end
end
```

Let's compute the cost of running `len` on a list of length $\backslash(k\backslash)$ (where we are only counting the number of `links` in the list, and ignoring the content of each first element (`f`), since `len` ignores them too).

Because the entire body of `len` is given by a conditional, we can proceed directly to building the table.

Let's consider the first row. The question costs three units (one each to evaluate the implicit `empty`-ness predicate, `l`, and to apply the former to the latter). This is evaluated once per element in the list and once more when the list is empty, i.e., $\lfloor(k+1\rfloor)$ times. The total cost of the question is thus $\lfloor(3(k+1)\rfloor)$. The answer takes one unit of time to compute, and is evaluated only once (when the list is empty). Thus it takes a total of one unit, for a total of $\lfloor(3k+4\rfloor)$ units.

Now for the second row. The question again costs three units, and is evaluated $\lfloor(k\rfloor)$ times. The answer involves two units to evaluate the rest of the list `l.rest`, which is implicitly hidden by the naming of `r`, two more to evaluate and apply `l +`, one more to evaluate `len`...and no more, because we are ignoring the time spent in the recursive call itself. In short, it takes five units of time (in addition to the recursion we've chosen to ignore).

In tabular form:

$|Q|$

$\#Q$

$TotQ$

$|A|$

$\#A$

$TotA$

Total

$\lfloor(3\rfloor)$

$\lfloor(k+1\rfloor)$

$\lfloor(3(k+1)\rfloor)$

$\lfloor(1\rfloor)$

$\lfloor(1\rfloor)$

$\lfloor(1\rfloor)$

$\lfloor(3k+4\rfloor)$

$\lfloor(3\rfloor)$

$\lfloor(k\rfloor)$

$\backslash(3k\backslash)$

$\backslash(5\backslash)$

$\backslash(k\backslash)$

$\backslash(5k\backslash)$

$\backslash(8k\backslash)$

Adding, we get $\backslash(11k + 4\backslash)$. Thus running `len` on a $\backslash(k\backslash)$ -element list takes $\backslash(11k+4\backslash)$ units of time.

Exercise

How accurate is this estimate? If you try applying `len` to different sizes of lists, do you obtain a consistent estimate for $\backslash(k\backslash)$?

13.6 Creating Recurrences We will now see a systematic way of analytically computing the time of a program. Suppose we have only one function `f`. We will define a function, $\backslash(T\backslash)$, to compute an upper-bound of the time of `f`. In general, we will have one such cost function for each function in the program. In such cases, it would be useful to give a different name to each function to easily tell them apart. Since we are looking at only one function for now, we'll reduce notational overhead by having only one $\backslash(T\backslash)$. $\backslash(T\backslash)$ takes as many parameters as `f` does. The parameters to $\backslash(T\backslash)$ represent the sizes of the corresponding arguments to `f`. Eventually we will want to arrive at a closed form solution to $\backslash(T\backslash)$, i.e., one that does not refer to $\backslash(T\backslash)$ itself. But the easiest way to get there is to write a solution that is permitted to refer to $\backslash(T\backslash)$, called a recurrence relation, and then see how to eliminate the self-reference [Solving Recurrences].

We repeat this procedure for each function in the program in turn. If there are many functions, first solve for the one with no dependencies on other functions, then use its solution to solve for a function that depends only on it, and progress thus up the dependency chain. That way, when we get to a function that refers to other functions, we will already have a closed-form solution for the referred function's running time and can simply plug in parameters to obtain a solution.

Exercise

The strategy outlined above doesn't work when there are functions that depend on each other.

How would you generalize it to handle this case?

The process of setting up a recurrence is easy. We simply define the right-hand-side of $\backslash(T\backslash)$ to add up the operations performed in `f`'s body. This is straightforward except for conditionals and recursion. We'll elaborate on the treatment of conditionals in a moment. If we get to a recursive call to `f` on the argument `a`, in the recurrence we turn this into a (self-)reference to $\backslash(T\backslash)$ on the size of `a`.

For conditionals, we use only the $|Q|$ and $|A|$ columns of the corresponding table. Rather than multiplying by the size of the input, we add up the operations that happen on one invocation of `f` other than the recursive call, and then add the cost of the recursive call in terms of a reference to $\backslash(T\backslash)$. Thus, if we were doing this for `len` above, we would define $\backslash(T(k)\backslash)$ —the time needed on an input of length $\backslash(k\backslash)$ —in two parts: the value of $\backslash(T(0)\backslash)$ (when the list is empty) and the value for non-zero values of $\backslash(k\backslash)$. We know that $\backslash(T(0) = 4\backslash)$ (the cost of the first conditional and its corresponding answer). If the list is non-empty, the cost is $\backslash(T(k) = 3 + 3 + 5 + T(k-1)\backslash)$ (respectively from the first question, the second question, the remaining operations in the second answer, and the recursive call on a list one element smaller). This gives the following recurrence:

```
\begin{equation*}T(k) = \begin{cases} 4 & \text{when } k = 0 \\ 11 + T(k-1) & \text{when } k > 0 \end{cases}\end{equation*}
```

For a given list that is $\lfloor p \rfloor$ elements long (note that $\lfloor p \geq 0 \rfloor$), this would take $\lfloor 11 \rfloor$ steps for the first element, $\lfloor 11 \rfloor$ more steps for the second, $\lfloor 11 \rfloor$ more for the third, and so on, until we run out of list elements and need $\lfloor 4 \rfloor$ more steps: a total of $\lfloor 11p + 4 \rfloor$ steps. Notice this is precisely the same answer we obtained by the tabular method!

Exercise

Why can we assume that for a list $\lfloor p \rfloor$ elements long, $\lfloor p \geq 0 \rfloor$? And why did we take the trouble to explicitly state this above?

With some thought, you can see that the idea of constructing a recurrence works even when there is more than one recursive call, and when the argument to that call is one element structurally smaller. What we haven't seen, however, is a way to solve such relations in general. That's where we're going next [Solving Recurrences].

13.7 A Notation for Functions We have seen above that we can describe the running time of `len` through a function. We don't have an especially good notation for writing such (anonymous) functions. Wait, we do—`lam(k) : (11 * k) + 4 end`—but my colleagues would be horrified if you wrote this on their exams. Therefore, we'll introduce the following notation to mean precisely the same thing:

$$\begin{array}{l} \text{\begin{equation*}} \\ [k \rightarrow 11k + 4] \\ \text{\end{equation*}} \end{array}$$

The brackets denote anonymous functions, with the parameters before the arrow and the body after.

13.8 Comparing Functions Let's return to the running time of `len`. We've written down a function of great precision: $11! 4!$ Is this justified?

At a fine-grained level already, no, it's not. We've lumped many operations, with different actual running times, into a cost of one. So perhaps we should not worry too much about the differences between, say, $\lfloor [k \rightarrow 11k + 4] \rfloor$ and $\lfloor [k \rightarrow 4k + 10] \rfloor$. If we were given two implementations with these running times, respectively, it's likely that we would pick other characteristics to choose between them.

What this boils down to is being able to compare two functions (representing the performance of implementations) for whether one is somehow quantitatively better in some meaningful sense than the other: i.e., is the quantitative difference so great that it might lead to a qualitative one. The example above suggests that small differences in constants probably do not matter.

That is, we want a way to compare two functions, $\lfloor f_1 \rfloor$ and $\lfloor f_2 \rfloor$. What does it mean for $\lfloor f_1 \rfloor$ to be "less" than $\lfloor f_2 \rfloor$, without worrying about constants? We obtain this definition:

$$\begin{array}{l} \text{\begin{equation*}} \\ \exists c . \forall n \in \mathbb{N}, f_1(n) \leq c \cdot f_2(n) \rightarrow f_1 \leq f_2 \text{\end{equation*}} \end{array}$$

This says that for all natural numbers ($\lfloor N \rfloor$), the value of $\lfloor f_1 \rfloor$ will always be less than the value of $\lfloor f_2 \rfloor$. However, to accommodate our intuition that multiplicative constants don't matter, the definition allows the value of $\lfloor f_2 \rfloor$ at all points to be multiplied by some constant $\lfloor c \rfloor$ to achieve the inequality. Observe, however, that $\lfloor c \rfloor$ is independent of $\lfloor n \rfloor$: it is chosen once and must then work for the infinite number of values. In practice, this means that the presence of $\lfloor c \rfloor$ lets us bypass some number of early values where $\lfloor f_1 \rfloor$ might have a greater value than $\lfloor f_2 \rfloor$, so long as, after a point, $\lfloor f_2 \rfloor$ dominates $\lfloor f_1 \rfloor$.

This definition has more flexibility than we might initially think. For instance, consider our running example compared with $\lfloor [k \rightarrow k^2] \rfloor$. Clearly, the latter function eventually dominates the former: i.e.,

$$\begin{array}{l} \text{\begin{equation*}} \\ [k \rightarrow 11k+4] \leq [k \rightarrow k^2] \text{\end{equation*}} \end{array}$$

We just need to pick a sufficiently large constant and we will find this to be true.

Exercise

What is the smallest constant that will suffice?

You will find more complex definitions in the literature and they all have merits, because they enable us to make finer-grained distinctions than this definition allows. For the purpose of this book, however, the above definition suffices.

Do Now!

Why are the quantifiers written in this and not the opposite order? What if we had swapped them, so that we could choose a $\forall(c)$ for each $\forall(n)$?

Had we swapped the order, it would mean that for every point along the number line, there must exist a constant—and there pretty much always does! The swapped definition would therefore be useless. What is important is that we can identify the constant no matter how large the parameter gets. That is what makes this truly a constant.

Observe that for a given function $\forall(f)$, there are numerous functions that are less than it. We use the notation $\forall(O(\cdot))$ to describe this family of functions. In computer science this is usually pronounced “big-Oh”, though some prefer to call it the Bachmann-Landau notation after its originators. Thus if $\forall(g \leq f)$, we can write $\forall(g \in O(f))$, which we can read as “ $\forall(f)$ is an upper-bound for $\forall(g)$ ”. Thus, for instance,

$$\begin{aligned} & \forall(k \rightarrow 3k) \in O([k \rightarrow 4k+12]) \end{aligned}$$

$$\begin{aligned} & \forall(k \rightarrow 4k+12) \in O([k \rightarrow k^2]) \end{aligned}$$

and so on. Obviously, the “bigger” function is likely to be a less useful bound than a “tighter” one. That said, it is conventional to write a “minimal” bound for functions, which means avoiding unnecessary constants, sum terms, and so on. The justification for this is given below [Combining Big-Oh Without Woe].

Pay especially close attention to our notation. We write $\forall(\in)$ rather than $\forall(=)$ or some other symbol, because $\forall(O(f))$ describes a family of functions of which $\forall(g)$ is a member. We also write $\forall(f)$ rather than $\forall(f(x))$ because we are comparing functions— $\forall(f)$ —rather than their values at particular points— $\forall(f(x))$ —which would be ordinary numbers! Most of the notation in most the books and Web sites suffers from one or both flaws. We know, however, that functions are values, and that functions can be anonymous. We have actually exploited both facts to be able to write

$$\begin{aligned} & \forall(k \rightarrow 3k) \in O([k \rightarrow 4k+12]) \end{aligned}$$

This is not the only notion of function comparison that we can have. For instance, given the definition of $\forall(\leq)$ above, we can define a natural relation $\forall(<)$. This then lets us ask, given a function $\forall(f)$, what are all the functions $\forall(g)$ such that $\forall(g \leq f)$ but not $\forall(g < f)$, i.e., those that are “equal” to $\forall(f)$. Look out! We are using quotes because this is not the same as ordinary function equality, which is defined as the two functions giving the same answer on all inputs. Here, two “equal” functions may not give the same answer on any inputs. This is the family of functions that are separated by at most a constant; when the functions indicate the order of growth of programs, “equal” functions signify programs that grow at the same speed (up to constants). We use the notation $\forall(\Theta(\cdot))$ to speak of this family of functions, so if $\forall(g)$ is equivalent to $\forall(f)$ by this notion, we can write $\forall(g \in \Theta(f))$ (and it would then also be true that $\forall(f \in \Theta(g))$).

Exercise

Convince yourself that this notion of function equality is an equivalence relation, and hence worthy of the name “equal”. It needs to be (a) reflexive (i.e., every function is related to itself); (b) antisymmetric (if $\forall(f \leq g)$ and $\forall(g \leq f)$ then $\forall(f)$ and $\forall(g)$ are equal); and (c) transitive ($\forall(f \leq g)$ and $\forall(g \leq h)$ implies $\forall(f \leq h)$).

13.9 Combining Big-Oh Without Woe Now that we’ve introduced this notation, we should inquire about its closure properties: namely, how do these families of functions combine? To nudge your intuitions, assume that in all cases we’re discussing the running time of functions. We’ll consider three cases:

- Suppose we have a function f (whose running time is) in $\forall(O(F))$. Let’s say we run it $\forall(p)$ times, for some given constant. The running time of the resulting code is then $\forall(p \times O(F))$. However, observe that this is really no different from $\forall(O(F))$: we can simply use a bigger constant for $\forall(c)$ in the definition of $\forall(O(\cdot))$ —in particular, we can just use $\forall(pc)$. Conversely, then, $\forall(O(pf))$ is equivalent to $\forall(O(F))$. This is the heart of the intuition that “multiplicative constants don’t matter”.
- Suppose we have two functions, f in $\forall(O(F))$ and g in $\forall(O(G))$. If we run f followed by g , we would expect the running time of the combination to be the sum of their individual running times, i.e., $\forall(O(F) + O(G))$. You should convince yourself that this is simply $\forall(O(\max(F, G)))$.
- Suppose we have two functions, f in $\forall(O(F))$ and g in $\forall(O(G))$. If f invokes g in each of its steps, we would expect the running time of the combination to be the product of their individual running times, i.e., $\forall(O(F) \times O(G))$. You should convince yourself that this is simply $\forall(O(F \times G))$.

These three operations—addition, multiplication by a constant, and multiplication by a function—cover just about all the cases. To ensure that the table fits in a reasonable width, we will abuse notation. For instance, we can use this to reinterpret the tabular operations above (assuming everything is a function of $\langle k \rangle$):

$|Q|$

$\#Q$

$TotQ$

$|A|$

$\#A$

$TotA$

Total

$\langle O(1) \rangle$

$\langle O(k) \rangle$

$\langle O(k) \rangle$

$\langle O(1) \rangle$

$\langle O(1) \rangle$

$\langle O(1) \rangle$

$\langle O(k) \rangle$

$\langle O(k) \rangle$

$\langle O(1) \rangle$

$\langle O(k) \rangle$

$\backslash(O(k)\backslash)$

$\backslash(O(k)\backslash)$

Because multiplication by constants doesn't matter, we can replace the $\backslash(3\backslash)$ with $\backslash(1\backslash)$. Because addition of a constant doesn't matter (run the addition rule in reverse), $\backslash(k+1\backslash)$ can become $\backslash(k\backslash)$. Adding this gives us $\backslash(O(k) + O(k) = 2 \backslash\text{times } O(k) \backslash\text{in } O(k)\backslash)$. This justifies claiming that running `len` on a $\backslash(k\backslash)$ -element list takes time in $\backslash(O([k \rightarrow k])\backslash)$, which is a much simpler way of describing its bound than $\backslash(O([k \rightarrow 11k + 4])\backslash)$. In particular, it provides us with the essential information and nothing else: as the input (list) grows, the running time grows proportional to it, i.e., if we add one more element to the input, we should expect to add a constant more of time to the running time.

13.10 Solving Recurrences There is a great deal of literature on solving recurrence equations. In this section we won't go into general techniques, nor will we even discuss very many different recurrences. Rather, we'll focus on just a handful that should be in the repertoire of every computer scientist. You'll see these over and over, so you should instinctively recognize their recurrence pattern and know what complexity they describe (or know how to quickly derive it).

Earlier we saw a recurrence that had two cases: one for the empty input and one for all others. In general, we should expect to find one case for each non-recursive call and one for each recursive one, i.e., roughly one per `cases` clause. In what follows, we will ignore the base cases so long as the size of the input is constant (such as zero or one), because in such cases the amount of work done will also be a constant, which we can generally ignore [Comparing Functions].

- $\backslash(T(k)\backslash)$
- $\backslash(T(k)\backslash)$
- $\backslash(T(k)\backslash)$
- $\backslash(T(k)\backslash)$
- $\backslash(T(k)\backslash)$
- $\backslash(T(k)\backslash)$

Exercise

Using induction, prove each of the above derivations.

contents ← prev up next →

14 Halloween Analysis

14 Halloween Analysis

In Predicting Growth, we introduced the idea of big-Oh complexity to measure the worst-case time of a computation. As we see in Choosing Between Representations, however, this is sometimes too coarse a bound when the complexity is heavily dependent on the exact sequence of operations run. Now, we will consider a different style of complexity analysis that better accommodates operation sequences.

14.1 A First Example Consider, for instance, a set that starts out empty, followed by a sequence of $\langle k \rangle$ insertions and then $\langle k \rangle$ membership tests, and suppose we are using the representation without duplicates. Insertion time is proportional to the size of the set (and list); this is initially $\langle 0 \rangle$, then $\langle 1 \rangle$, and so on, until it reaches size $\langle k \rangle$. Therefore, the total cost of the sequence of insertions is $\langle k \cdot (k+1) / 2 \rangle$. The membership tests cost $\langle k \rangle$ each in the worst case, because we've inserted up to $\langle k \rangle$ distinct elements into the set. The total time is then

$$\begin{aligned} & \text{\backslash begin\{equation*}\} k^2 / 2 + k / 2 + k^2 \text{\end\{equation*}} \end{aligned}$$

for a total of $\langle 2k \rangle$ operations, yielding an average of

$$\begin{aligned} & \text{\backslash begin\{equation*}\} \frac{3}{4} k + \frac{1}{4} \text{\end\{equation*}} \end{aligned}$$

steps per operation in the worst case.

14.2 The New Form of Analysis What have we computed? We are still computing a worst case cost, because we have taken the cost of each operation in the sequence in the worst case. We are then computing the average cost per operation. Therefore, this is a average of worst cases. Importantly, this is different from what is known as average-case analysis, which uses probability theory to compute the estimated cost of the computation. We have not used any probability here. Note that because this is an average per operation, it does not say anything about how bad any one operation can be (which, as we will see [Amortization Versus Individual Operations], can be quite a bit worse); it only says what their average is.

In the above case, this new analysis did not yield any big surprises. We have found that on average we spend about $\langle k \rangle$ steps per operation; a big-Oh analysis would have told us that we're performing $\langle 2k \rangle$ operations with a cost of $\langle O([k \rightarrow k]) \rangle$ each in the number of distinct elements; per operation, then, we are performing roughly linear work in the worst-case number of set elements.

As we will soon see, however, this won't always be the case: this new analysis can cough up pleasant surprises.

Before we proceed, we should give this analysis its name. Formally, it is called amortized analysis. Amortization is the process of spreading a payment out over an extended but fixed term. In the same way, we spread out the cost of a computation over a fixed sequence, then determine how much each payment will be. We have given it a whimsical name because Halloween is a(n American) holiday devoted to ghosts, ghouls, and other symbols of death. Amortization comes from the Latin root mort-, which means death, because an amortized analysis is one conducted “at the death”, i.e., at the end of a fixed sequence of operations.

14.3 An Example: Queues from Lists We have seen lists [From Tables to Lists] and sets [Several Variations on Sets]. Here we focus on queues, which too can be represented as lists: Queues from Lists. If you have not read that material, it's worth reading at least the early portions now. In this section, we will ignore the various programming niceties discussed there, and focus on raw list representations to make an algorithmic point.

14.3.1 List Representations Consider two natural ways of defining queues using lists. One is that every enqueue is implemented with `link`, while every dequeue requires traversing the whole list until its end. Conversely, we could make enqueueing traverse to the end, and dequeuing correspond to `.rest`. Either way, one of these operations will take constant time while the other will be linear in the length of the list representing the queue. (This should be loosely reminiscent of trade-offs we ran into when representing sets as lists: Representing Sets as Lists.)

In fact, however, the above paragraph contains a key insight that will let us do better.

Observe that if we store the queue in a list with most-recently-enqueued element first, enqueueing is cheap (constant time). In contrast, if we store the queue in the reverse order, then dequeuing is constant time. It would be wonderful if we could have both, but once we pick an order we must give up one or the other. Unless, that is, we pick...both.

One half of this is easy. We simply enqueue elements into a list with the most recent addition first. Now for the (first) crucial insight: when we need to dequeue, we reverse the list. Now, dequeuing also takes constant time.

14.3.2 A First Analysis Of course, to fully analyze the complexity of this data structure, we must also account for the reversal. In the worst case, we might argue that any operation might reverse (because it might be the first dequeue); therefore, the worst-case time of any operation is the time it takes to reverse, which is linear in the length of the list (which corresponds to the elements of the queue).

However, this answer should be unsatisfying. If we perform $\mathcal{O}(k)$ enqueues followed by $\mathcal{O}(k)$ dequeues, then each of the enqueues takes one step; each of the last $\mathcal{O}(k-1)$ dequeues takes one step; and only the first dequeue requires a reversal, which takes steps proportional to the number of elements in the list, which at that point is $\mathcal{O}(k)$. Thus, the total cost of operations for this sequence is $\mathcal{O}(k \cdot 1 + k + (k-1) \cdot 1 = 3k-1)$ for a total of $\mathcal{O}(2k)$ operations, giving an amortized complexity of effectively constant time per operation!

14.3.3 More Liberal Sequences of Operations In the process of this, however, we've quietly glossed over something that you may not have picked up on: in our candidate sequence all dequeues followed all enqueues. What happens on the next enqueue? Because the list is now reversed, it will have to take a linear amount of time! So we have only partially solved the problem.

Now we can introduce the second insight: have two lists instead of one. One of them will be the tail of the queue, where new elements get enqueued; the other will be the head of the queue, where they get dequeued:

```
data Queue<T>:
    | queue(tail :: List<T>, head :: List<T>)
end

mt-q :: Queue = queue(empty, empty)
```

Provided the tail is stored so that the most recent element is the first, then enqueueing takes constant time:

```
fun enqueue<T>(q :: Queue<T>, e :: T) -> Queue<T>:
    queue(link(e, q.tail), q.head)
end
```

For dequeuing to take constant time, the head of the queue must be stored in the reverse direction. However, how does any element ever get from the tail to the head? Easy: when we try to dequeue and find no elements in the head, we reverse the (entire) tail into the head (resulting in an empty tail). We will first define a datatype to represent the response from dequeuing:

```
data Response<T>:
    | elt-and-q(e :: T, r :: Queue<T>)
end
```

Now for the implementation of `dequeue`:

```
fun dequeue<T>(q :: Queue<T>) -> Response<T>:
    cases (List) q.head:
        | empty =>
            new-head = q.tail.reverse()
            elt-and-q(new-head.first,
```

```

    queue(empty, new-head.rest)
| link(f, r) =>
  elt-and-q(f,
    queue(q.tail, r))
end
end

```

14.3.4 A Second Analysis We can now reason about sequences of operations as we did before, by adding up costs and averaging. However, another way to think of it is this. Let's give each element in the queue three "credits". Each credit can be used for one constant-time operation.

One credit gets used up in enqueueing. So long as the element stays in the tail list, it still has two credits to spare. When it needs to be moved to the head list, it spends one more credit in the link step of reversal. Finally, the dequeuing operation performs one operation too.

Because the element does not run out of credits, we know it must have had enough. These credits reflect the cost of operations on that element. From this (very informal) analysis, we can conclude that in the worst case, any permutation of enqueues and dequeues will still cost only a constant amount of amortized time.

14.3.5 Amortization Versus Individual Operations Note, however, that the constant represents an average across the sequence of operations. It does not put a bound on the cost of any one operation. Indeed, as we have seen above, when dequeue finds the head list empty it reverses the tail, which takes time linear in the size of the tail—not constant at all! Therefore, we should be careful to not assume that every step in the sequence will is bounded. Nevertheless, an amortized analysis sometimes gives us a much more nuanced understanding of the real behavior of a data structure than a worst-case analysis does on its own.

14.4 Reading More At this point we have only briefly touched on the subject of amortized analysis. A very nice tutorial by Rebecca Fiebrink provides much more information. The authoritative book on algorithms, *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein, covers amortized analysis in extensive detail.

contents ← prev up next →

15.1 Sharing and Equality

15.1 Sharing and Equality

15.1.1 Re-Examining Equality Consider the following data definition and example values:

```
data BinTree:  
| leaf  
| node(v, l :: BinTree, r :: BinTree)  
end  
  
a-tree =  
node(5,  
    node(4, leaf, leaf),  
    node(4, leaf, leaf))  
  
b-tree =  
block:  
    four-node = node(4, leaf, leaf)  
    node(5,  
        four-node,  
        four-node)  
end
```

In particular, it might seem that the way we've written `b-tree` is morally equivalent to how we've written `a-tree`, but we've created a helpful binding to avoid code duplication.

Because both `a-tree` and `b-tree` are bound to trees with 5 at the root and a left and right child each containing 4, we can indeed reasonably consider these trees equivalent. Sure enough:

```
<equal-tests> ::=  
  
check:  
    a-tree  is b-tree  
    a-tree.l is a-tree.l  
    a-tree.l is a-tree.r  
    b-tree.l is b-tree.r  
end
```

However, there is another sense in which these trees are not equivalent. concretely, `a-tree` constructs a distinct node for each child, while `b-tree` uses the same node for both children. Surely this difference should show up somehow, but we have not yet seen a way to write a program that will tell these apart.

By default, the `is` operator uses the same equality test as Pyret's `==`. There are, however, other equality tests in Pyret. In particular, the way we can tell apart these data is by using Pyret's `identical` function, which implements reference equality. This checks not only whether two values are structurally equivalent but whether they are the result of the very same act of value construction. With this, we can now write additional tests:

```
check:  
    identical(a-tree, b-tree)      is false  
    identical(a-tree.l, a-tree.l)  is true  
    identical(a-tree.l, a-tree.r)  is false
```

```
    identical(b-tree.l, b-tree.r) is true
end
```

Let's step back for a moment and consider the behavior that gives us this result. We can visualize the different values by putting each distinct value in a separate location alongside the running program. We can draw the first step as creating a `node` with value 4:

```
a-tree =
  node(5,
    1001,
    node(4, leaf, leaf))

b-tree =
  block:
    four-node = node(4, leaf, leaf)
    node(5,
      four-node,
      four-node)
  end
```

Heap

- 1001:
 node(4, leaf, leaf)

The next step creates another node with value 4, distinct from the first:

```
a-tree =
  node(5, 1001, 1002)

b-tree =
  block:
    four-node = node(4, leaf, leaf)
    node(5,
      four-node,
      four-node)
  end
```

Heap

- 1001:
 node(4, leaf, leaf)
- 1002:
 node(4, leaf, leaf)

Then the `node` for `a-tree` is created:

```
a-tree = 1003

b-tree =
  block:
    four-node = node(4, leaf, leaf)
    node(5,
      four-node,
      four-node)
  end
```

Heap

- 1001:

```

    node(4, leaf, leaf)
• 1002:
    node(4, leaf, leaf)
• 1003:
    node(5, 1001, 1002)

```

When evaluating the `b-tree` for `b-tree`, first a single node is created for the `four-node` binding:

`a-tree = 1003`

```

b-tree =
block:
four-node = 1004
node(5,
    four-node,
    four-node)
end

```

Heap

- 1001:


```
node(4, leaf, leaf)
```
- 1002:


```
node(4, leaf, leaf)
```
- 1003:


```
node(5, 1001, 1002)
```
- 1004:


```
node(4, leaf, leaf)
```

These location values can be substituted just like any other, so they get substituted for `four-node` to continue evaluation of the block. We skipped substituting `a-tree` for the moment, that will come up later.

`a-tree = 1003`

```

b-tree =
block:
node(5, 1004, 1004)
end

```

Heap

- 1001:


```
node(4, leaf, leaf)
```
- 1002:


```
node(4, leaf, leaf)
```
- 1003:


```
node(5, 1001, 1002)
```
- 1004:


```
node(4, leaf, leaf)
```

Finally, the node for `b-tree` is created:

```
a-tree = 1003
```

```
b-tree = 1005
```

Heap

- 1001:
 node(4, leaf, leaf)
- 1002:
 node(4, leaf, leaf)
- 1003:
 node(5, 1001, 1002)
- 1004:
 node(4, leaf, leaf)
- 1005:
 node(5, 1004, 1004)

This visualization can help us explain the test we wrote using `identical`. Let's consider the test with the appropriate location references substituted for `a-tree` and `b-tree`:

```
check:  
  identical(1003, 1005)  
    is false  
  identical(1003.l, 1003.l)  
    is true  
  identical(1003.l, 1003.r)  
    is false  
  identical(1005.l, 1005.r)  
    is true  
end
```

Heap

- 1001:
 node(4, leaf, leaf)
- 1002:
 node(4, leaf, leaf)
- 1003:
 node(5, 1001, 1002)
- 1004:
 node(4, leaf, leaf)
- 1005:
 node(5, 1004, 1004)

```
check:  
  identical(1003, 1005)  
    is false  
  identical(1001, 1001)  
    is true  
  identical(1001, 1004)  
    is false
```

```

identical(1004, 1004)
    is true
end

```

Heap

- 1001:
node(4, leaf, leaf)
- 1002:
node(4, leaf, leaf)
- 1003:
node(5, 1001, 1002)
- 1004:
node(4, leaf, leaf)
- 1005:
node(5, 1004, 1004)

There is actually another way to write these tests in Pyret: the `is` operator can also be parameterized by a different equality predicate than the default `==`. Thus, the above block can equivalently be written as: We can use `is-not` to check for expected failure of equality.

```

check:
  a-tree  is-not%(identical) b-tree
  a-tree.l is%(identical)      a-tree.l
  a-tree.l is-not%(identical) a-tree.r
  b-tree.l is%(identical)      b-tree.r
end

```

We will use this style of equality testing from now on.

Observe how these are the same values that were compared earlier (`<equal-tests>`), but the results are now different: some values that were true earlier are now false. In particular,

```

check:
  a-tree  is           b-tree
  a-tree  is-not%(identical) b-tree
  a-tree.l is           a-tree.r
  a-tree.l is-not%(identical) a-tree.r
end

```

Later we will return both to what `identical` really means [Understanding Equality] (Pyret has a full range of equality operations suitable for different situations).

Exercise

There are many more equality tests we can and should perform even with the basic data above to make sure we really understand equality and, relatedly, storage of data in memory. What other tests should we conduct? Predict what results they should produce before running them!

15.1.2 The Cost of Evaluating References From a complexity viewpoint, it's important for us to understand how these references work. As we have hinted, `four-node` is computed only once, and each use of it refers to the same value: if, instead, it was evaluated each time we referred to `four-node`, there would be no real difference between `a-tree` and `b-tree`, and the above tests would not distinguish between them.

This is especially relevant when understanding the cost of function evaluation. We'll construct two simple examples that illustrate this. We'll begin with a contrived data structure:

```
L = range(0, 100)
```

Suppose we now define

```
L1 = link(1, L)
L2 = link(-1, L)
```

Constructing a list clearly takes time at least proportional to the length; therefore, we expect the time to compute `L` to be considerably more than that for a single `link` operation. Therefore, the question is how long it takes to compute `L1` and `L2` after `L` has been computed: constant time, or time proportional to the length of `L`?

The answer, for Pyret, and for most other contemporary languages (including Java, C#, OCaml, Racket, etc.), is that these additional computations take constant time. That is, the value bound to `L` is computed once and bound to `L`; subsequent expressions refer to this value (hence “reference”) rather than reconstructing it, as reference equality shows:

```
check:
  L1.rest is%(identical) L
  L2.rest is%(identical) L
  L1.rest is%(identical) L2.rest
end
```

Similarly, we can define a function, pass `L` to it, and see whether the resulting argument is `identical` to the original:

```
fun check-for-no-copy(another-l):
  identical(another-l, L)
end
```

```
check:
  check-for-no-copy(L) is true
end
```

or, equivalently,

```
check:
  L satisfies check-for-no-copy
end
```

Therefore, neither built-in operations (like `.rest`) nor user-defined ones (like `check-for-no-copy`) make copies of their arguments. Strictly speaking, of course, we cannot conclude that no copy was made. Pyret could have made a copy, discarded it, and still passed a reference to the original. Given how perverse this would be, we can assume—and take the language’s creators’ word for it—that this doesn’t actually happen. By creating extremely large lists, we can also use timing information to observe that the time of constructing the list grows proportional to the length of the list while the time of passing it as a parameter remains constant. The important thing to observe here is that, instead of simply relying on authority, we have used operations in the language itself to understand how the language behaves.

15.1.3 Notations for Equality Until now we have used `==` for equality. Now we have learned that it’s only one of multiple equality operators, and that there is another one called `identical`. However, these two have somewhat subtly different syntactic properties. `identical` is a name for a function, which can therefore be used to refer to it like any other function (e.g., when we need to mention it in a `is-not` clause). In contrast, `==` is a binary operator, which can only be used in the middle of expressions.

This should naturally make us wonder about the other two possibilities: a binary expression version of `identical` and a function name equivalent of `==`. They do, in fact, exist! The operation performed by `==` is called `equal-always`. Therefore, we can write the first block of tests equivalently, but more explicitly, as

```
check:
  a-tree  is%(equal-always) b-tree
  a-tree.l is%(equal-always) a-tree.l
  a-tree.l is%(equal-always) a-tree.r
  b-tree.l is%(equal-always) b-tree.r
end
```

Conversely, the binary operator notation for `identical` is `<=>`. Thus, we can equivalently write `check-for-no-copy` as

```
fun check-for-no-copy(another-l):
    another-l <=> L
end
```

15.1.4 On the Internet, Nobody Knows You're a DAG Despite the name we've given it, `b-tree` is not actually a tree. In a tree, by definition, there are no shared nodes, whereas in `b-tree` the node named by `four-node` is shared by two parts of the tree. Despite this, traversing `b-tree` will still terminate, because there are no cyclic references in it: if you start from any node and visit its "children", you cannot end up back at that node. There is a special name for a value with such a shape: directed acyclic graph (DAG).

Many important data structures are actually a DAG underneath. For instance, consider Web sites. It is common to think of a site as a tree of pages: the top-level refers to several sections, each of which refers to sub-sections, and so on. However, sometimes an entry needs to be cataloged under multiple sections. For instance, an academic department might organize pages by people, teaching, and research. In the first of these pages it lists the people who work there; in the second, the list of courses; and in the third, the list of research groups. In turn, the courses might have references to the people teaching them, and the research groups are populated by these same people. Since we want only one page per person (for both maintenance and search indexing purposes), all these personnel links refer back to the same page for people.

Let's construct a simple form of this. First a datatype to represent a site's content:

```
data Content:
| page(s :: String)
| section(title :: String, sub :: List<Content>)
end
```

Let's now define a few people:

```
people-pages :: Content =
section("People",
[list: page("Church"),
page("Dijkstra"),
page("Hopper")])
```

and a way to extract a particular person's page:

```
fun get-person(n): get(people-pages.sub, n) end
```

Now we can define theory and systems sections:

```
theory-pages :: Content =
section("Theory",
[list: get-person(0), get-person(1)])
systems-pages :: Content =
section("Systems",
[list: get-person(1), get-person(2)])
```

which are integrated into a site as a whole:

```
site :: Content =
section("Computing Sciences",
[list: theory-pages, systems-pages])
```

Now we can confirm that each of these luminaries needs to keep only one Web page current; for instance:

```
check:
theory = get(site.sub, 0)
systems = get(site.sub, 1)
theory-dijkstra = get(theory.sub, 1)
systems-dijkstra = get(systems.sub, 0)
```

```

theory-dijkstra is           systems-dijkstra
theory-dijkstra is%(identical) systems-dijkstra
end

```

15.1.5 It's Always Been a DAG What we may not realize is that we've actually been creating a DAG for longer than we think. To see this, consider a-tree, which very clearly seems to be a tree. But look more closely not at the nodes but rather at the leaf(s). How many actual leafs do we create?

One hint is that we don't seem to call a function when creating a leaf: the data definition does not list any fields, and when constructing a BinTree value, we simply write leaf, not (say) leaf(). Still, it would be nice to know what is happening behind the scenes. To check, we can simply ask Pyret:

```

check:
  leaf is%(identical) leaf
end

```

It's important that we not write leaf \leftrightarrow leaf here, because that is just an expression whose result is ignored. We have to write is to register this as a test whose result is checked and reported. and this check passes. That is, when we write a variant without any fields, Pyret automatically creates a singleton: it makes just one instance and uses that instance everywhere. This leads to a more efficient memory representation, because there is no reason to have lots of distinct leafs each taking up their own memory. However, a subtle consequence of that is that we have been creating a DAG all along.

If we really wanted each leaf to be distinct, it's easy: we can write

```

data BinTreeDistinct:
  | leaf()
  | node(v, l :: BinTreeDistinct, r :: BinTreeDistinct)
end

```

Then we would need to use the leaf function everywhere:

```

c-tree :: BinTreeDistinct =
  node(5,
    node(4, leaf(), leaf()),
    node(4, leaf(), leaf()))

```

And sure enough:

```

check:
  leaf() is-not%(identical) leaf()
end

```

15.1.6 From Acyclicity to Cycles Here's another example that arises on the Web. Suppose we are constructing a table of output in a Web page. We would like the rows of the table to alternate between white and grey. If the table had only two rows, we could map the row-generating function over a list of these two colors. Since we do not know how many rows it will have, however, we would like the list to be as long as necessary. In effect, we would like to write:

```
web-colors = link("white", link("grey", web-colors))
```

to obtain an indefinitely long list, so that we could eventually write

```
map2(color-table-row, table-row-content, web-colors)
```

which applies a color-table-row function to two arguments: the current row from table-row-content, and the current color from web-colors, proceeding in lockstep over the two lists.

Unfortunately, there are many things wrong with this attempted definition.

Do Now!

Do you see what they are?

Here are some problems in turn:

- This will not even parse. The identifier `web-colors` is not bound on the right of the `=`.
- Earlier, we saw a solution to such a problem: use `rec` [Streams From Functions]. What happens if we write

```
rec web-colors = link("white", link("grey", web-colors))
```

instead?

Exercise

Why does `rec` work in the definition of `ones` but not above?

- Assuming we have fixed the above problem, one of two things will happen. It depends on what the initial value of `web-colors` is. Because it is a dummy value, we do not get an arbitrarily long list of colors but rather a list of two colors followed by the dummy value. Indeed, this program will not even type-check.

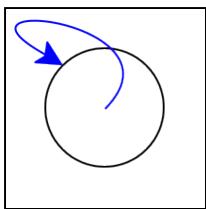
Suppose, however, that `web-colors` were written instead as a function definition to delay its creation:

```
fun web-colors(): link("white", link("grey", web-colors())) end
```

On its own this just defines a function. If, however, we use it—`web-colors()`—it goes into an infinite loop constructing links.

- Even if all that were to work, `map2` would either (a) not terminate because its second argument is indefinitely long, or (b) report an error because the two arguments aren't the same length.

All these problems are symptoms of a bigger issue. What we are trying to do here is not merely create a shared datum (like a DAG) but something much richer: a cyclic datum, i.e., one that refers back to itself:



When you get to cycles, even defining the datum becomes difficult because its definition depends on itself so it (seemingly) needs to already be defined in the process of being defined. We will return to cyclic data later in Cyclic Data, and to this specific example in Recursion and Cycles from Mutation.

contents ← prev up next →

15.2 The Size of a DAG

15.2 The Size of a DAG Let's start by defining a function to compute the size of a tree:

```
data BT:  
| mt  
| nd(v :: Number, l :: BT, r :: BT)  
end  
  
fun size-1(b :: BT) -> Number:  
cases (BT) b:  
| mt => 0  
| nd(v, l, r) => 1 + size-1(l) + size-1(r)  
end  
end
```

This is straightforward enough.

But let's say that our input isn't actually a tree, but rather a DAG. For instance:

```
#|  
 4  
 / \  
2   3  
 \ /  
  1  
|#  
  
n1 = nd(1, mt, mt)  
n2 = nd(2, mt, n1)  
n3 = nd(3, n1, mt)  
n4 = nd(4, n2, n3)
```

where `n4` is the DAG. There are two notions of size here. One is like a “print size”: how much space will it occupy when printed. The current size function computes that well. But another is the “allocation” size: how many nodes did we allocate. How do we fare?

15.2.1 Stage 1

```
check:  
size-1(n1) is 1  
size-1(n2) is 2  
size-1(n3) is 2  
size-1(n4) is 4  
end
```

Clearly the answer should be 4: we can just read off how many `nd` calls there are. And clearly the function is wrong.

The problem, of course, is that a DAG involves repeating nodes, and we aren't doing anything to track the repetition. So we need a stronger contract: we'll split the problem into two parts, a standard interface function that takes just the DAG and returns a number, and a richer helper function, which also takes a memory of the nodes already seen.

15.2.2 Stage 2

```
fun size-2-h(b :: BT, seen :: List<BT>) -> Number:
    if member-identical(seen, b):
        0
    else:
        cases (BT) b:
            | mt => 0
            | nd(v, l, r) =>
                new-seen = link(b, seen)
                1 + size-2-h(l, new-seen) + size-2-h(r, new-seen)
        end
    end
end
```

Exercise

Why does this code use `member-identical` rather than `member`?

Observe that if we replace every `member-identical` with `member` in this chapter, the code still behaves the same. Why?

Make changes to demonstrate the need for `member-identical`.

Is it odd that we return 0? Not if we reinterpret what the function does: it doesn't count the size, it counts the additional contribution to the size (relative to what has already been seen) of the BT it is given. A node already in `seen` makes no marginal contribution; it was already counted earlier.

Finally, we should not export such a function to the user, who has to deal with an unwieldy extra parameter and may send something poorly-formed, thereby causing our function to break. Instead, we should write a wrapper for it:

```
fun size-2(b :: BT): size-2-h(b, empty) end
```

This also enables us to use our old tests (renamed):

```
check:
    size-2(n1) is 1
    size-2(n2) is 2
    size-2(n3) is 2
    size-2(n4) is 4
end
```

Unfortunately, this still doesn't work!

Do Now!

Use Pyret's `spy` construct in `size-2-h` to figure out why.

15.2.3 Stage 3

Did you remember to use `spy`? Otherwise you may very well miss the problem! Be sure to use `spy` (feel free to elide the first few tests for now) to get a feel for the issue.

As you may have noted, the problem is that we want `seen` to be all the nodes ever seen. However, every time we return from one sub-computation, we also lose track of whatever was seen during its work. Instead, we have to also return everything that was seen, so as to properly preserve the idea that we're computing the marginal contribution of each node.

We can do this with the following data structure:

```
data Ret: ret(sz :: Number, sn :: List<BT>) end
```

which is returned by the helper function:

```
fun size-3-h(b :: BT, seen :: List<BT>) -> Ret:
    if member-identical(seen, b):
        ret(0, seen)
```

```

else:
    cases (BT) b:
        | mt => ret(0, seen)
        | nd(v, l, r) =>
            new-seen = link(b, seen)
            rl = size-3-h(l, new-seen)
            rr = size-3-h(r, rl.sn)
            ret(1 + rl.sz + rr.sz, rr.sn)
    end
end
end

```

Note, crucially, how the `seen` argument for the right branch is `rl.sn`: i.e., everything that was already seen in the left branch. This is the crucial step that avoids the bug.

Because of this richer return type, we have to extract the actual answer for the purpose of testing:

```

fun size-3(b :: BT): size-3-h(b, empty).sz end

check:
    size-3(n1) is 1
    size-3(n2) is 2
    size-3(n3) is 2
    size-3(n4) is 4
end

```

Exercise

Must `seen` be a list? What else can it be?

15.2.4 Stage 4 Observe that the `Ret` data structure is only of local interest. It's purely internal to the `size-3-h` function; even `size-3` ignores one half, and it will never be seen by the rest of the program. That is a good use of tuples, as we have seen before: Using Tuples!

```

fun size-4-h(b :: BT, seen :: List<BT>) -> {Number; List<BT>}:
    if member-identical(seen, b):
        {0; seen}
    else:
        cases (BT) b:
            | mt => {0; seen}
            | nd(v, l, r) =>
                new-seen = link(b, seen)
                {lsz; lsn} = size-4-h(l, new-seen)
                {rsz; rsn} = size-4-h(r, lsn)
                {1 + lsz + rsz; rsn}
        end
    end
end

```

```
fun size-4(b :: BT): size-4-h(b, empty).{0} end
```

```
check:
    size-4(n1) is 1
    size-4(n2) is 2
    size-4(n3) is 2
    size-4(n4) is 4
end
```

The notation `{0; seen}` makes an actual tuple; `{Number; List<BT>}` declares the contract of a tuple. Also, `.{0}` extracts the 0th element (the leftmost one) of a tuple.

15.2.5 Stage 5 Notice that we have the two instances of the code `{0; seen}`. Do they have to be that? What if we were to return `{0; empty}` instead in both places? Does anything break?

We might expect it to break in the case where `member-identical` returns `true`, but perhaps not in the `mt` case.

Do Now!

Make each of these changes. Does the outcome match your expectations?

Curiously, no! Making the change in the `mt` case has an effect but making it in the `member-identical` case doesn't! This almost seems counter-intuitive. How can we diagnose this?

Do Now!

Use `spy` to determine what is going on!

Okay, so it seems like returning `empty` when we revisit a node doesn't seem to do any harm. Does that mean it's okay to make that change?

Observe that nothing has actually depended on that `seen`-list being `empty`. That's why it appears to not matter. How can we make it matter? By making it "hurt" the computation by visiting a previously seen, but now forgotten, node yet again. So we need to visit a node at least three times: the first time to remember it; the second time to forget it; and a third time to incorrectly visit it again. Here's a DAG that will do that:

```
#|
  10
  / \
  11 12
  / \ /
13<--
|#

n13 = nd(13, mt, mt)
n11 = nd(11, n13, n13)
n12 = nd(12, n13, mt)
n10 = nd(10, n11, n12)

check:
  size-4(n10) is 4
end
```

Sure enough, if either tuple now returns `empty`, this test fails. Otherwise it succeeds.

15.2.6 What We've Learned We have learned three important principles here:

- A pattern for dealing with programs that need "memory". This is called threading (not in the sense of "multi-threading", which is a kind of parallel computation, but rather the pattern of how the `seen` list gets passed through the program).
- A good example of the use of tuples: local, where the documentation benefit of datatypes isn't necessary (and the extra datatype probably just clutters up the program), as opposed to distant, where it is. In general, it's always okay to make a new datatype; it's only sometimes okay to use tuples in their place.
- An important software-engineering principle, called mutation testing. This is an odd name because it would seem to be the name of a technique to test programs. Actually, it's a technique to test test suites. You have a tested program; you then "mutate" some part of your program that you feel must change the output, and see whether any tests break. If no tests break, then either you've misunderstood your program or, more likely, your test suite is not good enough. Improve your test suite to catch the error in your program, or convince yourself the change didn't matter.

There are mutation testing tools that will randomly try to alter your program using "mutant" strategies—e.g., replacing a `+` with a `--` and re-run your suites, and then report back on how many potential mutants the suites

actually caught. But we can't and shouldn't only rely on tools; we can also apply the principle of mutation testing by hand, as we have above. At the very least, it will help us understand our program better!

15.2.7 More on Value Printing: An Aside from Racket Earlier, we talked about how the standard recursive size can still be thought of as a “size of printed value” computation. However, that actually depends on your language’s value printer.

In Racket, you can turn on (it's slightly more expensive, so off by default) a value-printer that shows value sharing: Language | Choose Language ... | Show Details | Show sharing in values. So if we take the data definition above and translate it into Racket structures

```
(struct mt () #:transparent)
(struct nd (v l r) #:transparent)
```

and then construct (almost) the same data as in the first example:

```
(define n1 (nd 1 (mt) (mt)))
(define n2 (nd 2 (mt) n1))
(define n3 (nd 3 n1 (mt)))
(define n4 (nd 4 n2 n3))
```

and then ask Racket to print it, we get:

```
> n4
(nd 4 (nd 2 (mt) #0=(nd 1 (mt) (mt))) (nd 3 #0# (mt)))
```

The `#0=` notation is the moral equivalent of saying, “I’m going to refer to this value again later, so let’s call it the 0th value” and `#0#` is saying “Here I’m referring to the aforementioned 0th value”.

(Yes, there can be more than one shared value in an output, so each is given a different “name”. We’ll see that in a moment.)

The later example above translates to

```
(define n13 (nd 13 (mt) (mt)))
(define n11 (nd 11 n13 n13))
(define n12 (nd 12 n13 (mt)))
(define n10 (nd 10 n11 n12))
```

which prints as

```
> n10
(nd 10 (nd 11 #0=(nd 13 (mt) (mt)) #0#) (nd 12 #0# (mt)))
```

So it is possible for a language to reflect the sharing in its output. It’s just that most programming languages choose to not do that, even optionally.

Remember the “almost” above? What was that about?

In Racket, we’ve made a new instance of `mt` over and over. We can more accurately reflect what is happening in Pyret by instantiating it only once:

```
(struct mt () #:transparent)
(define the-mt (mt))
(struct nd (v l r) #:transparent)
```

We then rewrite the earlier example to use that one instance only:

```
(define n1 (nd 1 the-mt the-mt))
```

```
(define n2 (nd 2 the-mt n1))
(define n3 (nd 3 n1 the-mt))
(define n4 (nd 4 n2 n3))
```

And now when we print it:

```
> n4
```

```
(nd 4 (nd 2 #0=(mt) #1=(nd 1 #0# #0#)) (nd 3 #1# #0#))
```

And now you can see there are two different shared values, one is the single instance of mt, the other is the nd with 1 in it. Thus, Racket uses both `#0= / #0#` and `#1= / #1#`. Notice how all the leaves are sharing the same mt instance. (The numbering is picked in the order in which nodes are encountered while traversing, which is why the nd instance was `#0` the previous time and is `#1` this time.)

contents ← prev up next →

16.1 Introducing Graphs

16.1 Introducing Graphs In From Acyclicity to Cycles we introduced a special kind of sharing: when the data become cyclic, i.e., there exist values such that traversing other reachable values from them eventually gets you back to the value at which you began. Data that have this characteristic are called graphs. Technically, a cycle is not necessary to be a graph; a tree or a DAG is also regarded as a (degenerate) graph. In this section, however, we are interested in graphs that have the potential for cycles.

Lots of very important data are graphs. For instance, the people and connections in social media form a graph: the people are nodes or vertices and the connections (such as friendships) are links or edges. They form a graph because for many people, if you follow their friends and then the friends of their friends, you will eventually get back to the person you started with. (Most simply, this happens when two people are each others' friends.) The Web, similarly is a graph: the nodes are pages and the edges are links between pages. The Internet is a graph: the nodes are machines and the edges are links between machines. A transportation network is a graph: e.g., cities are nodes and the edges are transportation links between them. And so on. Therefore, it is essential to understand graphs to represent and process a great deal of interesting real-world data.

Graphs are important and interesting for not only practical but also principled reasons. The property that a traversal can end up where it began means that traditional methods of processing will no longer work: if it blindly processes every node it visits, it could end up in an infinite loop. Therefore, we need better structural recipes for our programs. In addition, graphs have a very rich structure, which lends itself to several interesting computations over them. We will study both these aspects of graphs below.

16.1.1 Understanding Graphs Consider again the binary trees we saw earlier [Re-Examining Equality]. Let's now try to distort the definition of a "tree" by creating ones with cycles, i.e., trees with nodes that point back to themselves (in the sense of `identical`). As we saw earlier [From Acyclicity to Cycles], it is not completely straightforward to create such a structure, but what we saw earlier [Streams From Functions] can help us here, by letting us suspend the evaluation of the cyclic link. That is, we have to not only use `rec`, we must also use a function to delay evaluation. In turn, we have to update the annotations on the fields. Since these are not going to be "trees" any more, we'll use a name that is suggestive but not outright incorrect:

```
data BinT:  
| leaf  
| node(v, l :: ( -> BinT), r :: ( -> BinT))  
end
```

Now let's try to construct some cyclic values. Here are a few examples:

```
rec tr = node("rec", lam(): tr end, lam(): tr end)  
t0 = node(0, lam(): leaf end, lam(): leaf end)  
t1 = node(1, lam(): t0 end, lam(): t0 end)  
t2 = node(2, lam(): t1 end, lam(): t1 end)
```

Now let's try to compute the size of a `BinT`. Here's the obvious program:

```
fun sizeinf(t :: BinT) -> Number:  
cases (BinT) t:  
| leaf => 0  
| node(v, l, r) =>  
  ls = sizeinf(l())  
  rs = sizeinf(r())  
  1 + ls + rs
```

```

    end
end

```

(We'll see why we call it `sizeinf` in a moment.)

Do Now!

What happens when we call `sizeinf(tr)`?

It goes into an infinite loop: hence the `inf` in its name.

There are two very different meanings for “size”. One is, “How many times can we traverse an edge?” The other is, “How many distinct nodes were constructed as part of the data structure?” With trees, by definition, these two are the same. With a DAG the former exceeds the latter but only by a finite amount. With a general graph, the former can exceed the latter by an infinite amount. In the case of a datum like `tr`, we can in fact traverse edges an infinite number of times. But the total number of constructed nodes is only one! Let's write this as test cases in terms of a `size` function, to be defined:

```

check:
  size(tr) is 1
  size(t0) is 1
  size(t1) is 2
  size(t2) is 3
end

```

It's clear that we need to somehow remember what nodes we have visited previously: that is, we need a computation with “memory”. In principle this is easy: we just create an extra data structure that checks whether a node has already been counted. As long as we update this data structure correctly, we should be all set. Here's an implementation.

```

fun sizect(t :: BinT) -> Number:
  fun szacc(shadow t :: BinT, seen :: List<BinT>) -> Number:
    if has-id(seen, t):
      0
    else:
      cases (BinT) t:
        | leaf => 0
        | node(v, l, r) =>
          ns = link(t, seen)
          ls = szacc(l(), ns)
          rs = szacc(r(), ns)
          1 + ls + rs
    end
  end
end
szacc(t, empty)
end

```

The extra parameter, `seen`, is called an accumulator, because it “accumulates” the list of seen nodes. Note that this could just as well be a set; it doesn't have to be a list. The support function it needs checks whether a given node has already been seen:

```

fun has-id<A>(seen :: List<A>, t :: A):
  cases (List) seen:
    | empty => false
    | link(f, r) =>
      if f <=> t: true
      else: has-id(r, t)
    end
  end
end

```

How does this do? Well, `sizect(tr)` is indeed 1, but `sizect(t1)` is 3 and `sizect(t2)` is 7!

Do Now!

Explain why these answers came out as they did.

The fundamental problem is that we're not doing a very good job of remembering! Look at this pair of lines:

```
ls = szacc(l(), ns)
rs = szacc(r(), ns)
```

The nodes seen while traversing the left branch are effectively forgotten, because the only nodes we remember when traversing the right branch are those in `ns`: namely, the current node and those visited “higher up”. As a result, any nodes that “cross sides” are counted twice.

The remedy for this, therefore, is to remember every node we visit. Then, when we have no more nodes to process, instead of returning only the size, we should return all the nodes visited until now. This ensures that nodes that have multiple paths to them are visited on only one path, not more than once. The logic for this is to return two values from each traversal—the size and all the visited nodes—and not just one.

```
fun size(t :: BinT) -> Number:
    fun szacc(shadow t :: BinT, seen :: List<BinT>)
        -> {n :: Number, s :: List<BinT>}:
        if has-id(seen, t):
            {n: 0, s: seen}
        else:
            cases (BinT) t:
                | leaf => {n: 0, s: seen}
                | node(v, l, r) =>
                    ns = link(t, seen)
                    ls = szacc(l(), ns)
                    rs = szacc(r(), ls.s)
                    {n: 1 + ls.n + rs.n, s: rs.s}
            end
        end
    end
    szacc(t, empty).n
end
```

Sure enough, this function satisfies the above tests.

16.1.2 Representations The representation we've seen above for graphs is certainly a start towards creating cyclic data, but it's not very elegant. It's both error-prone and inelegant to have to write `lam` everywhere, and remember to apply functions to `()` to obtain the actual values. Therefore, here we explore other representations of graphs that are more conventional and also much simpler to manipulate.

There are numerous ways to represent graphs, and the choice of representation depends on several factors:

1. The structure of the graph, and in particular, its density. We will discuss this further later [Measuring Complexity for Graphs].
2. The representation in which the data are provided by external sources. Sometimes it may be easier to simply adapt to their representation; in particular, in some cases there may not even be a choice.
3. The features provided by the programming language, which make some representations much harder to use than others.

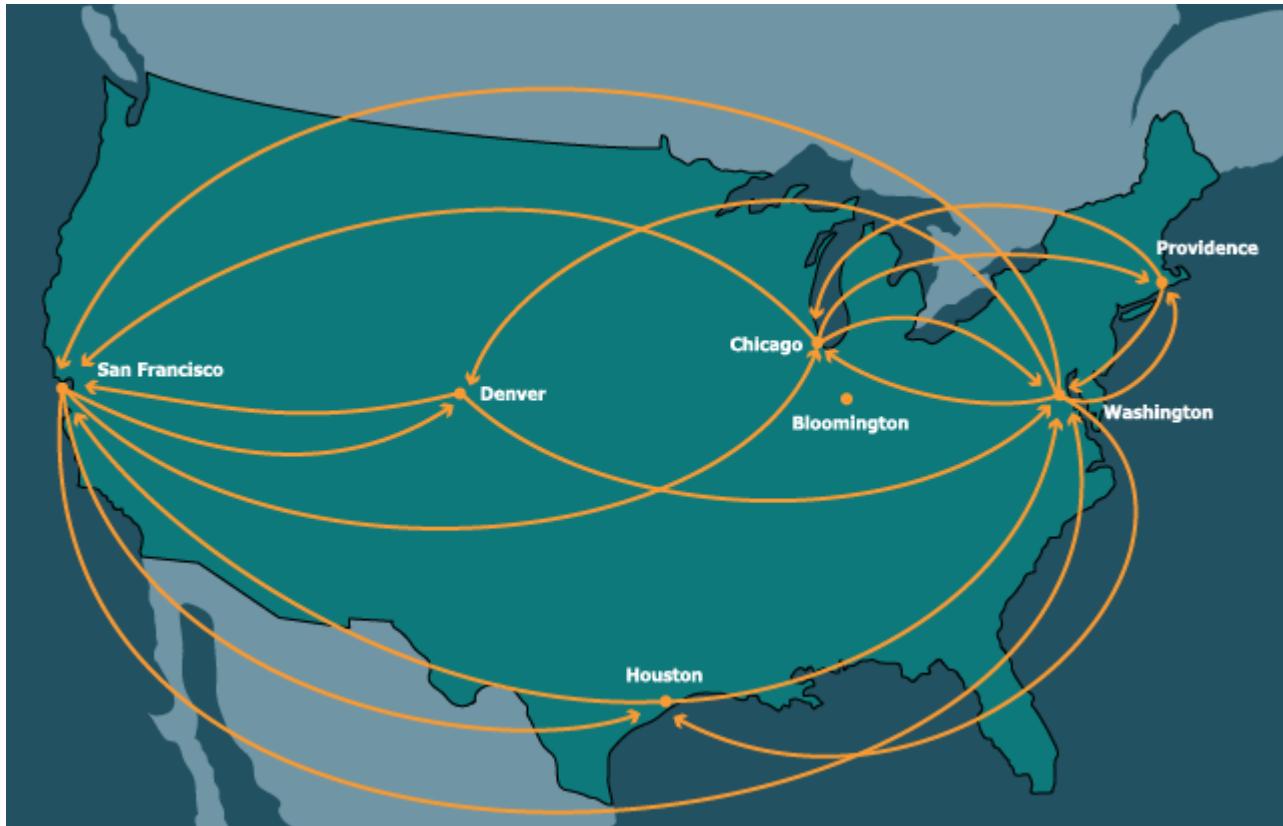
In [Several Variations on Sets], we explore the idea of having many different representations for one datatype. As we will see, this is very true of graphs as well. Therefore, it would be best if we could arrive at a common interface to process graphs, so that all later programs can be written in terms of this interface, without overly depending on the underlying representation.

In terms of representations, there are three main things we need:

1. A way to construct graphs.
2. A way to identify (i.e., tell apart) nodes or vertices in a graph.
3. Given a way to identify nodes, a way to get that node's neighbors in the graph.

Any interface that satisfies these properties will suffice. For simplicity, we will focus on the second and third of these and not abstract over the process of constructing a graph.

Our running example will be a graph whose nodes are cities in the United States and edges are direct flight connections between them:



16.1.2.1 Links by Name Here's our first representation. We will assume that every node has a unique name (such a name, when used to look up information in a repository of data, is sometimes called a key). A node is then a key, some information about that node, and a list of keys that refer to other nodes:

```
type Key = String

data KeyedNode:
  | keyed-node(key :: Key, content, adj :: List<String>)
end

type KNGraph = List<KeyedNode>

type Node = KeyedNode
type Graph = KNGraph
```

(Here we're assuming our keys are strings.)

Here's a concrete instance of such a graph: The prefix `kn-` stands for “keyed node”.

```
kn-cities :: Graph = block:
  knWAS = keyed-node("was", "Washington", [list: "chi", "den", "saf", "hou", "pwd"])
  knORD = keyed-node("chi", "Chicago", [list: "was", "saf", "pwd"])
```

```

knBLM = keyed-node("bmg", "Bloomington", [list: ])
knHOU = keyed-node("hou", "Houston", [list: "was", "saf"])
knDEN = keyed-node("den", "Denver", [list: "was", "saf"])
knSFO = keyed-node("saf", "San Francisco", [list: "was", "den", "chi", "hou"])
knPVD = keyed-node("pvd", "Providence", [list: "was", "chi"])
[list: knWAS, knORD, knBLM, knHOU, knDEN, knSFO, knPVD]
end

```

Given a key, here's how we look up its neighbor:

```

fun find-kn(key :: Key, graph :: Graph) -> Node:
    matches = for filter(n from graph):
        n.key == key
    end
    matches.first # there had better be exactly one!
end

```

Exercise

Convert the comment in the function into an invariant about the datum. Express this invariant as a refinement and add it to the declaration of graphs.

With this support, we can look up neighbors easily:

```

fun kn-neighbors(city :: Key, graph :: Graph) -> List<Key>:
    city-node = find-kn(city, graph)
    city-node.adj
end

```

When it comes to testing, some tests are easy to write. Others, however, might require describing entire nodes, which can be unwieldy, so for the purpose of checking our implementation it suffices to examine just a part of the result:

```

check:
    ns = kn-neighbors("hou", kn-cities)

    ns is [list: "was", "saf"]

    map(_.content, map(find-kn(_, kn-cities), ns)) is
        [list: "Washington", "San Francisco"]
end

```

16.1.2.2 Links by Indices In some languages, it is common to use numbers as names. This is especially useful when numbers can be used to get access to an element in a constant amount of time (in return for having a bound on the number of elements that can be accessed). Here, we use a list—which does not provide constant-time access to arbitrary elements—to illustrate this concept. Most of this will look very similar to what we had before; we'll comment on a key difference at the end.

First, the datatype: The prefix `ix-` stands for “indexed”.

```

data IndexedNode:
    | idxed-node(content, adj :: List<Number>)
end

```

```

type IXGraph = List<IndexedNode>

```

```

type Node = IndexedNode
type Graph = IXGraph

```

Our graph now looks like this:

```

ix-cities :: Graph = block:
    inWAS = idxed-node("Washington", [list: 1, 4, 5, 3, 6])

```

```

inORD = idxed-node("Chicago", [list: 0, 5, 6])
inBLM = idxed-node("Bloomington", [list: ])
inHOU = idxed-node("Houston", [list: 0, 5])
inDEN = idxed-node("Denver", [list: 0, 5])
inSFO = idxed-node("San Francisco", [list: 0, 4, 3])
inPVD = idxed-node("Providence", [list: 0, 1])
[list: inWAS, inORD, inBLM, inHOU, inDEN, inSFO, inPVD]
end

```

where we're assuming indices begin at 0. To find a node:

```

fun find-ix(idx :: Key, graph :: Graph) -> Node:
    lists.get(graph, idx)
end

```

We can then find neighbors almost as before:

```

fun ix-neighbors(city :: Key, graph :: Graph) -> List<Key>:
    city-node = find-ix(city, graph)
    city-node.adj
end

```

Finally, our tests also look similar:

```

check:
    ns = ix-neighbors(3, ix-cities)

    ns is [list: 0, 5]

    map(_.content, map(find-ix(_, ix-cities), ns)) is
        [list: "Washington", "San Francisco"]
end

```

Something deeper is going on here. The keyed nodes have intrinsic keys: the key is part of the datum itself. Thus, given just a node, we can determine its key. In contrast, the indexed nodes represent extrinsic keys: the keys are determined outside the datum, and in particular by the position in some other data structure. Given a node and not the entire graph, we cannot know for what its key is. Even given the entire graph, we can only determine its key by using `identical`, which is a rather unsatisfactory approach to recovering fundamental information. This highlights a weakness of using extrinsically keyed representations of information. (In return, extrinsically keyed representations are easier to reassemble into new collections of data, because there is no danger of keys clashing: there are no intrinsic keys to clash.)

16.1.2.3 A List of Edges The representations we have seen until now have given priority to nodes, making edges simply a part of the information in a node. We could, instead, use a representation that makes edges primary, and nodes simply be the entities that lie at their ends: The prefix `le-` stands for “list of edges”.

```

data Edge:
    | edge(src :: String, dst :: String)
end

type LEGraph = List<Edge>

type Graph = LEGraph

```

Then, our flight network becomes:

```

le-cities :: Graph =
[list:
    edge("Washington", "Chicago"),
    edge("Washington", "Denver"),
    edge("Washington", "San Francisco"),

```

```

edge("Washington", "Houston"),
edge("Washington", "Providence"),
edge("Chicago", "Washington"),
edge("Chicago", "San Francisco"),
edge("Chicago", "Providence"),
edge("Houston", "Washington"),
edge("Houston", "San Francisco"),
edge("Denver", "Washington"),
edge("Denver", "San Francisco"),
edge("San Francisco", "Washington"),
edge("San Francisco", "Denver"),
edge("San Francisco", "Houston"),
edge("Providence", "Washington"),
edge("Providence", "Chicago") ]

```

Observe that in this representation, nodes that are not connected to other nodes in the graph simply never show up! You'd therefore need an auxilliary data structure to keep track of all the nodes.

To obtain the set of neighbors:

```

fun le-neighbors(city :: Key, graph :: Graph) -> List<Key>:
    neighboring-edges = for filter(e from graph):
        city == e.src
    end
    names = for map(e from neighboring-edges): e.dst end
    names
end

```

And to be sure:

```

check:
    le-neighbors("Houston", le-cities) is
        [list: "Washington", "San Francisco"]
end

```

However, this representation makes it difficult to store complex information about a node without replicating it. Because nodes usually have rich information while the information about edges tends to be weaker, we often prefer node-centric representations. Of course, an alternative is to think of the node names as keys into some other data structure from which we can retrieve rich information about nodes.

16.1.2.4 Abstracting Representations We would like a general representation that lets us abstract over the specific implementations. We will assume that broadly we have available a notion of `Node` that has `content`, a notion of `Keys` (whether or not intrinsic), and a way to obtain the neighbors—a list of keys—given a key and a graph. This is sufficient for what follows. However, we still need to choose concrete keys to write examples and tests. For simplicity, we'll use string keys [Links by Name].

16.1.3 Measuring Complexity for Graphs Before we begin to define algorithms over graphs, we should consider how to measure the size of a graph. A graph has two components: its nodes and its edges. Some algorithms are going to focus on nodes (e.g., visiting each of them), while others will focus on edges, and some will care about both. So which do we use as the basis for counting operations: nodes or edges?

It would help if we can reduce these two measures to one. To see whether that's possible, suppose a graph has $\backslash(k\backslash)$ nodes. Then its number of edges has a wide range, with these two extremes:

- No two nodes are connected. Then there are no edges at all.
- Every two nodes is connected. Then there are essentially as many edges as the number of pairs of nodes.

The number of nodes can thus be significantly less or even significantly more than the number of edges. Were this difference a matter of constants, we could have ignored it; but it's not. As a graph tends towards the former extreme, the ratio of nodes to edges approaches $\backslash(k\backslash)$ (or even exceeds it, in the odd case where there are no edges, but this

graph is not very interesting); as it tends towards the latter, it is the ratio of edges to nodes that approaches $\backslash(k^2\backslash)$. In other words, neither measure subsumes the other by a constant independent of the graph.

Therefore, when we want to speak of the complexity of algorithms over graphs, we have to consider the sizes of both the number of nodes and edges. In a connected graphA graph is connected if, from every node, we can traverse edges to get to every other node., however, there must be at least as many edges as nodes, which means the number of edges dominates the number of nodes. Since we are usually processing connected graphs, or connected parts of graphs one at a time, we can bound the number of nodes by the number of edges.

contents ← prev up next →

16.2 Basic Graph Traversals

16.2 Basic Graph Traversals As with all the data we have seen so far, to process a datum we have to traverse it—i.e., visit the constituent data. With graphs, that can be quite interesting!

16.2.1 Reachability Many uses of graphs need to address reachability: whether we can, using edges in the graph, get from one node to another. For instance, a social network might suggest as contacts all those who are reachable from existing contacts. On the Internet, traffic engineers care about whether packets can get from one machine to another. On the Web, we care about whether all public pages on a site are reachable from the home page. We will study how to compute reachability using our travel graph as a running example.

16.2.1.1 Simple Recursion At its simplest, reachability is easy. We want to know whether there exists a pathA path is a sequence of zero or more linked edges. between a pair of nodes, a source and a destination. (A more sophisticated version of reachability might compute the actual path, but we'll ignore this for now.) There are two possibilities: the source and destination nodes are the same, or they're not.

- If they are the same, then clearly reachability is trivially satisfied.
- If they are not, we have to iterate through the neighbors of the source node and ask whether the destination is reachable from each of those neighbors.

This translates into the following function:

```
<graph-reach-1-main> ::=
fun reach-1(src :: Key, dst :: Key, g :: Graph) -> Boolean:
  if src == dst:
    true
  else:
    <graph-reach-1-loop>
    loop(neighbors(src, g))
  end
end
```

where the loop through the neighbors of `src` is:

```
<graph-reach-1-loop> ::=
fun loop(ns):
  cases (List) ns:
    | empty => false
    | link(f, r) =>
      if reach-1(f, dst, g): true else: loop(r) end
  end
end
```

We can test this as follows:

```
<graph-reach-tests> ::=
check:
  reach = reach-1
  reach("was", "was", kn-cities) is true
  reach("was", "chi", kn-cities) is true
```

```

reach("was", "bmg", kn-cities) is false
reach("was", "hou", kn-cities) is true
reach("was", "den", kn-cities) is true
reach("was", "saf", kn-cities) is true
end

```

Unfortunately, we don't find out about how these tests fare, because some of them don't complete at all. That's because we have an infinite loop, due to the cyclic nature of graphs!

Exercise

Which of the above examples leads to a cycle? Why?

16.2.1.2 Cleaning up the Loop Before we continue, let's try to improve the expression of the loop. While the nested function above is a perfectly reasonable definition, we can use Pyret's `for` to improve its readability.

The essence of the above loop is to iterate over a list of boolean values; if one of them is true, the entire loop evaluates to true; if they are all false, then we haven't found a path to the destination node, so the loop evaluates to false. Thus:

```

fun ormap(fun-body, l):
  cases (List) l:
    | empty => false
    | link(f, r) =>
      if fun-body(f): true else: ormap(fun-body, r) end
  end
end

```

With this, we can replace the loop definition and use with:

```

for ormap(n from neighbors(src, g)):
  reach-1(n, dst, g)
end

```

16.2.1.3 Traversal with Memory Because we have cyclic data, we have to remember what nodes we've already visited and avoid traversing them again. Then, every time we begin traversing a new node, we add it to the set of nodes we've already started to visit so that. If we return to that node, because we can assume the graph has not changed in the meanwhile, we know that additional traversals from that node won't make any difference to the outcome. This property is known as `idempotence`.

We therefore define a second attempt at reachability that take an extra argument: the set of nodes we have begun visiting (where the set is represented as a graph). The key difference from <graph-reach-1-main> is, before we begin to traverse edges, we should check whether we've begun processing the node or not. This results in the following definition:

```

<graph-reach-2> ::=
fun reach-2(src :: Key, dst :: Key, g :: Graph, visited :: List<Key>) -> Boolean:
  if visited.member(src):
    false
  else if src == dst:
    true
  else:
    new-visited = link(src, visited)
    for ormap(n from neighbors(src, g)):
      reach-2(n, dst, g, new-visited)
    end
  end
end

```

In particular, note the extra new conditional: if the reachability check has already visited this node before, there is no point traversing further from here, so it returns `false`. (There may still be other parts of the graph to explore, which other recursive calls will do.)

Exercise

Does it matter if the first two conditions were swapped, i.e., the beginning of `reach-2` began with

```
if src == dst:  
    true  
else if visited.member(src):  
    false
```

? Explain concretely with examples.

Exercise

We repeatedly talk about remembering the nodes that we have begun to visit, not the ones we've finished visiting. Does this distinction matter? How?

16.2.1.4 A Better Interface As the process of testing `reach-2` shows, we may have a better implementation, but we've changed the function's interface; now it has a needless extra argument, which is not only a nuisance but might also result in errors if we accidentally misuse it. Therefore, we should clean up our definition by moving the core code to an internal function:

```
fun reach-3(s :: Key, d :: Key, g :: Graph) -> Boolean:  
    fun reacher(src :: Key, dst :: Key, visited :: List<Key>) -> Boolean:  
        if visited.member(src):  
            false  
        else if src == dst:  
            true  
        else:  
            new-visited = link(src, visited)  
            for ormap(n from neighbors(src, g)):  
                reacher(n, dst, new-visited)  
            end  
        end  
    end  
    reacher(s, d, empty)  
end
```

We have now restored the original interface while correctly implementing reachability.

Exercise

Does this really gives us a correct implementation? In particular, does this address the problem that the `size` function above addressed? Create a test case that demonstrates the problem, and then fix it.

16.2.2 Depth- and Breadth-First Traversals It is conventional for computer science texts to call these depth- and breadth-first search. However, searching is just a specific purpose; traversal is a general task that can be used for many purposes.

The reachability algorithm we have seen above has a special property. At every node it visits, there is usually a set of adjacent nodes at which it can continue the traversal. It has at least two choices: it can either visit each immediate neighbor first, then visit all of the neighbors' neighbors; or it can choose a neighbor, recur, and visit the next immediate neighbor only after that visit is done. The former is known as breadth-first traversal, while the latter is depth-first traversal.

The algorithm we have designed uses a depth-first strategy: inside `<graph-reach-1-loop>`, we recur on the first element of the list of neighbors before we visit the second neighbor, and so on. The alternative would be to have a

data structure into which we insert all the neighbors, then pull out an element at a time such that we first visit all the neighbors before their neighbors, and so on. This naturally corresponds to a queue [An Example: Queues from Lists].

Exercise

Using a queue, implement breadth-first traversal.

If we correctly check to ensure we don't re-visit nodes, then both breadth- and depth-first traversal will properly visit the entire reachable graph without repetition (and hence not get into an infinite loop). Each one traverses from a node only once, from which it considers every single edge. Thus, if a graph has $\lvert N \rvert$ nodes and $\lvert E \rvert$ edges, then a lower-bound on the complexity of traversal is $\lvert O([N, E \rightarrow N + E]) \rvert$. We must also consider the cost of checking whether we have already visited a node before (which is a set membership problem, which we address elsewhere: Several Variations on Sets). Finally, we have to consider the cost of maintaining the data structure that keeps track of our traversal. In the case of depth-first traversal, recursion—which uses the machine's stack—does it automatically at constant overhead. In the case of breadth-first traversal, the program must manage the queue, which can add more than constant overhead. In practice, too, the stack will usually perform much better than a queue, because it is supported by machine hardware.

This would suggest that depth-first traversal is always better than breadth-first traversal. However, breadth-first traversal has one very important and valuable property. Starting from a node $\lvert N \rvert$, when it visits a node $\lvert P \rvert$, count the number of edges taken to get to $\lvert P \rvert$. Breadth-first traversal guarantees that there cannot have been a shorter path to $\lvert P \rvert$: that is, it finds a shortest path to $\lvert P \rvert$.

Exercise

Why “a” rather than “the” shortest path?

Exercise

Prove that breadth-first traversal finds a shortest path.

contents ← prev up next →

16.3 Graphs With Weighted Edges

16.3 Graphs With Weighted Edges Consider a transportation graph: we are usually interested not only in whether we can get from one place to another, but also in what it “costs” (where we may have many different cost measures: money, distance, time, units of carbon dioxide, etc.). On the Internet, we might care about the latency or bandwidth over a link. Even in a social network, we might like to describe the degree of closeness of a friend. In short, in many graphs we are interested not only in the direction of an edge but also in some abstract numeric measure, which we call its weight.

In the rest of this study, we will assume that our graph edges have weights. This does not invalidate what we’ve studied so far: if a node is reachable in an unweighted graph, it remains reachable in a weighted one. But the operations we are going to study below only make sense in a weighted graph. We can, however, always treat an unweighted graph as a weighted one by giving every edge the same, constant, positive weight (say one).

Exercise

When treating an unweighted graph as a weighted one, why do we care that every edge be given a positive weight?

Exercise

Revise the graph data definitions to account for edge weights.

Exercise

Weights are not the only kind of data we might record about edges. For instance, if the nodes in a graph represent people, the edges might be labeled with their relationship (“mother”, “friend”, etc.). What other kinds of data can you imagine recording for edges?

contents ← prev up next →

16.4 Shortest (or Lightest) Paths

16.4 Shortest (or Lightest) Paths Imagine planning a trip: it's natural that you might want to get to your destination in the least time, or for the least money, or some other criterion that involves minimizing the sum of edge weights. This is known as computing the shortest path.

We should immediately clarify an unfortunate terminological confusion. What we really want to compute is the lightest path—the one of least weight. Unfortunately, computer science terminology has settled on the terminology we use here; just be sure to not take it literally.

Exercise

Construct a graph and select a pair of nodes in it such that the shortest path from one to the other is not the lightest one, and vice versa.

We have already seen [Depth- and Breadth-First Traversals] that breadth-first search constructs shortest paths in unweighted graphs. These correspond to lightest paths when there are no weights (or, equivalently, all weights are identical and positive). Now we have to generalize this to the case where the edges have weights.

We will proceed inductively, gradually defining a function seemingly of this type

```
w :: Key -> Number
```

that reflects the weight of the lightest path from the source node to that one. But let's think about this annotation: since we're building this up node-by-node, initially most nodes have no weight to report; and even at the end, a node that is unreachable from the source will have no weight for a lightest (or indeed, any) path. Rather than make up a number that pretends to reflect this situation, we will instead use an option type:

```
w :: Key -> Option<Number>
```

When there is `some` value it will be the weight; otherwise the weight will be `none`.

Now let's think about this inductively. What do we know initially? Well, certainly that the source node is at a distance of zero from itself (that must be the lightest path, because we can't get any lighter). This gives us a (trivial) set of nodes for which we already know the lightest weight. Our goal is to grow this set of nodes—modestly, by one, on each iteration—until we either find the destination, or we have no more nodes to add (in which case our destination is not reachable from the source).

Inductively, at each step we have the set of all nodes for which we know the lightest path (initially this is just the source node, but it does mean this set is never empty, which will matter in what we say next). Now consider all the edges adjacent to this set of nodes that lead to nodes for which we don't already know the lightest path. Choose a node, $\backslash(q\backslash)$, that minimizes the total weight of the path to it. We claim that this will in fact be the lightest path to that node.

If this claim is true, then we are done. That's because we would now add $\backslash(q\backslash)$ to the set of nodes whose lightest weights we now know, and repeat the process of finding lightest outgoing edges from there. This process has thus added one more node. At some point we will find that there are no edges that lead outside the known set, at which point we can terminate.

It stands to reason that terminating at this point is safe: it corresponds to having computed the reachable set. The only thing left is to demonstrate that this greedy algorithm yields a lightest path to each node.

We will prove this by contradiction. Suppose we have the path $\backslash(s \rightarrow d\backslash)$ from source $\backslash(s\backslash)$ to node $\backslash(d\backslash)$, as found by the algorithm above, but assume also that we have a different path that is actually lighter. At every node, when we added a node along the $\backslash(s \rightarrow d\backslash)$ path, the algorithm would have added a lighter path if it

existed. The fact that it did not falsifies our claim that a lighter path exists (there could be a different path of the same weight; this would be permitted by the algorithm, but it also doesn't contradict our claim). Therefore the algorithm does indeed find the lightest path.

What remains is to determine a data structure that enables this algorithm. At every node, we want to know the least weight from the set of nodes for which we know the least weight to all their neighbors. We could achieve this by sorting, but this is overkill: we don't actually need a total ordering on all these weights, only the lightest one. A heap see Wikipedia gives us this.

Exercise

What if we allowed edges of weight zero? What would change in the above algorithm?

Exercise

What if we allowed edges of negative weight? What would change in the above algorithm?

For your reference, this algorithm is known as Dijkstra's Algorithm.

contents ← prev up next →

16.5 Moravian Spanning Trees

16.5 Moravian Spanning Trees At the turn of the millennium, the US National Academy of Engineering surveyed its members to determine the “Greatest Engineering Achievements of the 20th Century”. The list contained the usual suspects: electronics, computers, the Internet, and so on. But a perhaps surprising idea topped the list: (rural) electrification. Read more about it on their site.

16.5.1 The Problem To understand the history of national electrical grids, it helps to go back to Moravia in the 1920s. Like many parts of the world, it was beginning to realize the benefits of electricity and intended to spread it around the region. A Moravian academic named Otakar Borůvka heard about the problem, and in a remarkable effort, described the problem abstractly, so that it could be understood without reference to Moravia or electrical networks. He modeled it as a problem about graphs.

Borůvka observed that at least initially, any solution to the problem of creating a network must have the following characteristics:

- The electrical network must reach all the towns intended to be covered by it. In graph terms, the solution must be spanning, meaning it must visit every node in the graph.
- Redundancy is a valuable property in any network: that way, if one set of links goes down, there might be another way to get a payload to its destination. When starting out, however, redundancy may be too expensive, especially if it comes at the cost of not giving someone a payload at all. Thus, the initial solution was best set up without loops or even redundant paths. In graph terms, the solution had to be a tree.
- Finally, the goal was to solve this problem for the least cost possible. In graph terms, the graph would be weighted, and the solution had to be a minimum.

Thus Borůvka defined the Moravian Spanning Tree (MST) problem.

16.5.2 A Greedy Solution Borůvka had published his problem, and another Czech mathematician, Vojtěch Jarník, came across it. Jarník came up with a solution that should sound familiar:

- Begin with a solution consisting of a single node, chosen arbitrarily. For the graph consisting of this one node, this solution is clearly a minimum, spanning, and a tree.
- Of all the edges incident on nodes in the solution that connect to a node not already in the solution, pick the edge with the least weight. Note that we consider only the incident edges, not their weight added to the weight of the node to which they are incident.
- Add this edge to the solution. The claim is that for the new solution will be a tree (by construction), spanning (also by construction), and a minimum. The minimality follows by an argument similar to that used for Dijkstra’s Algorithm.

Jarník had the misfortune of publishing this work in Czech in 1930, and it went largely ignored. It was rediscovered by others, most notably by R.C. Prim in 1957, and is now generally known as Prim’s Algorithm, though calling it Jarník’s Algorithm would attribute credit in the right place.

Implementing this algorithm is pretty easy. At each point, we need to know the lightest edge incident on the current solution tree. Finding the lightest edge takes time linear in the number of these edges, but the very lightest one may create a cycle. We therefore need to efficiently check for whether adding an edge would create a cycle, a problem we will return to multiple times [Checking Component Connectedness]. Assuming we can do that effectively, we then want to add the lightest edge and iterate. Even given an efficient solution for checking cyclicity, this would seem to

require an operation linear in the number of edges for each node. With better representations we can improve on this complexity, but let's look at other ideas first.

16.5.3 Another Greedy Solution Recall that Jarník presented his algorithm in 1930, when computers didn't exist, and Prim his in 1957, when they were very much in their infancy. Programming computers to track heaps was a non-trivial problem, and many algorithms were implemented by hand, where keeping track of a complex data structure without making errors was harder still. There was need for a solution that was required less manual bookkeeping (literally speaking).

In 1956, Joseph Kruskal presented such a solution. His idea was elegantly simple. The Jarník algorithm suffers from the problem that each time the tree grows, we have to revise the content of the heap, which is already a messy structure to track. Kruskal noted the following.

To obtain a minimum solution, surely we want to include one of the edges of least weight in the graph. Because if not, we can take an otherwise minimal solution, add this edge, and remove one other edge; the graph would still be just as connected, but the overall weight would be no more and, if the removed edge were heavier, would be less. Note the careful wording: there may be many edges of the same least weight, so adding one of them may remove another, and therefore not produce a lighter tree; but the key point is that it certainly will not produce a heavier one. By the same argument we can add the next lightest edge, and the next lightest, and so on. The only time we cannot add the next lightest edge is when it would create a cycle (that problem again!).

Therefore, Kruskal's algorithm is utterly straightforward. We first sort all the edges, ordered by ascending weight. We then take each edge in ascending weight order and add it to the solution provided it will not create a cycle. When we have thus processed all the edges, we will have a solution that is a tree (by construction), spanning (because every connected vertex must be the endpoint of some edge), and of minimum weight (by the argument above). The complexity is that of sorting (which is $\mathcal{O}(|e| \log |e|)$) where $|e|$ is the size of the edge set. We then iterate over each element in $|e|$, which takes time linear in the size of that set—modulo the time to check for cycles. This algorithm is also easy to implement on paper, because we sort all the edges once, then keep checking them off in order, crossing out the ones that create cycles—with no dynamic updating of the list needed.

16.5.4 A Third Solution Both the Jarník and Kruskal solutions have one flaw: they require a centralized data structure (the priority heap, or the sorted list) to incrementally build the solution. As parallel computers became available, and graph problems grew large, computer scientists looked for solutions that could be implemented more efficiently in parallel—which typically meant avoiding any centralized points of synchronization, such as these centralized data structures.

In 1965, M. Sollin constructed an algorithm that met these needs beautifully. In this algorithm, instead of constructing a single solution, we grow multiple solution components (potentially in parallel if we so wish). Each node starts out as a solution component (as it was at the first step of Jarník's Algorithm). Each node considers the edges incident to it, and picks the lightest one that connects to a different component (that problem again!). If such an edge can be found, the edge becomes part of the solution, and the two components combine to become a single component. The entire process repeats.

Because every node begins as part of the solution, this algorithm naturally spans. Because it checks for cycles and avoids them, it naturally forms a tree. Note that avoiding cycles yields a DAG and is not automatically guaranteed to yield a tree. We have been a bit lax about this difference throughout this section. Finally, minimality follows through similar reasoning as we used in the case of Jarník's Algorithm, which we have essentially run in parallel, once from each node, until the parallel solution components join up to produce a global solution.

Of course, maintaining the data for this algorithm by hand is a nightmare. Therefore, it would be no surprise that this algorithm was coined in the digital age. The real surprise, therefore, is that it was not: it was originally created by Otakar Borůvka himself.

Borůvka, you see, had figured it all out. He'd not only understood the problem, he had:

- pinpointed the real problem lying underneath the electrification problem so it could be viewed in a context-independent way,
- created a descriptive language of graph theory to define it precisely, and
- even solved the problem in addition to defining it.

He'd just come up with a solution so complex to implement by hand that Jarník had in essence de-parallelized it so it could be done sequentially. And thus this algorithm lay unnoticed until it was reinvented (several times, actually) by Sollin in time for parallel computing folks to notice a need for it. But now we can just call this Borůvka's Algorithm, which is only fitting.

As you might have guessed by now, this problem is indeed called the MST in other textbooks, but "M" stands not for Moravia but for "Minimum". But given Borůvka's forgotten place in history, we prefer the more whimsical name.

16.5.5 Checking Component Connectedness As we've seen, we need to be able to efficiently tell whether two nodes are in the same component. One way to do this is to conduct a depth-first traversal (or breadth-first traversal) starting from the first node and checking whether we ever visit the second one. (Using one of these traversal strategies ensures that we terminate in the presence of loops.) Unfortunately, this takes a linear amount of time (in the size of the graph) for every pair of nodes—and depending on the graph and choice of node, we might do this for every node in the graph on every edge addition! So we'd clearly like to do this better.

It is helpful to reduce this problem from graph connectivity to a more general one: of disjoint-set structure (colloquially known as union-find for reasons that will soon be clear). If we think of each connected component as a set, then we're asking whether two nodes are in the same set. But casting it as a set membership problem makes it applicable in several other applications as well.

The setup is as follows. For arbitrary values, we want the ability to think of them as elements in a set. We are interested in two operations. One is obviously `union`, which merges two sets into one. The other would seem to be something like `is-in-same-set` that takes two elements and determines whether they're in the same set. Over time, however, it has proven useful to instead define the operator `find` that, given an element, "names" the set (more on this in a moment) that the element belongs to. To check whether two elements are in the same set, we then have to get the "set name" for each element, and check whether these names are the same. This certainly sounds more roundabout, but this means we have a primitive that may be useful in other contexts, and from which we can easily implement `is-in-same-set`.

Now the question is, how do we name sets? The real question we should ask is, what operations do we care to perform on these names? All we care about is, given two names, they represent the same set precisely when the names are the same. Therefore, we could construct a new string, or number, or something else, but we have another option: simply pick some element of the set to represent it, i.e., to serve as its name. Thus we will associate each set element with an indicator of the "set name" for that element; if there isn't one, then its name is itself (the `none` case of `parent`):

```
data Element<T>:
    | elt(val : T, parent :: Option<Element>)
end
```

We will assume we have some equality predicate for checking when two elements are the same, which we do by comparing their value parts, ignoring their parent values:

```
fun is-same-element(e1, e2): e1.val <=> e2.val end
```

Do Now!

Why do we check only the value parts?

We will assume that for a given set, we always return the same representative element. (Otherwise, equality will fail even though we have the same set.) Thus:We've used the name `fynd` because `find` is already defined to mean something else in Pyret. If you don't like the misspelling, you're welcome to use a longer name like `find-root`.

```
fun is-in-same-set(e1 :: Element, e2 :: Element, s :: Sets)
    -> Boolean:
        s1 = fynd(e1, s)
        s2 = fynd(e2, s)
        identical(s1, s2)
end
```

where `Sets` is the list of all elements:

```
type Sets = List<Element>
```

How do we find the representative element for a set? We first find it using `is-same-element`; when we do, we check the element's `parent` field. If it is `none`, that means this very element names its set; this can happen either because the element is a singleton set (we'll initialize all elements with `none`), or it's the name for some larger set. Either way, we're done. Otherwise, we have to recursively find the parent:

```
fun fynd(e :: Element, s :: Sets) -> Element:
    cases (List) s:
        | empty => raise("fynd: shouldn't have gotten here")
        | link(f, r) =>
            if is-same-element(f, e):
                cases (Option) f.parent:
                    | none => f
                    | some(p) => fynd(p, s)
                end
            else:
                fynd(e, r)
            end
        end
    end
end
```

Exercise

Why is there a recursive call in the nested `cases`?

What's left is to implement `union`. For this, we find the representative elements of the two sets we're trying to union; if they are the same, then the two sets are already in a union; otherwise, we have to update the data structure:

```
fun union(e1 :: Element, e2 :: Element, s :: Sets) -> Sets:
    s1 = fynd(e1, s)
    s2 = fynd(e2, s)
    if identical(s1, s2):
        s
    else:
        update-set-with(s, s1, s2)
    end
end
```

To update, we arbitrarily choose one of the set names to be the name of the new compound set. We then have to update the parent of the other set's name element to be this one:

```
fun update-set-with(s :: Sets, child :: Element, parent :: Element)
    -> Sets:
    cases (List) s:
        | empty => raise("update: shouldn't have gotten here")
        | link(f, r) =>
            if is-same-element(f, child):
                link(elt(f.val, some(parent)), r)
            else:
                link(f, update-set-with(r, child, parent))
            end
        end
    end
end
```

Here are some tests to illustrate this working:

```
check:
s0 = map(elt(_, none), [list: 0, 1, 2, 3, 4, 5, 6, 7])
s1 = union(get(s0, 0), get(s0, 2), s0)
s2 = union(get(s1, 0), get(s1, 3), s1)
s3 = union(get(s2, 3), get(s2, 5), s2)
print(s3)
```

```

is SAME ELEMENT(fynd(get(s0, 0), s3), fynd(get(s0, 5), s3)) IS TRUE
is SAME ELEMENT(fynd(get(s0, 2), s3), fynd(get(s0, 5), s3)) IS TRUE
is SAME ELEMENT(fynd(get(s0, 3), s3), fynd(get(s0, 5), s3)) IS TRUE
is SAME ELEMENT(fynd(get(s0, 5), s3), fynd(get(s0, 5), s3)) IS TRUE
is SAME ELEMENT(fynd(get(s0, 7), s3), fynd(get(s0, 7), s3)) IS TRUE

```

end

Unfortunately, this implementation suffers from two major problems:

- First, because we are performing functional updates, the value of the `parent` reference keeps “changing”, but these changes are not visible to older copies of the “same” value. An element from different stages of unioning has different parent references, even though it is arguably the same element throughout. This is a place where functional programming hurts.
- Relatedly, the performance of this implementation is quite bad. recursively traverses parents to find the set’s name, but the elements traversed are not updated to record this new name. We certainly could update them by reconstructing the set afresh each time, but that complicates the implementation and, as we will soon see, we can do much better.

Even worse, it may not even be correct!

Exercise

Is it? Consider constructing `unions` that are not quite so skewed as above, and see whether you get the results you expect.

The bottom line is that pure functional programming is not a great fit with this problem. We need a better implementation strategy: Union-Find.

[contents](#) ← [prev](#) [up](#) [next](#) →

17.1 Representing Sets as Lists

17.1 Representing Sets as Lists Earlier [Sets as Collective Data] we introduced sets. Recall that the elements of a set have no specific order, and ignore duplicates. If these ideas are not familiar, please read Sets as Collective Data, since they will be important when discussing the representation of sets. At that time we relied on Pyret's built-in representation of sets. Now we will discuss how to build sets for ourselves. In what follows, we will focus only on sets of numbers.

We will start by discussing how to represent sets using lists. Intuitively, using lists to represent sets of data seems problematic, because lists respect both order and duplication. For instance,

```
check:  
  [list: 1, 2, 3] is [list: 3, 2, 1, 1]  
end
```

fails, but the corresponding sets are equal.

In principle, we want sets to obey the following interface: Note that a type called `Set` is already built into Pyret, so below we will use the name `LSet` for a set represented as a list.

```
<set-operations> ::=
```

```
mt-set :: Set  
is-in :: (T, Set<T> -> Bool)  
insert :: (T, Set<T> -> Set<T>)  
union :: (Set<T>, Set<T> -> Set<T>)  
size :: (Set<T> -> Number)  
to-list :: (Set<T> -> List<T>)
```

We may also find it also useful to have functions such as

```
insert-many :: (List<T>, Set<T> -> Set<T>)
```

which, combined with `mt-set`, easily gives us a `to-set` function.

Sets can contain many kinds of values, but not necessarily any kind: we need to be able to check for two values being equal (which is a requirement for a set, but not for a list!), which can't be done with all values (such as functions). We discuss the nuances of this elsewhere [Equality and Ordering]. For now, we can ignore these issues by focusing on sets of (non-rough) numbers.

17.1.1 Representation Choices The empty list can stand in for the empty set—

```
type LSet = List  
mt-set = empty
```

—and we can presumably define `size` as

```
fun size<T>(s :: LSet<T>) -> Number:  
  s.length()  
end
```

However, this reduction (of sets to lists) can be dangerous:

1. There is a subtle difference between lists and sets. The list

```
[list: 1, 1]
```

is not the same as

```
[list: 1]
```

because the first list has length two whereas the second has length one. Treated as a set, however, the two are the same: they both have size one. Thus, our implementation of `size` above is incorrect if we don't take into account duplicates (either during insertion or while computing the size).

2. We might falsely make assumptions about the order in which elements are retrieved from the set due to the ordering guaranteed provided by the underlying list representation. This might hide bugs that we don't discover until we change the representation.
3. We might have chosen a set representation because we didn't need to care about order, and expected lots of duplicate items. A list representation might store all the duplicates, resulting in significantly more memory use (and slower programs) than we expected.

To avoid these perils, we have to be precise about how we're going to use lists to represent sets. One key question (but not the only one, as we'll soon see [Choosing Between Representations]) is what to do about duplicates. One possibility is for `insert` to check whether an element is already in the set and, if so, leave the representation unchanged; this incurs a cost during insertion but avoids unnecessary duplication and lets us use `length` to implement `size`. The other option is to define `insert` as `link`—literally,

```
insert = link
```

—and have some other procedure perform the filtering of duplicates.

17.1.2 Time Complexity What is the complexity of this representation of sets? Let's consider just `insert`, `is-in`, and `size`. Suppose the size of the set is $\langle k \rangle$ (where, to avoid ambiguity, we let $\langle k \rangle$ represent the number of distinct elements). The complexity of these operations depends on whether or not we store duplicates:

- If we don't store duplicates, then `size` is simply `length`, which takes time linear in $\langle k \rangle$. Similarly, `is-in` only needs to traverse the list once to determine whether or not an element is present, which also takes time linear in $\langle k \rangle$. But `insert` needs to check whether an element is already present, which takes time linear in $\langle k \rangle$, followed by at most a constant-time operation (`link`).
- If we do store duplicates, then `insert` is constant time: it simply `links` on the new element without regard to whether it already is in the set representation. `is-in` traverses the list once, but the number of elements it needs to visit could be significantly greater than $\langle k \rangle$, depending on how many duplicates have been added. Finally, `size` needs to check whether or not each element is duplicated before counting it.

Do Now!

What is the time complexity of `size` if the list has duplicates?

One implementation of `size` is

```
fun size<T>(s :: LSet<T>) -> Number:  
  cases (List) s:  
    | empty => 0  
    | link(f, r) =>  
      if r.member(f):  
        size(r)  
      else:  
        1 + size(r)  
      end  
    end  
  end
```

Let's now compute the complexity of the body of the function, assuming the number of distinct elements in `s` is $\langle k \rangle$ but the actual number of elements in `s` is $\langle d \rangle$, where $\langle d \geq k \rangle$. To compute the time to run `size` on $\langle d \rangle$ elements, $\langle T(d) \rangle$, we should determine the number of operations in each question and answer. The first question has a constant number of operations, and the first answer also a constant. The second question also has a constant number of operations. Its answer is a conditional, whose first question (`r.member(f)`) needs to traverse

the entire list, and hence has $\mathcal{O}(k \rightarrow d)$ operations. If it succeeds, we recur on something of size $\mathcal{T}(d-1)$; else we do the same but perform a constant more operations. Thus $\mathcal{T}(0)$ is a constant, while the recurrence (in big-Oh terms) is

$$\begin{aligned} \text{begin}\{\text{equation}^*\} T(d) &= d + T(d-1) \\ \text{end}\{\text{equation}^*\} \end{aligned}$$

Thus $\mathcal{T} \in \mathcal{O}(d \rightarrow d^2)$. Note that this is quadratic in the number of elements in the list, which may be much bigger than the size of the set.

17.1.3 Choosing Between Representations Now that we have two representations with different complexities, it's worth thinking about how to choose between them. To do so, let's build up the following table. The table distinguishes between the interface (the set) and the implementation (the list), because—owing to duplicates in the representation—these two may not be the same. In the table we'll consider just two of the most common operations, insertion and membership checking:

With Duplicates

Without Duplicates

insert

is-in

insert

is-in

Size of Set

constant

linear

linear

Size of List

constant

linear

linear

linear

A naive reading of this would suggest that the representation with duplicates is better because it's sometimes constant and sometimes linear, whereas the version without duplicates is always linear. However, this masks a very important distinction: what the linear means. When there are no duplicates, the size of the list is the same as the size of the set. However, with duplicates, the size of the list can be arbitrarily larger than that of the set!

Based on this, we can draw several lessons:

1. Which representation we choose is a matter of how much duplication we expect. If there won't be many duplicates, then the version that stores duplicates pays a small extra price in return for some faster operations.
2. Which representation we choose is also a matter of how often we expect each operation to be performed. The representation without duplication is "in the middle": everything is roughly equally expensive (in the worst case). With duplicates is "at the extremes": very cheap insertion, potentially very expensive membership. But if we will mostly only insert without checking membership, and especially if we know membership checking will only occur in situations where we're willing to wait, then permitting duplicates may in fact be the smart choice. (When might we ever be in such a situation? Suppose your set represents a backup data structure; then we add lots of data but very rarely—indeed, only in case of some catastrophe—ever need to look for things in it.)
3. Another way to cast these insights is that our form of analysis is too weak. In situations where the complexity depends so heavily on a particular sequence of operations, big-Oh is too loose and we should instead study the complexity of specific sequences of operations. We will address precisely this question later [Halloween Analysis].

Moreover, there is no reason a program should use only one representation. It could well begin with one representation, then switch to another as it better understands its workload. The only thing it would need to do to switch is to convert all existing data between the representations.

How might this play out above? Observe that data conversion is very cheap in one direction: since every list without duplicates is automatically also a list with (potential) duplicates, converting in that direction is trivial (the representation stays unchanged, only its interpretation changes). The other direction is harder: we have to filter duplicates (which takes time quadratic in the number of elements in the list). Thus, a program can make an initial guess about its workload and pick a representation accordingly, but maintain statistics as it runs and, when it finds its assumption is wrong, switch representations—and can do so as many times as needed.

17.1.4 Other Operations

Exercise

Implement the remaining operations catalogued above (<set-operations>) under each list representation.

Exercise

Implement the operation

```
remove :: (Set<T>, T -> Set<T>)
```

under each list representation (renaming **Set** appropriately. What difference do you see?

Do Now!

Suppose you're asked to extend sets with these operations, as the set analog of **first** and **rest**:

```
one :: (Set<T> -> T)
others :: (Set<T> -> T)
```

You should refuse to do so! Do you see why?

With lists the "first" element is well-defined, whereas sets are defined to have no ordering. Indeed, just to make sure users of your sets don't accidentally assume anything about your implementation (e.g., if you implement **one** using **first**, they may notice that **one** always returns the element most recently added to the list), you really ought to return a random element of the set on each invocation.

Unfortunately, returning a random element means the above interface is unusable. Suppose `s` is bound to a set containing 1, 2, and 3. Say the first time `one(s)` is invoked it returns 2, and the second time 1. (This already means `one` is not a function.) The third time it may again return 2. Thus `others` has to remember which element was returned the last time `one` was called, and return the set sans that element. Suppose we now invoke `one` on the result of calling `others`. That means we might have a situation where `one(s)` produces the same result as `one(others(s))`.

Exercise

Why is it unreasonable for `one(s)` to produce the same result as `one(others(s))`?

Exercise

Suppose you wanted to extend sets with a `subset` operation that partitioned the set according to some condition. What would its type be?

Exercise

The types we have written above are not as crisp as they could be. Define a `has-no-duplicates` predicate, refine the relevant types with it, and check that the functions really do satisfy this criterion.

contents ← prev up next →

17.2 Making Sets Grow on Trees

17.2 Making Sets Grow on Trees In Representing Sets as Lists we saw multiple list representations of sets. They all came with at least some operations having linear time complexity—linear in different ways, but always linear in at least the number of distinct elements in the set. Can we do better?

Let's start by noting that it seems better, if at all possible, to avoid storing duplicates. Duplicates are only problematic during insertion due to the need for a membership test. But if we can make membership testing cheap, then we would be better off using it to check for duplicates and storing only one instance of each value (which also saves us space). Thus, let's try to improve the time complexity of membership testing (and, hopefully, of other operations too).

It seems clear that with a (duplicate-free) list representation of a set, we cannot really beat linear time for membership checking. This is because at each step, we can eliminate only one element from contention which in the worst case requires a linear amount of work to examine the whole set. Instead, we need to eliminate many more elements with each comparison—more than just a constant.

In our handy set of recurrences [Solving Recurrences], one stands out: $\mathcal{O}(T(k) = T(k/2) + c)$. It says that if, with a constant amount of work we can eliminate half the input, we can perform membership checking in logarithmic time. This will be our goal.

Before we proceed, it's worth putting logarithmic growth in perspective. Asymptotically, logarithmic is obviously not as nice as constant. However, logarithmic growth is very pleasant because it grows so slowly. For instance, if an input doubles from size $\mathcal{O}(k)$ to $\mathcal{O}(2k)$, its logarithm—and hence resource usage—grows only by $\mathcal{O}(\log 2k - \log k = \log 2)$, which is a constant. Indeed, for just about all problems, practically speaking the logarithm of the input size is bounded by a constant (that isn't even very large). Therefore, in practice, for many programs, if we can shrink our resource consumption to logarithmic growth, it's probably time to move on and focus on improving some other part of the system.

We have actually just made an extremely subtle assumption. In the list representation of sets, when we check one element for membership and eliminate it, we have eliminated only that one element. To obtain this logarithmic complexity, we need comparing against one element to remove an entire set of elements. Because we are constructing sets of numbers, we don't need to confront this issue here. Instead, we go into it in much more detail in Converting Values to Ordered Values.

17.2.1 Using Binary Trees

Because logs come from trees.

Clearly, a list representation does not let us eliminate half the elements with a constant amount of work; instead, we need a tree. Thus we define a binary tree of (for simplicity) numbers:

```
data BT:  
| leaf  
| node(v :: Number, l :: BT, r :: BT)  
end
```

Given this definition, let's define the membership checker:

```
fun is-in-bt(e :: Number, s :: BT) -> Boolean:  
cases (BT) s:  
| leaf => false  
| node(v, l, r) =>  
  if e == v:  
    true
```

```

    else:
        is-in-bt(e, l) or is-in-bt(e, r)
    end
end

```

Oh, wait. If the element we're looking for isn't the root, what do we do? It could be in the left child or it could be in the right; we won't know for sure until we've examined both. Thus, we can't throw away half the elements; the only one we can dispose of is the value at the root. Furthermore, this property holds at every level of the tree. Thus, membership checking needs to examine the entire tree, and we still have complexity linear in the size of the set.

How can we improve on this? The comparison needs to help us eliminate not only the root but also one whole sub-tree. We can only do this if the comparison "speaks for" an entire sub-tree. It can do so if all elements in one sub-tree are less than or equal to the root value, and all elements in the other sub-tree are greater than or equal to it. Of course, we have to be consistent about which side contains which subset; it is conventional to put the smaller elements to the left and the bigger ones to the right. This refines our binary tree definition to give us a binary search tree (BST).

Do Now!

Here is a candidate predicate for recognizing when a binary tree is in fact a binary search tree:

```

fun is-a-bst-buggy(b :: BT) -> Boolean:
    cases (BT) b:
        | leaf => true
        | node(v, l, r) =>
            (is-leaf(l) or (l.v <= v)) and
            (is-leaf(r) or (v <= r.v)) and
            is-a-bst-buggy(l) and
            is-a-bst-buggy(r)
    end
end

```

Is this definition correct?

It's not. To actually throw away half the tree, we need to be sure that everything in the left sub-tree is less than the value in the root and similarly, everything in the right sub-tree is greater than the root. We have used \leq instead of $<$ above because even though we don't want to permit duplicates when representing sets, in other cases we might not want to be so stringent; this way we can reuse the above implementation for other purposes. But the definition above performs only a "shallow" comparison. Thus we could have a root a with a right child, b , such that $b > a$; and the b node could have a left child c such that $c < b$; but this does not guarantee that $c > a$. In fact, it is easy to construct a counter-example that passes this check:

```

check:
    node(5, node(3, leaf, node(6, leaf, leaf)), leaf)
        satisfies is-a-bst-buggy # FALSE!
end

```

Exercise

Fix the BST checker.

With a corrected definition, we can now define a refined version of binary trees that are search trees:

```
type BST = BT%(is-a-bst)
```

We can also remind ourselves that the purpose of this exercise was to define sets, and define **TSets** to be tree sets:

```
type TSet = BST
mt-set = leaf
```

Now let's implement our operations on the BST representation. First we'll write a template:

```
fun is-in(e :: Number, s :: BST) -> Bool:
```

```

cases (BST) s:
| leaf => ...
| node(v, l :: BST, r :: BST) => ...
... is-in(l) ...
... is-in(r) ...
end
end

```

Observe that the data definition of a BST gives us rich information about the two children: they are each a BST, so we know their elements obey the ordering property. We can use this to define the actual operations:

```

fun is-in(e :: Number, s :: BST) -> Boolean:
cases (BST) s:
| leaf => false
| node(v, l, r) =>
  if e == v:
    true
  else if e < v:
    is-in(e, l)
  else if e > v:
    is-in(e, r)
  end
end
end

fun insert(e :: Number, s :: BST) -> BST:
cases (BST) s:
| leaf => node(e, leaf, leaf)
| node(v, l, r) =>
  if e == v:
    s
  else if e < v:
    node(v, insert(e, l), r)
  else if e > v:
    node(v, l, insert(e, r))
  end
end
end

```

In both functions we are strictly assuming the invariant of the BST, and in the latter case also ensuring it. Make sure you identify where, why, and how.

You should now be able to define the remaining operations. Of these, `size` clearly requires linear time (since it has to count all the elements), but because `is-in` and `insert` both throw away one of two children each time they recur, they take logarithmic time.

Exercise

Suppose we frequently needed to compute the size of a set. We ought to be able to reduce the time complexity of `size` by having each tree cache its size, so that `size` could complete in constant time (note that the size of the tree clearly fits the criterion of a cache, since it can always be reconstructed). Update the data definition and all affected functions to keep track of this information correctly.

17.2.2 Checking the Complexity But wait a minute. Are we actually done? Our recurrence takes the form $\mathcal{O}(T(k) = T(k/2) + c)$, but what in our data definition guaranteed that the size of the child traversed by `is-in` will be half the size?

Do Now!

Construct an example—consisting of a sequence of `inserts` to the empty tree—such that the resulting tree is not balanced. Show that searching for certain elements in this tree will take linear, not logarithmic, time in its size.

Imagine starting with the empty tree and inserting the values 1, 2, 3, and 4, in order. The resulting tree would be `check`:

```
insert(4, insert(3, insert(2, insert(1, mt-set)))) is
  node(1, leaf,
    node(2, leaf,
      node(3, leaf,
        node(4, leaf, leaf))))
end
```

Searching for 4 in this tree would have to examine all the set elements in the tree. In other words, this binary search tree is degenerate—it is effectively a list, and we are back to having the same complexity we had earlier.

Therefore, using a binary tree, and even a BST, does not guarantee the complexity we want: it does only if our inputs have arrived in just the right order. However, we cannot assume any input ordering; instead, we would like an implementation that works in all cases. Thus, we must find a way to ensure that the tree is always balanced, so each recursive call in `is-in` really does throw away half the elements.

Exercise

Observe that we have not talked about computing the size of the set. Even if we could assume that the binary tree is balanced, how do we determine the size in logarithmic-or-better time?

17.2.3 A Fine Balance: Tree Surgery Let's define a balanced binary search tree (BBST). It must obviously be a search tree, so let's focus on the “balanced” part. We have to be careful about precisely what this means: we can't simply expect both sides to be of equal size because this demands that the tree (and hence the set) have an even number of elements and, even more stringently, to have a size that is a power of two.

Exercise

Define a predicate for a BBST that consumes a BT and returns a Boolean indicating whether or not it a balanced search tree.

Therefore, we relax the notion of balance to one that is both accommodating and sufficient. We use the term balance factor for a node to refer to the height of its left child minus the height of its right child (where the height is the depth, in edges, of the deepest node). We allow every node of a BBST to have a balance factor of $\backslash(-1\backslash)$, $\backslash(0\backslash)$, or $\backslash(1\backslash)$ (but nothing else): that is, either both have the same height, or the left or the right can be one taller. Note that this is a recursive property, but it applies at all levels, so the imbalance cannot accumulate making the whole tree arbitrarily imbalanced.

Exercise

Given this definition of a BBST, show that the number of nodes is exponential in the height. Thus, always recurring on one branch will terminate after a logarithmic (in the number of nodes) number of steps.

Here is an obvious but useful observation: every BBST is also a BST (this was true by the very definition of a BBST). Why does this matter? It means that a function that operates on a BST can just as well be applied to a BBST without any loss of correctness.

So far, so easy. All that leaves is a means of creating a BBST, because it's responsible for ensuring balance. It's easy to see that the constant `empty-set` is a BBST value. So that leaves only `insert`.

Here is our situation with `insert`. Assuming we start with a BBST, we can determine in logarithmic time whether the element is already in the tree and, if so, ignore it. To implement a bag we count how many of each element are in it, which does not affect the tree's height. When inserting an element, given balanced trees, the `insert` for a BST takes only a logarithmic amount of time to perform the insertion. Thus, if performing the insertion does not affect the tree's balance, we're done. Therefore, we only need to consider cases where performing the insertion throws off the balance.

Observe that because $\langle \rangle$ and $\rangle \langle$ are symmetric (likewise with \leq and \geq), we can consider insertions into one half of the tree and a symmetric argument handles insertions into the other half. Thus, suppose we have a tree that is currently balanced into which we are inserting the element e . Let's say e is going into the left sub-tree and, by virtue of being inserted, will cause the entire tree to become imbalanced. Some trees, like family trees (Data Design Problem – Ancestry Data) represent real-world data. It makes no sense to “balance” a family tree: it must accurately model whatever reality it represents. These set-representing trees, in contrast, are chosen by us, not dictated by some external reality, so we are free to rearrange them.

There are two ways to proceed. One is to consider all the places where we might insert e in a way that causes an imbalance and determine what to do in each case.

Exercise

Enumerate all the cases where insertion might be problematic, and dictate what to do in each case.

The number of cases is actually quite overwhelming (if you didn't think so, you missed a few...). Therefore, we instead attack the problem after it has occurred: allow the existing BST `insert` to insert the element, assume that we have an imbalanced tree, and show how to restore its balance. The insight that a tree can be made “self-balancing” is quite remarkable, and there are now many solutions to this problem. This particular one, one of the oldest, is due to G.M. Adelson-Velskii and E.M. Landis. In honor of their initials it is called an AVL Tree, though the tree itself is quite evident; their genius is in defining re-balancing.

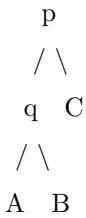
Thus, in what follows, we begin with a tree that is balanced; `insert` causes it to become imbalanced; we have assumed that the insertion happened in the left sub-tree. In particular, suppose a (sub-)tree has a balance factor of 2 (positive because we're assuming the left is imbalanced by insertion). The procedure for restoring balance depends critically on the following property:

Exercise

Show that if a tree is currently balanced, i.e., the balance factor at every node is -1 , 0 , or 1 , then `insert` can at worst make the balance factor ± 2 .

The algorithm that follows is applied as `insert` returns from its recursion, i.e., on the path from the inserted value back to the root. Since this path is of logarithmic length in the set's size (due to the balancing property), and (as we shall see) performs only a constant amount of work at each step, it ensures that insertion also takes only logarithmic time, thus completing our challenge.

To visualize the algorithm, let's use this tree schematic:



Here, p is the value of the element at the root (though we will also abuse terminology and use the value at a root to refer to that whole tree), q is the value at the root of the left sub-tree (so $q < p$), and A , B , and C name the respective sub-trees. We have assumed that e is being inserted into the left sub-tree, which means $e < p$.

Let's say that C is of height k . Before insertion, the tree rooted at q must have had height $k+1$ (or else one insertion cannot create imbalance). In turn, this means A must have had height k or $k-1$, and likewise for B .

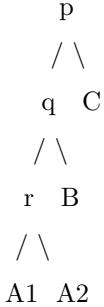
Suppose that after insertion, the tree rooted at q has height $k+2$. Thus, either A or B has height $k+1$ and the other must have height less than that (either k or $k-1$).

Exercise

Why can they both not have height $k+1$ after insertion?

This gives us two cases to consider.

17.2.3.1 Left-Left Case Let's say the imbalance is in $\setminus(A)$, i.e., it has height $\setminus(k+1)$. Let's expand that tree:



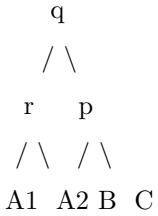
We know the following about the data in the sub-trees. We'll use the notation $\setminus(T < a)$ where $\setminus(T)$ is a tree and $\setminus(a)$ is a single value to mean every value in $\setminus(T)$ is less than $\setminus(a)$.

- $\setminus(A_1 < r)$.
- $\setminus(r < A_2 < q)$.
- $\setminus(q < B < p)$.
- $\setminus(p < C)$.

Let's also remind ourselves of the sizes:

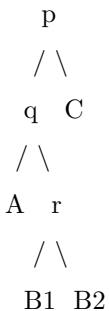
- The height of $\setminus(A_1)$ or of $\setminus(A_2)$ is $\setminus(k)$ (the cause of imbalance).
- The height of the other $\setminus(A_i)$ is $\setminus(k-1)$ (see the exercise above).
- The height of $\setminus(C)$ is $\setminus(k)$ (initial assumption; $\setminus(k)$ is arbitrary).
- The height of $\setminus(B)$ must be $\setminus(k-1)$ or $\setminus(k)$ (argued above).

Imagine this tree is a mobile, which has gotten a little skewed to the left. You would naturally think to suspend the mobile a little further to the left to bring it back into balance. That is effectively what we will do:



Observe that this preserves each of the ordering properties above. In addition, the $\setminus(A)$ subtree has been brought one level closer to the root than earlier relative to $\setminus(B)$ and $\setminus(C)$. This restores the balance (as you can see if you work out the heights of each of $\setminus(A_i)$, $\setminus(B)$, and $\setminus(C)$). Thus, we have also restored balance.

17.2.3.2 Left-Right Case The imbalance might instead be in $\setminus(B)$. Expanding:



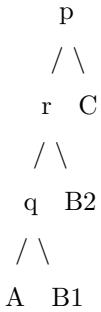
Again, let's record what we know about data order:

- $\setminus(A < q\setminus)$.
- $\setminus(q < B_1 < r\setminus)$.
- $\setminus(r < B_2 < p\setminus)$.
- $\setminus(p < C\setminus)$.

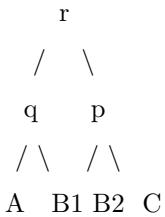
and sizes:

- Suppose the height of $\setminus(C\setminus)$ is $\setminus(k\setminus)$.
- The height of $\setminus(A\setminus)$ must be $\setminus(k-1\setminus)$ or $\setminus(k\setminus)$.
- The height of $\setminus(B_1\setminus)$ or $\setminus(B_2\setminus)$ must be $\setminus(k\setminus)$, but not both (see the exercise above). The other must be $\setminus(k-1\setminus)$.

We therefore have to somehow bring $\setminus(B_1\setminus)$ and $\setminus(B_2\setminus)$ one level closer to the root of the tree. By using the above data ordering knowledge, we can construct this tree:



Of course, if $\setminus(B_1\setminus)$ is the problematic sub-tree, this still does not address the problem. However, we are now back to the previous (left-left) case; rotating gets us to:



Now observe that we have precisely maintained the data ordering constraints. Furthermore, from the root, $\setminus(A\setminus)$'s lowest node is at height $\setminus(k+1\setminus)$ or $\setminus(k+2\setminus)$; so is $\setminus(B_1\setminus)$'s; so is $\setminus(B_2\setminus)$'s; and $\setminus(C\setminus)$'s is at $\setminus(k+2\setminus)$.

17.2.3.3 Any Other Cases? Were we a little too glib before? In the left-right case we said that only one of $\setminus(B_1\setminus)$ or $\setminus(B_2\setminus)$ could be of height $\setminus(k\setminus)$ (after insertion); the other had to be of height $\setminus(k-1\setminus)$. Actually, all we can say for sure is that the other has to be at most height $\setminus(k-2\setminus)$.

Exercise

- Can the height of the other tree actually be $\setminus(k-2\setminus)$ instead of $\setminus(k-1\setminus)$?
- If so, does the solution above hold? Is there not still an imbalance of two in the resulting tree?
- Is there actually a bug in the above algorithm?

contents ← prev up next →

17.3 Union-Find

17.3 Union-Find We have previously [Checking Component Connectedness] seen how to check connectedness of components, but found that solution unsatisfactory. Recall that it comes down to two set operations: we want to construct the unions of sets, and then determine whether two elements are in the same set.

We will now see how to do this using state. We will try to keep things as similar to the previous version as possible, to enhance comparison.

17.3.1 Implementing with State First, we have to update the definition of an element, making the `parent` field be mutable:

```
data Element:  
    | elt(val, ref parent :: Option<Element>)  
end
```

To determine whether two elements are in the same set, we will still rely on `fynd`. However, as we will soon see, `fynd` no longer needs to be given the entire set of elements. Because the only reason `is-in-same-set` consumed that set was to pass it on to `fynd`, we can remove it from here. Nothing else changes:

```
fun is-in-same-set(e1 :: Element, e2 :: Element) -> Boolean:  
    s1 = fynd(e1)  
    s2 = fynd(e2)  
    identical(s1, s2)  
end
```

Updating is now the crucial difference: we use mutation to change the value of the parent:

```
fun update-set-with(child :: Element, parent :: Element):  
    child!{parent: some(parent)}  
end
```

In `parent: some(parent)`, the first `parent` is the name of the field, while the second one is the parameter name. In addition, we must use `some` to satisfy the option type. Naturally, it is not `none` because the entire point of this mutation is to change the parent to be the other element, irrespective of what was there before.

Given this definition, `union` also stays largely unchanged, other than the change to the return type. Previously, it needed to return the updated set of elements; now, because the update is performed by mutation, there is no longer any need to return anything:

```
fun union(e1 :: Element, e2 :: Element):  
    s1 = fynd(e1)  
    s2 = fynd(e2)  
    if identical(s1, s2):  
        s1  
    else:  
        update-set-with(s1, s2)  
    end  
end
```

Finally, `fynd`. Its implementation is now remarkably simple. There is no longer any need to search through the set. Previously, we had to search because after union operations have occurred, the parent reference might have no longer been valid. Now, any such changes are automatically reflected by mutation. Hence:

```

fun fynd(e :: Element) -> Element:
  cases (Option) e!parent:
    | none => e
    | some(p) => fynd(p)
  end
end

```

17.3.2 Optimizations Look again at . In the `some` case, the element bound to `e` is not the set name; that is obtained by recursively traversing `parent` references. As this value returns, however, we don't do anything to reflect this new knowledge! Instead, the next time we try to find the parent of this element, we're going to perform this same recursive traversal all over again.

Using mutation helps address this problem. The idea is as simple as can be: compute the value of the parent, and update it.

```

fun fynd(e :: Element) -> Element:
  cases (Option) e!parent block:
    | none => e
    | some(p) =>
      new-parent = fynd(p)
      e!{parent: some(new-parent)}
      new-parent
    end
end

```

Note that this update will apply to every element in the recursive chain to find the set name. Therefore, applying to any of those elements the next time around will benefit from this update. This idea is called path compression.

There is one more interesting idea we can apply. This is to maintain a rank of each element, which is roughly the depth of the tree of elements for which that element is their set name. When we union two elements, we then make the one with larger rank the parent of the one with the smaller rank. This has the effect of avoiding growing very tall paths to set name elements, instead tending towards “bushy” trees. This too reduces the number of parents that must be traversed to find the representative.

17.3.3 Analysis This optimized union-find data structure has a remarkable analysis. In the worst case, of course, we must traverse the entire chain of parents to find the name element, which takes time proportional to the number of elements in the set. However, once we apply the above optimizations, we never need to traverse that same chain again! In particular, if we conduct an amortized analysis over a sequence of set equality tests after a collection of union operations, we find that the cost for subsequent checks is very small—indeed, about as small a function can get without being constant. The actual analysis is quite sophisticated; it is also one of the most remarkable algorithm analyses in all of computer science.

contents ← prev up next →

17.4 Hashes, Sets, and Key-Values

17.4 Hashes, Sets, and Key-Values We have seen several solutions to set membership [Several Variations on Sets]. In particular, trees [Making Sets Grow on Trees] gave us logarithmic complexity for insertion and membership. Now we will see one more implementation of sets, with different complexity. To set this up, we assume you are familiar with the concept of hashing [Converting Values to Ordered Values], which we saw was useful for constructing search trees. Here, we will use it to construct sets in a very different way. We will then generalize sets to another important data structure: key-value repositories. But first...

17.4.1 A Hash Function for Strings As we have seen in Converting Values to Ordered Values, we have multiple strategies for converting arbitrary values into numbers, which we will rely on here. Therefore, we could write this material around numbers alone. To make the examples more interesting, and to better illustrate some real-world issues, we will instead use strings. To hash them, we will use `hash-of`, defined there, which simply adds up a string's code points.

We use this function for multiple reasons. First, it is sufficient to illustrate some of the consequences of hashing. Second, in practice, when built-in hashing does not suffice, we do write (more complex versions of) functions like it. And finally, because it's all laid bare, it's easy for us to experiment with.

17.4.2 Sets from Hashing Suppose we are given a set of strings. We can hash each element of that set. Each string is now mapped to a number. Each of these numbers is a member of the set; every other number is not a member of this set.

Therefore, a simple representation is to just store this list of numbers. For instance, we can store the list `[list: "Hello", "World!", "Attention! ;Danger! "]` as `[list: 500, 553, 195692]`.

Unfortunately, this does not help very much. Insertion can be done in constant time, but checking membership requires us to traverse the entire list, which takes linear time in the worst case. Alternatively, maybe we have some clever scheme that involves sorting the list. But note:

- inserting the element can now take as much as linear time; or,
- we store the elements as a tree instead of a list, but then
 1. we have to make sure the tree is balanced, so
 2. we will have essentially reconstructed the BBST.

In other words, we are recapitulating the discussion from Representing Sets as Lists and Making Sets Grow on Trees.

Notice that the problem here is traversal: if we have to visit more than a constant number of elements, we have probably not improved anything over the BBST. So, given a hash, how can we perform only a constant amount of work? For that, lists and trees don't work: they both require at least some amount of (non-constant) traversal to get to an arbitrary element. Instead we need a different data structure...

17.4.3 Arrays Arrays are another linear data structure, like lists. There are two key differences between lists and arrays that reflect each one's strength and weakness.

The main benefit to arrays is that we can access any element in the array in constant time. This is in contrast to lists where, to get to the $\backslash(n\backslash)$ th element, we have to first traverse the previous $\backslash(n-1\backslash)$ elements (using successive `rests`).

However, this benefit comes at a cost. The reason arrays can support constant-time access is because the size of an array is fixed at creation time. Thus, while we can keep extending a list using link, we cannot grow the size of an

array “in place”; rather, we must make a new array and copy the entire array’s content into the new array, which takes linear time. (We can do a better job of this by using Halloween Analysis, but there is no real free ride.)

The arrays in Pyret are documented here. While not necessary in principle, it is conventional to think of arrays as data structures that support mutation, and that is how we will use them here.

17.4.4 Sets from Hashing and Arrays Okay, so now we have a strategy. When we want to insert a string into the set, we compute its hash, go to the corresponding location in the array, and record the presence of that string. If we want to check for membership, we similarly compute its hash and see whether the corresponding location has been set. Traditionally, each location in the array is called a bucket, and this data structure is called a hashtable.

```
BUCKET-COUNT = 1000
```

```
buckets = array-of(false, BUCKET-COUNT)

fun insert(s :: String):
    h = hash-of(s)
    buckets.set-now(h, true)
end

fun is-in(s :: String):
    h = hash-of(s)
    buckets.get-now(h)
end
```

Observe that if this were to work, we would have constant time insertion and membership checking. Unfortunately, two things make this plan untenable in general.

17.4.5 Collisions First, our choice of hash function. For the above scheme to work, two different strings have to map to two different locations.

Do Now!

Is the above hash function invertible?

We just need to find two strings that have the same hash. Given the definition of hash-of, it’s easy to see that any rearrangement of the letters produces the same hash:

```
hash-of("Hello")
500
hash-of("olleH")
500
```

Similarly, this test suite passes:

```
check:
    hash-of("Hello") is hash-of("olleH")
    hash-of("Where") is hash-of("Weird")
    hash-of("Where") is hash-of("Wired")
    hash-of("Where") is hash-of("Whine")
end
```

When multiple values hash to the same location, we call this a hash collision.

Hash-collisions are problematic! With the above hash function, we get:

```
check:
    insert("Hello")
    is-in("Hello") is true
    is-in("Where") is false
    is-in("elloH") is true
```

```
end
```

where two of these tests are desirable but the third is definitely not.

Note that collisions are virtually inevitable. If we have uniformly distributed data, then collisions show up sooner than we might expect. This follows from the reasoning behind what is known as the birthday problem, commonly presented as how many people need to be in a room before the likelihood that two of them share a birthday exceeds some percentage. For the likelihood to exceed half we need just 23 people! Therefore, it is wise to prepare for the possibility of collisions.

The key is to know something about the distribution of hash values. For instance, if we knew our hash values are all multiples of 10, then using a table size of 10 would be a terrible idea (because all elements would hash to the same bucket, turning our hash table into a list). In practice, it is common to use uncommon prime numbers as the table size, since a random value is unlikely to have it as a divisor. This does not yield a theoretical improvement (unless you can make certain assumptions about the input, or work through the math very carefully), but it works well in practice. In particular, since the typical hashing function uses memory addresses for objects on the heap, and on most systems these addresses are multiples of 4, using a prime like 31 is often a fairly good bet.

While collisions are probabilistic, and depend on the choice of hash function, we have an even more fundamental and unavoidable reason for collisions. We have to store an array of the largest possible hash size. However, not only can hash values be very large (try to run `insert("Attention! ;Danger! ")` and see what happens), there isn't even an *a priori* limit to the size of a hash. This fundamentally flies in the face of arrays, which must have a fixed size.

To handle arbitrarily large values, we:

- use an array size that is reasonable given our memory constraints
- use the remainder of the hash relative to the array's size to find the bucket

That is:

```
fun insert(s :: String):  
    h = hash-of(s)  
    buckets.set-now(num-remainder(h, BUCKET-COUNT), true)
```

```
end
```

```
fun is-in(s :: String):  
    h = hash-of(s)  
    buckets.get-now(num-remainder(h, BUCKET-COUNT))
```

```
end
```

This addresses the second problem: we can also store the pirate flag:

```
check:  
    is-in("Attention! ;Danger! ") is false  
    insert("Attention! ;Danger! ")  
    is-in("Attention! ;Danger! ") is true
```

```
end
```

Observe, however, we have simply created yet another source of collisions: the remainder computation. If we have 10 buckets, then the hashes 5, 15, 25, 35, ... all refer to the same bucket. Thus, there are two sources of collision, and we have to deal with them both.

17.4.6 Resolving Collisions Surprisingly or disappointingly, we have a very simple solution to the collision problems. Each bucket is not a single Boolean value, but rather a list of the actual values that hashed to that bucket. Then, we just check for membership in that list.

First, we will abstract over finding the bucket number in `insert` and `is-in`:

```
fun index-of(s :: String):  
    num-remainder(hash-of(s), BUCKET-COUNT)
```

```
end
```

Next, we change what is held in each bucket: not a Boolean, but rather a list of the actual strings:

```
buckets = array-of(empty, BUCKET-COUNT)
```

Now we can write the more nuanced membership checker:

```
fun is-in(s :: String):
    b = index-of(s)
    member(buckets.get-now(b), s)
end
```

Similarly, when inserting, we first make sure the element isn't already there (to avoid the complexity problems caused by having duplicates), and only then insert it:

```
fun insert(s :: String):
    b = index-of(s)
    l = buckets.get-now(b)
    when not(member(l, s)):
        buckets.set-now(b, link(s, l))
    end
end
```

Now our tests pass as intended:

```
check:
    insert("Hello")
    is-in("Hello") is true
    is-in("Where") is false
    is-in("elloH") is false
end
```

17.4.7 Complexity Now we have yet another working implementation for (some primitives of) sets. The use of arrays supposedly enables us to get constant-time complexity. Yet we should feel at least some discomfort. After all, the constant time applied when the arrays contained only Boolean values. However, that solution was weak in two ways: it could not handle hash-collisions by non-invertible hash functions, and it required potentially enormous arrays. If we relaxed either assumption, the implementation was simply wrong, in that it was easily fooled by values that caused collisions either through hashing or through computing the remainder.

The solution we have shown above is called hash chaining, where “chain” refers to the list stored in each bucket. The benefit of hash-chaining is that insertion can still be constant-time: it takes a constant amount of time to find a bucket, and inserting can be as cheap as link. Of course, this assumes that we don’t mind duplicates; otherwise we will pay the same price we saw earlier in Representing Sets as Lists. But lookup takes time linear in the size of the bucket (which, with duplicates, could be arbitrarily larger relative to the number of distinct elements). And even if we check for duplicates, we run the risk that most or even all the elements could end up in the same bucket (e.g., suppose the elements are "Where", "Weird", "Wired", "Whine"). In that case, our sophisticated implementation reduces to the list-based representation and its complexity!

There’s an additional subtlety here. When we check membership of the string in the list of strings, we have to consider the cost of comparing each pair of strings. In the worst case, that is proportional to the length of the shorter string. Usually this is bounded by a small constant, but one can imagine settings where this is not guaranteed to be true. However, this same cost has to be borne by all set implementations; it is not a new complexity introduced here.

Thus, in theory, hash-based sets can support insertion and membership in as little as constant time, and (ignoring the cost of string comparisons) as much as linear time, where “linear” has the same caveats about duplicates as the list-based representation. In many cases—depending on the nature of the data and parameters set for the array—they can be much closer to constant time. As a result, they tend to be very popular in practice.

17.4.8 Bloom Filters Another way to improve the space and time complexity is to relax the properties we expect of the operations. Right now, set membership gives perfect answers, in that it answers `true` exactly when the element being checked was previously inserted into the set. But suppose we’re in a setting where we can accept a more relaxed notion of correctness, where membership tests can “lie” slightly in one direction or the other (but not both, because that makes the representation almost useless). Specifically, let’s say that “no means no” (i.e., if the set representation says the element isn’t present, it really isn’t) but “yes sometimes means no” (i.e., if the set

representation says an element is present, sometimes it might not be). In short, if the set says the element isn't in it, this should be guaranteed; but if the set says the element is present, it may not be. In the latter case, we either need some other—more expensive—technique to determine truth, or we might just not care.

Where is such a data structure of use? Suppose we are building a Web site that uses password-based authentication. Because many passwords have been leaked in well-publicized breaches, it is safe to assume that hackers have them and will guess them. As a result, we want to not allow users to select any of these as passwords. We could use a hash-table to reject precisely the known leaked passwords. But for efficiency, we could use this imperfect hash instead. If it says “no”, then we allow the user to use that password. But if it says “yes”, then either they are using a password that has been leaked, or they have an entirely different password that, purely by accident, has the same hash value, but no matter; we can just disallow that password as well. A related use is for filtering out malicious Web sites. The URL shortening system, bitly, uses it for this purpose.

Another example is in updating databases or memory stores. Suppose we have a database of records, which we update frequently. It is often more efficient to maintain a journal of changes: i.e., a list that sequentially records all the changes that have occurred. At some interval (say overnight), the journal is “flushed”, meaning all these changes are applied to the database proper. But that means every read operation has become highly inefficient, because it has to check the entire journal first (for updates) before accessing the database. Again, here we can use this faulty notion of a hash table: if the hash of the record locator says “no”, then the record certainly hasn't been modified and we go directly to the database; if it says “yes” then we have to check the journal.

We have already seen a simple example implementation of this idea earlier, when we used a single array, with modular arithmetic, to represent the set. When an element was not present in the array, we knew for a fact that it was definitely not present. When the array indicated an element was present, we couldn't be sure that what was present was the exact value we were looking for. To get around this uncertainty, we used chaining.

However, there is something else we could have done. Chaining costs both space (to store all the actual values) and time (to look through all the values). Suppose, instead, a bucket is only a Boolean value. This results in a slightly useful, but potentially very inaccurate, data structure; furthermore, it exhibits correlated failure tied to the modulus.

But suppose we have not only one array, but several! When an element is added to the set, it is added to each array; when checking for membership, every array is consulted. The set only answers affirmatively to membership if all the arrays do so.

Naturally, using multiple arrays offers absolutely no advantage if the arrays are all the same size: since both insertion and lookup are deterministic, all will yield the same answer. However, there is a simple antidote to this: use different array sizes. In particular, by using array sizes that are relatively prime to one another, we minimize the odds of a clash (only hashes that are the product of all the array sizes will fool the array).

This data structure, called a Bloom Filter, is a probabilistic data structure. Unlike our earlier set data structure, this one is not guaranteed to always give the right answer; but contrary to the space-time tradeoff, we save both space and time by changing the problem slightly to accept incorrect answers. If we know something about the distribution of hash values, and we have some acceptable bound of error, we can design hash table sizes so that with high probability, the Bloom Filter will lie within the acceptable error bounds.

17.4.9 Generalizing from Sets to Key-Values Above, we focused on sets: that is, a string effectively mapped to a Boolean value, indicating whether it was present or not. However, there are many settings where it is valuable to associate one value with another. For instance, given an identity number we might want to pull up a person's records; given a computer's name, we might want to retrieve its routing information; given a star's catalog entry, we might want its astronomical information. This kind of data structure is so ubiquitous that it has several names, some of which are more general and some implying specific implementations: key-value store, associative array, hash map, dictionary, etc.

In general, the names “key-value” and “dictionary” are useful because they suggest a behavioral interface. In contrast, associative array implies the use of arrays, and hash table suggests the use of an array (and of hashing). In fact, real systems use a variety of implementation strategies, including balanced binary search trees. The names “key-value” and “dictionary” avoid commitment to a particular implementation. Here, too, “dictionary” evokes a common mental image of unique words that map to descriptions. The term “key value” is even more technically useful because keys are meant to all be distinct (i.e., no two different key-value pairs can have the same key; alternatively, one key can

map to only one value). This makes sense because we view this as a generalization of sets, so the keys are the set elements, which must necessarily have no duplicates; the values take the place of the Boolean.

To extend our set representation to handle a dictionary or key-value store, we need to make a few changes. First, we introduce the key-value representation:

```
data KV<T>: kv(key :: String, value :: T) end
```

Each bucket is still an empty list, but we understand it to be a list of key-value pairs.

Previously, we only had `is-in` to check whether an element was present in a set or not. That element is now the key, and we could have a similar function to check whether the key is present. However, we rarely want to know just that; in fact, because we already know the key, we usually want the associated value.

Therefore, we can just have this one function:

```
getkv :: <T> String -> T
```

Of course, `getkv` may fail: the key may not be present. That is, it has become a partial function [Partial Domains]. We therefore have all the usual strategies for dealing with partial functions. Here, for simplicity we choose to return an error if the key is not present, but all the other strategies we discuss for handling partiality are valid (and often better in a robust implementation).

Similarly, we have:

```
putkv :: <T> String, T -> Nothing
```

This is the generalization of `insert`. However, `insert` had no reason to return an error: inserting an element twice was harmless. However, because keys must now be associated with only one value, insertion has to check whether the key is already present, and signal an error otherwise. In short, it is also partial. This is not partial due to a mathematical reason, but rather because of state: the same key may have been inserted previously.

Once we have agreed on this interface, getting a value is a natural extension of checking for membership:

```
fun getkv(k):
    b = index-of(k)
    r = find({(kvp): kvp.key == k}, buckets.get-now(b))
    cases (Option) r:
        | none => raise("getkv can't find " + k)
        | some(v) => v.value
    end
end
```

Having found the index, we look in the bucket for whether any key-value pair has the desired key. If it does, then we return the corresponding value. Otherwise, we error.

Inserting a key-value pair similarly generalizes adding an element to the set:

```
fun putkv(k, v):
    b = index-of(k)
    keys = map(_.key, buckets.get-now(b))
    if member(keys, k):
        raise("putkv already has a value for key " + k)
    else:
        buckets.set-now(b, link(kv(k, v), buckets.get-now(b)))
    end
end
```

Once again, we check the bucket for whether the key is already present. If it is, we choose to halt with an error. Otherwise, we make the key-value pair and link it to the existing bucket contents, and modify the array to refer to the new list.

Exercise

Do the above pair of functions do all the necessary error-checking?

This concludes our brief tour of sets (yet again!) and key-value stores or dictionaries. We have chosen to implement both using arrays, which required us to employ hashes. For more on string dictionaries, see the Pyret documentation. Observe that Pyret offers two kinds of dictionaries: one mutable (like we have shown here) and one (the default) functional.

contents ← prev up next →

17.5 Equality, Ordering, and Hashing

17.5 Equality, Ordering, and Hashing

17.5.1 Converting Values to Ordered Values In Making Sets Grow on Trees, we noted that a single comparison needs to eliminate an entire set of values. With numbers, we were able to accomplish that easily: every bigger or smaller number was excluded by a comparison. But what if the data in the set are not actually numbers? Then we have to convert an arbitrary datum into a datatype that permits such comparison. This is known as hashing.

A hash function consumes an arbitrary value and produces a comparable representation of it (its hash)—most commonly (but not strictly necessarily), a number. A hash function must naturally be deterministic: a fixed value should always yield the same hash (otherwise, we might conclude that an element in the set is not actually in it, etc.). Particular uses may need additional properties, as we discuss in Equality and Ordering.

Let us now consider how one can compute hashes. If the input datatype is a number, it can serve as its own hash. Comparison simply uses numeric comparison (e.g., $<$). Then, transitivity of $<$ ensures that if an element \set{A} is less than another element \set{B} , then \set{A} is also less than all the other elements bigger than \set{B} .

Suppose instead the input is a string. We can of course use the principle above for strings: e.g., replacing number inequality with string inequality. Strings have a lexicographic (or “alphabetic”) ordering that permit them to be treated similar to numbers.

But what if we are handed more complex datatypes?

Before we answer that, consider that in practice numbers are more efficient to compare than strings (since comparing two numbers is very nearly constant time). Thus, although we could use strings directly, it may be convenient to find a numeric representation of strings. We convert each character of the string into a number, e.g., using its code point. Based on that, here are two different hash functions:

1. Consider a list of primes as long as the string. Raise each prime by the corresponding number, and multiply the result. For instance, if the string is represented by the character codes [6, 4, 5] (the first character has code 6, the second one 4, and the third 5), we get the hash

```
num-expt(2, 6) * num-expt(3, 4) * num-expt(5, 5)
```

or 16200000.

2. Simply add together all the character codes. For the above example, this would correspond to the has

```
6 + 4 + 5
```

or 15.

The first representation is invertible, using the Fundamental Theorem of Arithmetic: given the resulting number, we can reconstruct the input unambiguously (i.e., 16200000 can only map to the input above, and none other). This is also known as the Gödel encoding. This is computationally expensive. The second encoding is, of course, not invertible (e.g., simply permute the characters and, by commutativity, the sum will be the same), but computationally much cheaper. It is also easy to implement:

```
fun hash-of(s :: String):
  fold({{a :: Number, b :: Number}: a + b},
    0,
    string-to-code-points(s))
end
```

```

check:
hash-of("Hello") is 500
hash-of("World!") is 553
hash-of("Attention! ;Danger! ") is 195692
end

```

Now let us consider more general datatypes. The principle of hashing will be similar. If we have a datatype with several variants, we can order the variants lexicographically, and use a numeric tag to represent the variants, and recursively encode the datum and the variant tag. For each field of a record, we need an ordering of the fields—the lexicographic ordering of the field names suffices—and must hash their contents recursively; having done so, we get in effect a string of numbers, which we have shown how to handle.

The critical thing to remember is that we don't actually need a meaningful operation. Observe that Gödel encodings are not “meaningful”, either. We don't actually care if a hash function concludes that the hash of 4 is less than the hash of 3! All we need is a function that is

- non-trivial: not everything should be equal; and
- deterministic: every time we ask for a hash, we should get the same answer.

Exercise

Why do we care about these two properties? Think about what would go wrong if each one was violated.

17.5.2 Hashing in Practice In practice, programmers do not want hash functions to do what we have described above. While Gödel encoding is extremely expensive, even computing `hash-of` takes time linear in the size of a string, which can get quite expensive if strings are large or we compute hashes often or both.

Instead, many programming languages do something very pragmatic. They need a value that can be compared for equality and ordering [Equality and Ordering]. Integers, we've already seen, already fit this bill very nicely. But how to obtain an integer out of arbitrary values, even datatype instances, quickly?

Simple: They just use the memory address of the datum. Every value has a memory address, and the language can obtain it in constant time by looking up the directory. Granted, these values may be allocated anywhere with respect to each other, but that's okay—we only want consistency, not “meaningfulness”.

In practice, however, things are not quite so simple. For instance, suppose we want two structurally equivalent values to have the same hash. If they are allocated in different addresses, they will hash differently. Therefore, many languages that use such a strategy also allow programmers to write their own hashing functions, often to work in conjunction with this built-in notion of hashing. These end up looking not too different from the hashing strategies we described above. Therefore, some of that complexity is inescapable, especially if a programmer wants structural rather than reference equality—which they very often do.

In the rest of this material, we will therefore continue with the simple hash function above, for multiple reasons. First, it is sufficient to illustrate how hashing works. Second, in practice, when built-in hashing does not suffice, we do write (more complex versions of) functions like the above. And finally, because it's all laid bare, it's easy for us to experiment with.

17.5.3 Equality and Ordering What we've seen [A Fine Balance: Tree Surgery] for the construction of balanced binary search trees is that we need some way of putting elements in order. In the examples we used numbers because they're a very friendly datatype: they have several properties that we take for granted. However, not all data have these properties.

The critical property that numbers have is that they are orderable. This follows because they are comparable, and the comparison is ternary: it produces three answers, “less than”, “equal to”, and “greater than”.

However, not all data have this property. What are data that might not have these properties? Actually, there are multiple possible properties here: Is something orderable? Is something even comparable?

Orderable

Numbers

Yes (but not Roughnums!)

Yes

Booleans

Yes

Yes

Data instances

Yes

Not by default

Roughnums

No

Yes

Functions

Not really

No

So...life is complicated.

That means you could potentially misuse a BBST on the wrong kind of data. Ideally, we would want to know if we're doing this. In Pyret's type system we chose not to build this in, but in some languages, the type system actually lets you capture these properties.

In Haskell, for instance, there's a mechanism called the type-class; in Java, there are interfaces. They aren't really the same, but they're useful to conflate for our purposes. Only things that meet a particular interface or type class provide certain operations. For instance, in Haskell, if you want to use `==` or `/=` (not equal), you have to be in the `Eq` type-class. Thus the comparable datatypes above would be part of `Eq`. Similarly, there's a type-class `Ord`, which ensures the availability of (and requires the implementation of) operations like `<`, `>`, `<=`, and `>=`. In Haskell, everything that is `Ord` must also be `Eq`, i.e., `Eq` is weaker than `Ord` (things can be `Eq` without being `Ord`). Pyret's Roughnums contradict that...but Haskell is okay with it. But if you try to compare two functions in Haskell,

$(\lambda x \rightarrow x + 1) < (\lambda x \rightarrow x)$

you get an error like

* No instance for (Ord (Integer -> Integer))

arising from a use of ‘<’

contents ← prev up next →

17.6 Sets as a Case Study

17.6 Sets as a Case Study We have spent a lot of time on sets. That is not only because they are useful in their own right, but also because they offer a window into a variety of possible designs. In particular, they illustrate several tradeoffs that we can make in the design of data structures, based on our needs.

There are several dimensions along which we can divide our designs.

17.6.1 Nature of the Data If the data cannot even be comparable for quality, then we can't construct sets out of them, because equality is central to the definition of a set.

If the data can be compared for equality but not for ordering, then we can only construct list-sets [Representing Sets as Lists], with their linear-time complexity. However, if we can hash the values [Converting Values to Ordered Values], then we can construct trees [Making Sets Grow on Trees] and hashtables [Sets from Hashing and Arrays]. Trees give us logarithmic complexity for the most expensive atomic operations, while hashtables give us constant-to-linear complexity.

17.6.2 Nature of the Operations Another dimension of variation is the collection of operations we need. We began with a fairly ambitious, but standard, collection of operations [<set-operations>], but gradually ignored many of them. In particular, some interpretations of sets, like Union-Find, achieve excellent complexity at the cost of most of these operations. Bloom Filters provide another instance of this. There is a general computer science principle at work here: the fewer operations we need to support, the better we can (sometimes) make the complexity of the remaining ones.

17.6.3 Nature of the Guarantee Most subtly, there was another distinction: whether or not we needed reliable results. Most of our set representations are reliable. However, we also saw one situation [Bloom Filters] where we intentionally abandoned complete reliability, replacing it with a statistical guarantee. In return, this gave us (potentially) much higher performance.

Thus, sets provide a useful microcosm of computer science itself.

contents ← prev up next →

18 State and Equality

18 State and Equality

In State, Change, and Testing, we introduced the notion of mutable data. We also saw the impact it has on testing. Underlying testing is some notion of equality: when we write a test in Pyret using `is`, we are implicitly making a statement about equality between the two sides. Here we will examine equality in the presence of state in more detail.

18.1 Boxes: A Canonical Mutable Structure In State, Change, and Testing we saw a motivating example using bank accounts. To focus our study of equality, it can be convenient to have an even simpler mutable data structure, called a box (which you will find in other programming languages as well). A box has only one field—the value being boxed—and supports just three operations:

1. `box` consumes a value and creates a mutable box containing that value.
2. `unbox-now` consumes a box and returns the value contained in the box.
3. `set-box-now` consumes a box, a new value, and changes the box to contain the value. All subsequent `unbox-nows` of that box will now return the new value—unless it is mutated again.

Here are the corresponding definitions in Pyret:

```
data Box<T>:  
    | box(ref v :: T)  
end  
  
fun unbox-now<T>(b :: Box<T>) -> T:  
    b!v  
end  
  
fun set-box-now<T>(b :: Box<T>, new-v :: T) -> Box<T>:  
    b!{v: new-v}  
end
```

Observe that we use `b!v` to extract the current value, and use the naming convention of `-now` to make clear these are stateful operations, so the value now may not be the same as the value later.

18.2 Mutation and Types In terms of types, whenever we replace the value in a box, we want it to be type-consistent with what was previously there. Otherwise it would be very difficult to program against a box, because the type of its content would keep changing.

These definitions obey the following tests:

```
check:  
    n1 = box(1)  
    n2 = box(2)  
    set-box-now(n1, 3)  
    set-box-now(n2, 4)  
    unbox-now(n1) is 3  
    unbox-now(n2) is 4  
end
```

However, we cannot write `set-box-now(n1, "hi")`, because that would violate the type of `n1`, which is `Box<Number>`. We could make this explicit by writing

```
n1 :: Box<Number> = box(1)
```

if we wanted to be explicit. However, note that `n1` being a box of numbers does not preclude us from having a box of strings:

```
n3 :: Box<String> = box("hello")
```

or indeed a box of any other type. We just need its type to remain consistent, whatever that type is.

This is a general rule we want to follow with mutable data: the new value must be the same type as the old value. This gives programs a consistent interface to program against. For instance, above, we know that we can always perform numeric operations against the value extracted from `n1`—there is no danger that it will suddenly produce a string. This discipline can either be enforced by a system of annotations, or has to be manually maintained by the programmer.

18.3 Mutation and Equality We've already seen [Re-Examining Equality] that equality is subtle. It's about to become much subtler with the introduction of mutation!

As a running example, we'll work with:

```
<three-boxes> ::=
```

```
b1 = box(7)  
b2 = box(7)  
b3 = b1
```

Observe that `b1` and `b3` are referring to the same box, while `b2` is referring to a different one. We can see this from a memory diagram:

Directory

- `b1`
→ 1001
- `b2`
→ 1002
- `b3`
→ 1001

Heap

- 1001: `box(7)`
- 1002: `box(7)`

We can confirm this using the following tests:

```
check:  
  b1 is-not%(identical) b2  
  b1 is%(identical) b3  
  b2 is-not%(identical) b3  
end
```

In other words, `b1` and `b3` are aliases for the same box, but neither is an alias to the box referred to by `b2`. Since `identical` is transitive, it follows from the first two tests that the third test must also pass (and thankfully, Pyret confirms this for us!).

Now, you might wonder why we have used `identical` and not `equal-always` [Notations for Equality], i.e., plain old `is`.

Do Now!

Let's try that:

```
check:  
  b1 is b3  
  b1 is b2  
end
```

What do you see?

It's unsurprising that the first test, `b1 is b3`, passes. However, the second, `b1 is b2`, fails! And the name suggests why: the two are not guaranteed to always be equal. That is, suppose we were to modify the box referred to by `b1`:

```
set-box-now(b1, 8)
```

Sure enough, the values in the boxes are not the same, but because `b1` and `b3` are aliases, their values change in lock-step (more accurately, there is only one value—the box at 1001):

```
check:  
  unbox-now(b1) is-not unbox-now(b2)  
  unbox-now(b1) is unbox-now(b3)  
end
```

18.4 Another Equality Predicate Suppose we return to the state where we have defined the three boxes [`<three-boxes>`] but not mutated `b1`. That is, when printed, all three boxes have the same value, `box(7)`. We have seen that `b1` and `b3` are both `equal-always` and `identical` to each other. However, we have also seen that `b1` and `b2` are neither of those. This is somewhat frustrating, because there is clearly some sense in which they are “equal”: at the moment, they contain the same value, even if later on one of them might not.

Therefore, Pyret offers a third equality predicate that is designed for just these situations: it is (as you might guess) called `equal-now`:

```
check:  
  b1 is%(equal-now) b2  
  b2 is%(equal-now) b3  
end
```

The `-now` in the name reminds us that these values are equal at the moment, but may not be equal later. Sure enough, if we add

```
set-box-now(b1, 8)
```

back into the program, the above `equal-now` tests fail: now, they are no longer equal!

Recall that the other two equality predicates have an binary operator notation: `==` for `equal-always` and `<=>` for `identical`. Similarly, `equal-now` has the binary operator `=~`. You should view that as `=` with hand-waving `~`: it's equal for now, but don't expect it to remain so. That is, we can rewrite the above tests as:

```
check:  
  equal-now(b1, b2) is true  
  (b2 =~ b3) is true  
end
```

Whether they pass, of course, depends on the state of the program: whether `b1`, `b2`, or `b3` has had its content modified.

18.5 A Hierarchy of Equality As you might guess, the equality operators have a hierarchy of implication. That is, if one operator is true of two expressions, the other necessarily is, but not vice versa.

Do Now!

Can you work out this hierarchy of implication?

Observe that if two expressions are `identical`, then they are aliases, i.e., they are referring to one and the same value. Therefore, the values produced by those expressions must be `equal-always`. If they are always equal, then clearly at any given moment, they must also be `equal-now`.

Even if two expressions are not `identical`, they may be `equal-always`. This would never be true of mutable data (because there is the possibility of a future mutation), but it can be true of immutable data that have the same structure and contents. In that case, if they are always equal, then again they must be `equal-now`.

However, the converses are not true.

If two data are `equal-now`, they may not be `equal-always`: if they are mutable, a future mutation may change the equality, as we have seen above. Similarly, two data may be `equal-always` but not be `identical`, because they reside at different heap addresses and are therefore truly different data.

In most languages, it is common to have two equality operators, corresponding to `identical` (known as reference equality) and `equal-now` (known as structural equality). Pyret is rare in having a third operator, `equal-always`. For most programs, this is in fact the most useful equality operator: it is not overly bothered with details of aliasing, which can be difficult to predict; at the same time it makes decisions that stand the test of time, thereby forming a useful basis for various optimizations (which may not even be conscious of their temporal assumptions). This is why `is` in testing uses `equal-always` by default, and forces users to explicitly pick a different primitive if they want it.

18.6 Space and Time Complexity `identical` always takes constant time. Indeed, some programs use `identical` precisely because they want constant-time equality, carefully structuring their program so that values that should be considered equal are aliases to the same value. Of course, maintaining this programming discipline is tricky.

`equal-always` and `equal-now` both must traverse at least the immutable part of data. Therefore, they take time proportional to the smaller datum (because if the two data are of different size, they must not be equal anyway, so there is no need to visit the extra data). The difference is that `equal-always` reduces to `identical` at references, thereby performing less computation than `equal-now` would.

18.7 What it Means to be Identical Return for a moment to the state where we have just defined the three boxes [`<three-boxes>`]. We could have written the following:

```
hold-b1-value = unbox-now(b1)
set-box-now(b1, hold-b1-value + 1)
```

Now, we can compare the contents of the various boxes:

```
b1-id-b2 = unbox-now(b1) == unbox-now(b2)
b1-id-b3 = unbox-now(b1) == unbox-now(b3)
```

And at the end of performing comparisons, we can restore them:

```
set-box-now(b1, hold-b1-value)
```

Observe that `b1-id-b2` would be `false` but `b1-id-b3` would be `true`. And notice that this would always be true when the two expressions are identical, but not otherwise.

Thus, at the end there has been no change, but by making the change we can check which values are and aren't aliases of others. In other words, this represents the essence of `identical`.

In practice, `identical` does not behave this way: it would be too disruptive. It is also not the most efficient implementation possible, when Pyret can simply check the memory addresses being the same. Nevertheless, it does demonstrate the basic idea behind `identical`: two values are `identical` precisely when, when you make changes to one, you see the changes manifest on the "other" (i.e., there is really only one value, but with potentially multiple names for it).

18.8 Comparing Functions We haven't actually provided the full truth about equality because we haven't discussed functions. Defining equality for functions—especially extensional equality, namely whether two functions have the same graph, i.e., for each input produce the same output—is complicated (a euphemism for impossible) due to the Halting Problem.

Because of this, most languages have tended to use approximations for function equality, most commonly reference equality. This is, however, a very weak approximation: even if the exact same function text in the same environment is allocated as two different closures, these would not be reference-equal. At least when this is done as part of the definition of `identical`, it makes sense; if other operators do this, however, they are actively lying, which is something the equality operators do not usually do.

There is one other approach we can take: simply disallow function comparison. This is what Pyret does: all three equality operators above will result in an error if you try to compare two functions. (You can compare against just one function, however, and you will get the answer `false`.) This ensures that the language's comparison operators are never trusted falsely.

Pyret did have the choice of allowing reference equality for functions inside `identical` and erroring only in the other two cases. Had it done so, however, it would have violated the chain of implication above [A Hierarchy of Equality]. The present design is arguably more elegant. Programmers who do want to use reference equality on functions can simply embed the functions inside a mutable structure like boxes.

There is one problem with erroring when comparing two functions: a completely generic procedure that compares two arbitrary values may error if both of the values given are functions. Because this can cause unpredictable program failure, Pyret offers a three-valued version of each of the above three operators (`identical3`, `equal-always3` and `equal-now3`), all of which return `EqualityResult` values that correspond to truth, falsity, and ignorance (returned in the case when both arguments are functions). Programmers can use this in place of the Boolean-valued comparison operators if they are uncertain about the types of the parameters.

contents ← prev up next →

19 Recursion and Cycles from Mutation

19 Recursion and Cycles from Mutation

Earlier [From Acyclicity to Cycles], we saw the difficulty of constructing cyclic data, and saw how we could address this problem using state [Cyclic Data]. Let us now return to the earlier example of creating a cyclic list of alternating colors. We had tried to write:

```
web-colors = link("white", link("grey", web-colors))
```

which, as we noted, does not pass muster because `web-colors` is not bound on the right of the `=`. (Why not? Because otherwise, if we try to substitute `web-colors` on the right, we would end up in an infinite regress.)

Something about this should make you a little suspicious: we have been able to write recursive functions all the time, without difficulty. Why are they different? For two reasons:

- The first reason is the fact that we're defining a function. A function's body is not evaluated right away—only when we apply it—so the language can wait for the body to finish being defined. (We'll see what this might mean in a moment.)
- The second reason isn't actually a reason: function definitions actually are special. But we are about to expose what's so special about them—it's the use of a box! [Boxes: A Canonical Mutable Structure]—so that any definition can avail of it.

Returning to our example above, recall that we can't make up our list using `links`, because we want the list to never terminate. Therefore, let us first define a new datatype to hold an cyclic list:

```
data Pair: p(hd, t1) end
```

You should think of this as analogous to a list, where `hd` is the first element and `t1` is the rest.

Observe that we have carefully avoided writing type definitions for the fields; we will instead try to figure them out as we go along. Also, however, this definition as written cannot work.

Do Now!

Do you see why not?

Let's decompose the intended infinite list into two pieces: lists that begin with white and ones that begin with grey. What follows white? A grey list. What follows grey? A white list. It is clear we can't write down these two definitions because one of them must precede the other, but each one depends on the other. (This is the same problem as trying to write a single definition above.)

19.1 Partial Definitions What we need to instead do is to partially define each list, and then complete the definition using the other one. However, that is impossible using the above definition, because we cannot change anything once it is constructed. Instead, therefore, we need:

```
data Pair: p(hd, ref t1) end
```

Note that this datatype lacks a base case, which should remind you of definitions we saw in Streams From Functions.

Using this, we can define:

```
white-pair = p("white", "dummy")
grey-pair = p("grey", "dummy")
```

Each of these definitions is quite useless by itself, but they each represent what we want, and they have a mutable field for the rest, currently holding a dummy value. Therefore, it's clear what we must do next: update the mutable field.

```
white-pair!{tl: grey-pair}
grey-pair!{tl: white-pair}
```

Because we have ordained that our colors must alternate beginning with white, this rounds up our definition:

```
web-colors = white-pair
```

If we ask Pyret to inspect the value of `web-colors`, we notice that it employs an algorithm to prevent traversing infinite objects. You can learn more about how that works separately [Detecting Cycles].

We can define a helper function, `take`, a variation of which we saw for streams [Streams From Functions], to inspect a finite prefix of an infinite list:

```
fun ctake(n :: Number, il :: Pair) -> List:
    if n == 0:
        empty
    else:
        link(il.hd, ctake(n - 1, il!tl))
    end
end
```

such that:

```
check:
    ctake(4, web-colors) is
        [list: "white", "grey", "white", "grey"]
end
```

19.2 Recursive Functions Based on this, we can now understand recursive functions. Consider a very simple example, such as this:

```
fun sum(n):
    if n > 0:
        n + sum(n - 1)
    else:
        0
    end
end
```

We might like to think this is equivalent to:

```
sum =
    lam(n):
        if n > 0:
            n + sum(n - 1)
        else:
            0
        end
    end
```

but if you enter this, Pyret will complain that `sum` is not bound. We must instead write

```
rec sum =
    lam(n):
        if n > 0:
            n + sum(n - 1)
        else:
            0
        end
```

```
end
```

What do you think `rec` does? It binds `sum` to a box initially containing a dummy value; it then defines the function in an environment where the name is bound, unboxing the use of the name; and finally, it replaces the box's content with the defined function, following the same pattern we saw earlier for `web-colors`.

19.3 Premature Evaluation Observe that the above description reveals that there is a time between the creation of the name and the assignment of a value to it. Can this intermediate state be observed? It sure can!

There are generally three solutions to this problem:

1. Make sure the value is sufficiently obscure so that it can never be used in a meaningful context. This means values like 0 are especially bad, and indeed most common datatypes should be shunned. Indeed, there is no value already in use that can be used here that might not be confusing in some context.
2. The language might create a new type of value just for use here. For instance, imagine this definition of `CList`:

```
data CList:  
| undef  
| clink(v, ref r)  
end
```

`undef` appears to be a “base case”, thus making `CList` very similar to `List`. In truth, however, the `undef` is present only until the first mutation happens, after which it will never again be present: the intent is that `r` only contain a reference to other `clinks`.

The `undef` value can now be used by the language to check for premature uses of a cyclic list. However, while this is technically feasible, it imposes a run-time penalty. Therefore, this check is usually only performed by languages focused on teaching; professional programmers are assumed to be able to manage the consequences of such premature use by themselves.

3. Allow the recursion constructor to be used only in the case of binding functions, and then make sure that the right-hand side of the binding is syntactically a function. This solution precludes some reasonable programs, but is certainly safe.

19.4 Cyclic Lists Versus Streams The color list example above is, as we have noted, very reminiscent of stream examples. What is the relationship between the two ways of defining infinite data?

Cyclic lists have on their side simplicity. The pattern of definition used above can actually be encapsulated into a language construct, so programmers do not need to wrestle with mutable fields (as above) or thunks (as streams demand). This simplicity, however, comes at a price: cyclic lists can only represent strictly repeating data, i.e., you cannot define `nats` or `fibs` as cyclic lists. In contrast, the function abstraction in a stream makes it generative: each invocation can create a truly novel datum (such as the next natural or Fibonacci number). Therefore, it is straightforward to implement cyclic lists as streams, but not vice versa.

contents ← prev up next →

20 Detecting Cycles

20.1 A Running Example As you may have noticed, Pyret will check for and print cycles. For instance,

```
import lists as L

data Pair: p(hd, ref tl) end

p0 = p(0, 0)
p1 = p(1, 1)

p2 = p(2, 3)
p3 = p(3, 4)
p2!{tl: p3}
p3!{tl: p2}

p4 = p(4, p3)
p5 = p(5, p4)

p6 = p(6, "dummy")
p6!{tl: p6}
```

Do Now!

Sketch out the above pairs to make sure you see all the cycles.

So we have two that participate in no cyclic behavior (`p0` and `p1`), two (`p2` and `p3` that are mutually-cyclic, one (`p6`) that is a self-cycle, and two (`p4` and `p5`) that lead to a cycle.

20.2 Types As an aside, imagine we try to type-check this program. We have to provide a type for `tl`, but it's not clear what this can be: sometimes it's a `Number`, and other times it's a `Pair`. However, we might observe that if our goal is to create cyclic data, then we want `tl` to refer to a `Pair` or to nothing at all. That suggests that a useful type is:

```
data Pair: p(hd :: Number, ref tl :: Option<Pair>) end
```

so that we can write

```
p0 = p(0, none)
p1 = p(1, none)

p2 = p(2, none)
p3 = p(3, none)
p2!{tl: some(p3)}
p3!{tl: some(p2)}
p4 = p(4, some(p3))
p5 = p(5, some(p4))

p6 = p(6, none)
p6!{tl: some(p6)}
```

This works, but we have to deal with the `Option` everywhere. Since our goal is to focus on cycles, and this would become unwieldy, we ignore the typed version from now on.

20.3 A First Checker

Okay, back to the untyped version.

So let's try to figure out whether, given a Pair, it leads to a cycle. What should the type be?

```
cc :: Pair -> Boolean
```

where `cc` stands for “check cycle”.

Critically, it's important that this be a total function: i.e., it always terminates.

So let's write the most obvious solution:

```
fun cc(e):
    fun loop(cur, hist):
        if is-p(cur):
            if L.member-identical(hist, cur):
                true
            else:
                loop(cur!tl, link(cur, hist))
            end
        else:
            false
        end
    end
    loop(e, empty)
end
```

First of all, does this even terminate? It could take a while to visit all the nodes, but a cycle demands that somewhere, we revisit a node we saw before. Since we track that, we can't not terminate. Therefore, termination is guaranteed, and the function is total. Indeed, all these tests pass:

```
check:
    cc(p0) is false
    cc(p1) is false
    cc(p2) is true
    cc(p3) is true
    cc(p4) is true
    cc(p5) is true
    cc(p6) is true
end
```

As another aside, observe that we could have written these tests instead like

```
check:
    p0 violates cc
    p2 satisfies cc
end
```

which would be more concise, but that would also be misleading: it would suggest that `cc` is a desirable property, so `p2` is a “good” instance and `p0` is a “bad” one. However, `cc` is not a judgment of quality—its two responses have equal weight—so this would be confusing to a later reader.

20.4 Complexity

Now that we have determined that it terminates, we can ask for its time and space complexity. First we have to decide what we are even computing the complexity over. If the sequence is finite, then the size is clearly the size of the sequence. But if it's infinite, we don't want to traverse the “whole thing”: rather, we mean its finite part (excluding any repetition). So the meaningful measure in either case is the number of `p` nodes, i.e., the finite size. It may just be that some of these lead back to themselves, so that a naïve traversal will go on forever.

Okay, so we visit each node once. We keep track of all the nodes just in case we double back over, either until we run out of nodes or we repeat one. Therefore, the space complexity is linear in the length of the sum of the prefix (from the starting node) and the cycle. The time complexity is that but also, at each point, we have to check membership, so it's quadratic in the length of that prefix + cycle. So: linear space, quadratic time, in the size of the prefix + cycle.

Now, some degree of linear behavior is unavoidable: we clearly have to keep going until we run out or hit a cycle, so for detecting the cycle having something be linear in the size of the prefix (get it out of the way) + length of the cycle (find the cycle) seems essential. But can we improve on this complexity? It seems unlikely: by definition, how can we check for a cycle if we don't remember everything we've seen?

Our first hunch might be, "Maybe there's another space-time tradeoff!" But it's not so clear here. Our space is linear and time quadratic, so we may think we can flip those around. But the time can't be less than the space! If, for instance, we had linear time and quadratic space, that wouldn't make sense, because we'd need at least quadratic time just to fill the space. So that's not going to work so well.

Instead, the best way to improve seems to have a better lookup data structure. We'd still take linear space—as we said, linear was unavoidable (and we can't just be linear in the size of the cycle, because the whole point is we don't even know we have a cycle, much less which parts are prefix and which parts cycle)—and the time complexity would hopefully reduce from quadratic to linear-times-something-sublinear.

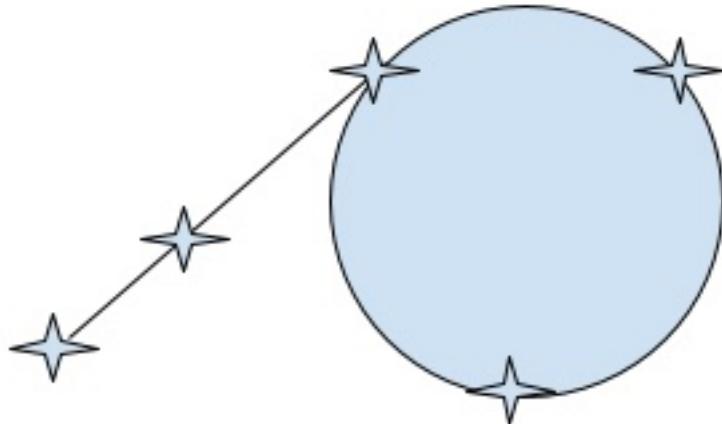
20.5 A Fabulous Improvement

It turns out we can do a lot better! It's called the tortoise-and-hare algorithm.

We start off with two references into the sequence, one called the tortoise and the other the hare.

At each step, the tortoise tries to advance by one node. If it cannot, we've run out of sequence, and we're done. The hare, being a hare and not a tortoise, tries to advance by two nodes. Again, if it cannot, we've run out of sequence, and we're done. Otherwise both advance, and check if they're at the same place. If they are, because they started out being at distinct nodes, we've found a cycle! If they are not, then we iterate.

Why does this even work? In the finite case it's clear, because the hare will run out of next nodes. We only have to think about the infinite case. There, in general, we have this kind of situation:



There is some prefix of nodes, followed by a cycle. Now, we don't know how long the prefix is, so we don't know how far ahead of the tortoise the hare is. Nevertheless, there is some first point at which the tortoise enters the cycle. (There must be, because the tortoise always makes progress, and the prefix can only be finite.) From this point on, we know that on each step, the relative speed of the two animals is 1. That means the hare "gains" 1 on the tortoise every step. We can see that eventually, the hare must catch up with the tortoise—or, alternatively, that the tortoise catches up with the hare!

Now let's analyze this. The tortoise will get caught by the time it has completed one loop of the cycle. Because the tortoise moves one step at a time, the total time is the length of the prefix + length of the loop. In terms of space, however, we no longer need any history at all; we only need the current positions of the tortoise and hare. Therefore, our time complexity is linear, but the space complexity is now significantly smaller: down to constant!

Here is the code:

```
fun th(e):
    fun loop(tt, hr):
        if tt <= hr:
            true
        else:
            if is-p(tt) and is-p(hr):
                new-tt = tt!tl
                int-hr = hr!tl
                if is-p(int-hr):
                    new-hr = int-hr!tl
                    loop(new-tt, new-hr)
                else:
                    false
            end
        else:
            false
    end
end
loop(e, e!tl)
end
```

20.6 Testing While it might be tempting to write tests like

```
check:
    cc(p0) is false
    cc(p2) is true
end
```

(i.e., the same as before, but with `cc` replaced by `ph`), we should instead write them as follows:

```
check:
    cc(p0) is th(p0)
    cc(p1) is th(p1)
    cc(p2) is th(p2)
    cc(p3) is th(p3)
    cc(p4) is th(p4)
    cc(p5) is th(p5)
    cc(p6) is th(p6)
end
```

This confers two advantages. First, if we change the example, we don't have to update two tests, only one. But the much more important reason is that we intend for `pr` to be an optimized version of `cc`. That is, we expect the two to produce the same result. We can think of `cc` as our clear, reference implementation. That is, this is another instance of model-based testing.

As an aside, this algorithm is not exactly what Pyret does, because we need to check for arbitrary graph-ness, not just cycles. It's also complicated due to user-defined functions, etc.

contents ← prev up next →

21 Avoiding Recomputation by Remembering Answers

21 Avoiding Recomputation by Remembering Answers

We have on several instances already referred to a space-time tradeoff. The most obvious tradeoff is when a computation “remembers” prior results and, instead of recomputing them, looks them up and returns the answers. This is an instance of the tradeoff because it uses space (to remember prior answers) in place of time (recomputing the answer). Let’s see how we can write such computations.

21.1 An Interesting Numeric Sequence Suppose we want to create properly-parenthesized expressions, and ignore all non-parenthetical symbols. How many ways are there of creating parenthesized expressions given a certain number of opening (equivalently, closing) parentheses?

If we have zero opening parentheses, the only expression we can create is the empty expression. If we have one opening parenthesis, the only one we can construct is “()” (there must be a closing parenthesis since we’re interested only in properly-parenthesized expressions). If we have two opening parentheses, we can construct “((())” and “()()”. Given three, we can construct “(((())”, “((())()”, “()((())”, “()()()”, and “()()()”, for a total of five. And so on. Observe that the solutions at each level use all the possible solutions at one level lower, combined in all the possible ways.

There is actually a famous mathematical sequence that corresponds to the number of such expressions, called the Catalan sequence. It has the property of growing quite large very quickly: starting from the modest origins above, the tenth Catalan number (i.e., tenth element of the Catalan sequence) is 16796. A simple recurrence formula gives us the Catalan number, which we can turn into a simple program:

```
fun catalan(n):
    if n == 0: 1
    else if n > 0:
        for fold(acc from 0, k from range(0, n)):
            acc + (catalan(k) * catalan(n - 1 - k))
        end
    end
end
```

This function’s tests look as follows—

```
<catalan-tests> ::=
check:
    catalan(0) is 1
    catalan(1) is 1
    catalan(2) is 2
    catalan(3) is 5
    catalan(4) is 14
    catalan(5) is 42
    catalan(6) is 132
    catalan(7) is 429
    catalan(8) is 1430
    catalan(9) is 4862
    catalan(10) is 16796
    catalan(11) is 58786
end
```

but beware! When we time the function's execution, we find that the first few tests run very quickly, but somewhere between a value of 10 and 20—depending on your machine and programming language implementation—you ought to see things start to slow down, first a little, then with extreme effect.

Do Now!

Check at what value you start to observe a significant slowdown on your machine. Plot the graph of running time against input size. What does this suggest?

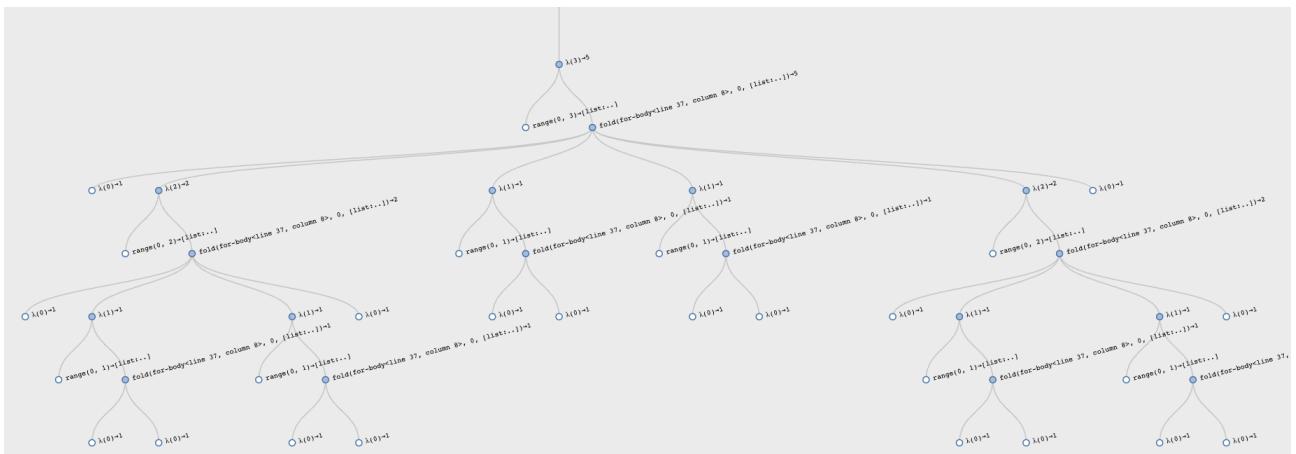
The reason the Catalan computation takes so long is precisely because of what we alluded to earlier: at each level, we depend on computing the Catalan number of all the smaller levels; this computation in turn needs the numbers of all of its smaller levels; and so on down the road.

Exercise

Map the subcomputations of `catalan` to see why the computation time explodes as it does.

What is the worst-case time complexity of this function?

Here is a graphical representation of all the sub-computations the Catalan function does for input 3:



Observe the very symmetric computation, reflecting the formula.

21.1.1 Using State to Remember Past Answers Therefore, this is clearly a case where trading space for time is likely to be of help. How do we do this? We need a notion of memory that records all previous answers and, on subsequent attempts to compute them, checks whether they are already known and, if so, just returns them instead of recomputing them.

Do Now!

What critical assumption is this based on?

Naturally, this assumes that for a given input, the answer will always be the same. As we have seen, functions with state violate this liberally, so typical stateful functions cannot utilize this optimization. Ironically, we will use state to implement this optimization, so we will have a stateful function that always returns the same answer on a given input—and thereby use state in a stateful function to simulate a stateless one. Groovy, dude!

First, then, we need some representation of memory. We can imagine several, but here's a simple one:

```
data MemoryCell:
    | mem(in, out)
end

var memory :: List<MemoryCell> = empty
```

Now how does `catalan` need to change? We have to first look for whether the value is already in `memory`; if it is, we return it without any further computation, but if it isn't, then we compute the result, store it in `memory`, and then return it:

```

fun catalan(n :: Number) -> Number:
    answer = find(lam(elt): elt.in == n end, memory)
    cases (Option) answer block:
        | none =>
            result =
                if n == 0: 1
                else if n > 0:
                    for fold(acc from 0, k from range(0, n)):
                        acc + (catalan(k) * catalan(n - 1 - k))
                    end
                end
            memory := link(mem(n, result), memory)
            result
        | some(v) => v.out
    end
end

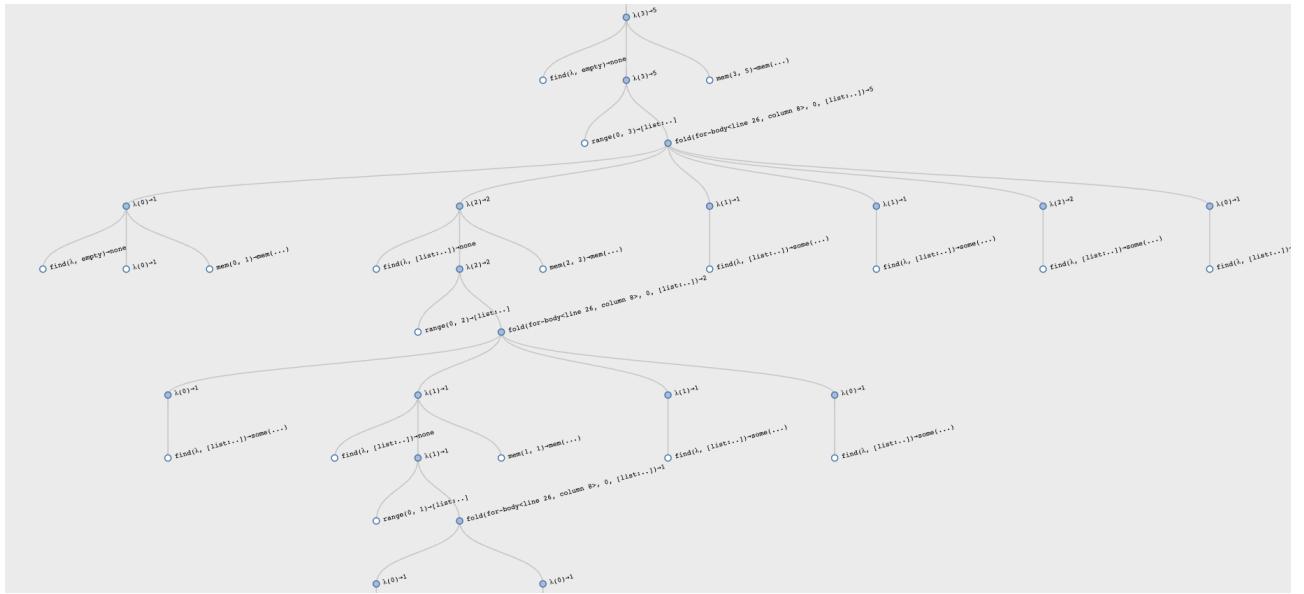
```

And that's it! Now running our previous tests will reveal that the answer computes much quicker, but in addition we can dare to run bigger computations such as `catalan(50)`.

Do Now!

Trace through a call of this revised function and see how many calls it makes.

Here is a revised visualization of computing for input 3:



Observe the asymmetric computation: the early calls perform the computations, while the latter calls simply look up the results.

This process, of converting a function into a version that remembers its past answers, is called memoization.

21.1.2 From a Tree of Computation to a DAG What we have subtly done is to convert a tree of computation into a DAG over the same computation, with equivalent calls being reused. Whereas previously each call was generating lots of recursive calls, which induced still more recursive calls, now we are reusing previous recursive calls—i.e., sharing the results computed earlier. This, in effect, points the recursive call to one that had occurred earlier. Thus, the shape of computation converts from a tree to a DAG of calls.

This has an important complexity benefit. Whereas previously we were performing a super-exponential number of calls, now we perform only one call per input and share all previous calls—thereby reducing `catalan(n)` to take a number of fresh calls proportional to n . Looking up the result of a previous call takes time proportional to the size of `memory` (because we've represented it as a list; better representations would improve on that), but that only

contributes another linear multiplicative factor, reducing the overall complexity to quadratic in the size of the input. This is a dramatic reduction in overall complexity. In contrast, other uses of memoization may result in much less dramatic improvements, turning the use of this technique into a true engineering trade-off.

21.1.3 The Complexity of Numbers As we start to run larger computations, however, we may start to notice that our computations are starting to take longer than linear growth. This is because our numbers are growing arbitrarily large—for instance, `catalan(100)` is 896519947090131496687170070074100632420837521538745909320—and computations on numbers can no longer be constant time, contrary to what we said earlier [The Size of the Input]. Indeed, when working on cryptographic problems, the fact that operations on numbers do not take constant time are absolutely critical to fundamental complexity results (and, for instance, the presumed unbreakability of contemporary cryptography). (See also Factoring Numbers.)

21.1.4 Abstracting Memoization Now we've achieved the desired complexity improvement, but there is still something unsatisfactory about the structure of our revised definition of `catalan`: the act of memoization is deeply intertwined with the definition of a Catalan number, even though these should be intellectually distinct. Let's do that next.

In effect, we want to separate our program into two parts. One part defines a general notion of memoization, while the other defines `catalan` in terms of this general notion.

What does the former mean? We want to encapsulate the idea of “memory” (since we presumably don't want this stored in a variable that any old part of the program can modify). This should result in a function that takes the input we want to check; if it is found in the memory we return that answer, otherwise we compute the answer, store it, and return it. To compute the answer, we need a function that determines how to do so. Putting together these pieces:

```
data MemoryCell:
    | mem(in, out)
end

fun memoize-1<T, U>(f :: (T -> U)) -> (T -> U):

    var memory :: List<MemoryCell> = empty

    lam(n):
        answer = find(lam(elt): elt.in == n end, memory)
        cases (Option) answer block:
            | none =>
                result = f(n)
                memory := link(mem(n, result), memory)
                result
            | some(v) => v.out
        end
    end
end
```

We use the name `memoize-1` to indicate that this is a memoizer for single-argument functions. Observe that the code above is virtually identical to what we had before, except where we had the logic of Catalan number computation, we now have the parameter `f` determining what to do.

With this, we can now define `catalan` as follows:

```
rec catalan :: (Number -> Number) =
    memoize-1(
        lam(n):
            if n == 0: 1
            else if n > 0:
                for fold(acc from 0, k from range(0, n)):
                    acc + (catalan(k) * catalan(n - 1 - k))
```

```
    end
  end
end)
```

Note several things about this definition:

1. We don't write `fun catalan(...): ...`; because the procedure bound to `catalan` is produced by `memoize-1`.
2. Note carefully that the recursive calls to `catalan` have to be to the function bound to the result of memoization, thereby behaving like an object. Failing to refer to this same shared procedure means the recursive calls will not be memoized, thereby losing the benefit of this process.
3. We need to use `rec` for reasons we saw earlier [Streams From Functions].
4. Each invocation of `memoize-1` creates a new table of stored results. Therefore the memoization of different functions will each get their own tables rather than sharing tables, which is a bad idea!

Exercise

Why is sharing memoization tables a bad idea? Be concrete.

21.2 Edit-Distance for Spelling Correction Text editors, word processors, mobile phones, and various other devices now routinely implement spelling correction or offer suggestions on (mis-)spellings. How do they do this? Doing so requires two capabilities: computing the distance between words, and finding words that are nearby according to this metric. In this section we will study the first of these questions. (For the purposes of this discussion, we will not dwell on the exact definition of what a "word" is, and just deal with strings instead. A real system would need to focus on this definition in considerable detail.)

Do Now!

Think about how you might define the "distance between two words". Does it define a metric space?

Exercise

Will the definition we give below define a metric space over the set of words?

Though there may be several legitimate ways to define distances between words, here we care about the distance in the very specific context of spelling mistakes. Given the distance measure, one use might be to compute the distance of a given word from all the words in a dictionary, and offer the closest word (i.e., the one with the least distance) as a proposed correction. Obviously, we can't compute the distance to every word in a large dictionary on every single entered word. Making this process efficient constitutes the other half of this problem. Briefly, we need to quickly discard most words as unlikely to be close enough, for which a representation such as a bag-of-words (here, a bag of characters) can greatly help. Given such an intended use, we would like at least the following to hold:

- That the distance from a word to itself be zero.
- That the distance from a word to any word other than itself be strictly positive. (Otherwise, given a word that is already in the dictionary, the "correction" might be a different dictionary word.)
- That the distance between two words be symmetric, i.e., it shouldn't matter in which order we pass arguments.

Exercise

Observe that we have not included the triangle inequality relative to the properties of a metric. Why not? If we don't need the triangle inequality, does this let us define more interesting distance functions that are not metrics?

Given a pair of words, the assumption is that we meant to type one but actually typed the other. Here, too, there are several possible definitions, but a popular one considers that there are three ways to be fat-fingered:

1. we left out a character;
2. we typed a character twice; or,
3. we typed one character when we meant another.

In particular, we are interested in the fewest edits of these forms that need to be performed to get from one word to the other. For natural reasons, this notion of distance is called the edit distance or, in honor of its creator, the Levenshtein distance. See more on Wikipedia.

There are several variations of this definition possible. For now, we will consider the simplest one, which assumes that each of these errors has equal cost. For certain input devices, we may want to assign different costs to these mistakes; we might also assign different costs depending on what wrong character was typed (two characters adjacent on a keyboard are much more likely to be a legitimate error than two that are far apart). We will return briefly to some of these considerations later [Nature as a Fat-Fingered Typist].

Under this metric, the distance between “kitten” and “sitting” is 3 because we have to replace “k” with “s”, replace “e” with “i”, and insert “g” (or symmetrically, perform the opposite replacements and delete “g”). Here are more examples:

```
<levenshtein-tests> ::=

check:
  levenshtein(empty, empty) is 0
  levenshtein([list: "x"], [list: "x"]) is 0
  levenshtein([list: "x"], [list: "y"]) is 1
# one of about 600
  levenshtein(
    [list: "b", "r", "i", "t", "n", "e", "y"],
    [list: "b", "r", "i", "t", "a", "n", "y"])
  is 3
# http://en.wikipedia.org/wiki/Levenshtein_distance
  levenshtein(
    [list: "k", "i", "t", "t", "e", "n"],
    [list: "s", "i", "t", "t", "i", "n", "g"])
  is 3
  levenshtein(
    [list: "k", "i", "t", "t", "e", "n"],
    [list: "k", "i", "t", "e", "n"])
  is 0
# http://en.wikipedia.org/wiki/Levenshtein_distance
  levenshtein(
    [list: "S", "u", "n", "d", "a", "y"],
    [list: "S", "a", "t", "u", "r", "d", "a", "y"])
  is 3
# http://www.merriampark.com/ld.htm
  levenshtein(
    [list: "g", "u", "m", "b", "o"],
    [list: "g", "a", "m", "b", "o", "l"])
  is 2
# http://www.csse.monash.edu.au/~lloyd/tildeStrings/Alignment/92.IPL.html
  levenshtein(
    [list: "a", "c", "g", "t", "a", "c", "g", "t", "a", "c", "g", "t"],
    [list: "a", "c", "a", "t", "a", "c", "t", "g", "t", "a", "c", "t"])
  is 4
  levenshtein(
    [list: "s", "u", "p", "e", "r", "c", "a", "l", "i",
     "f", "r", "a", "g", "i", "l", "i", "s", "t"],
    [list: "s", "u", "p", "e", "r", "c", "a", "l", "y",
     "f", "r", "a", "g", "i", "l", "e", "s", "t"])
  is 2
end
```

The basic algorithm is in fact very simple:

```

<levenshtein> ::=

rec levenshtein :: (List<String>, List<String> -> Number) =
    <levenshtein-body>

```

where, because there are two list inputs, there are four cases, of which two are symmetric:

```
<levenshtein-body> ::=
```

```

lam(s, t):
    <levenshtein-both-empty>
    <levenshtein-one-empty>
    <levenshtein-neither-empty>
end

```

If both inputs are empty, the answer is simple:

```
<levenshtein-both-empty> ::=

if is-empty(s) and is-empty(t): 0
```

When one is empty, then the edit distance corresponds to the length of the other, which needs to inserted (or deleted) in its entirety (so we charge a cost of one per character):

```
<levenshtein-one-empty> ::=

else if is-empty(s): t.length()
else if is-empty(t): s.length()
```

If neither is empty, then each has a first character. If they are the same, then there is no edit cost associated with this character (which we reflect by recurring on the rest of the words without adding to the edit cost). If they are not the same, however, we consider each of the possible edits:

```
<levenshtein-neither-empty> ::=

else:
    if s.first == t.first:
        levenshtein(s.rest, t.rest)
    else:
        min3(
            1 + levenshtein(s.rest, t),
            1 + levenshtein(s, t.rest),
            1 + levenshtein(s.rest, t.rest))
    end
end
```

In the first case, we assume **s** has one too many characters, so we compute the cost as if we're deleting it and finding the lowest cost for the rest of the strings (but charging one for this deletion); in the second case, we symmetrically assume **t** has one too many; and in the third case, we assume one character got replaced by another, so we charge one but consider the rest of both words (e.g., assume "s" was typed for "k" and continue with "itten" and "itting"). This uses the following helper function:

```
fun min3(a :: Number, b :: Number, c :: Number):
    num-min(a, num-min(b, c))
end
```

This algorithm will indeed pass all the tests we have written above, but with a problem: the running time grows exponentially. That is because, each time we find a mismatch, we recur on three subproblems. In principle, therefore, the algorithm takes time proportional to three to the power of the length of the shorter word. In practice, any prefix that matches causes no branching, so it is mismatches that incur branching (thus, confirming that the distance of a word with itself is zero only takes time linear in the size of the word).

Observe, however, that many of these subproblems are the same. For instance, given "kitten" and "sitting", the mismatch on the initial character will cause the algorithm to compute the distance of "itten" from "itting" but also "itten" from "sitting" and "itten" from "itting". Those latter two distance computations will also involve matching

“itten” against “itting”. Thus, again, we want the computation tree to turn into a DAG of expressions that are actually evaluated.

The solution, therefore, is naturally to memoize. First, we need a memoizer that works over two arguments rather than one:

```
data MemoryCell2<T, U, V>:
    | mem(in-1 :: T, in-2 :: U, out :: V)
end

fun memoize-2<T, U, V>(f :: (T, U -> V)) -> (T, U -> V):

    var memory :: List<MemoryCell2<T, U, V>> = empty

    lam(p, q):
        answer = find(
            lam(elt): (elt.in-1 == p) and (elt.in-2 == q) end,
            memory)
        cases (Option) answer block:
            | none =>
                result = f(p, q)
                memory :=
                    link(mem(p, q, result), memory)
                result
            | some(v) => v.out
        end
    end
end
```

Most of the code is unchanged, except that we store two arguments rather than one, and correspondingly look up both.

With this, we can redefine `levenshtein` to use memoization:

```
<levenshtein-memo> ::=

rec levenshtein :: (List<String>, List<String> -> Number) =
    memoize-2(
        lam(s, t):
            if is-empty(s) and is-empty(t): 0
            else if is-empty(s): t.length()
            else if is-empty(t): s.length()
            else:
                if s.first == t.first:
                    levenshtein(s.rest, t.rest)
                else:
                    min3(
                        1 + levenshtein(s.rest, t),
                        1 + levenshtein(s, t.rest),
                        1 + levenshtein(s.rest, t.rest))
            end
        end
    end)
```

where the argument to `memoize-2` is precisely what we saw earlier as `<levenshtein-body>` (and now you know why we defined `levenshtein` slightly oddly, not using `fun`).

The complexity of this algorithm is still non-trivial. First, let’s introduce the term suffix: the suffix of a string is the rest of the string starting from any point in the string. (Thus “kitten”, “itten”, “ten”, “n”, and “” are all suffixes of “kitten”.) Now, observe that in the worst case, starting with every suffix in the first word, we may need to perform a

comparison against every suffix in the second word. Fortunately, for each of these suffixes we perform a constant computation relative to the recursion. Therefore, the overall time complexity of computing the distance between strings of length $\lfloor m \rfloor$ and $\lfloor n \rfloor$ is $\lfloor O([m, n] \rightarrow m \cdot n) \rfloor$. (We will return to space consumption later [Contrasting Memoization and Dynamic Programming].)

Exercise

Modify the above algorithm to produce an actual (optimal) sequence of edit operations. This is sometimes known as the traceback.

21.3 Nature as a Fat-Fingered Typist We have talked about how to address mistakes made by humans. However, humans are not the only bad typists: nature is one, too!

When studying living matter we obtain sequences of amino acids and other such chemicals that comprise molecules, such as DNA, that hold important and potentially determinative information about the organism. These sequences consist of similar fragments that we wish to identify because they represent relationships in the organism's behavior or evolution. This section may need to be skipped in some states and countries. Unfortunately, these sequences are never identical: like all low-level programmers, nature slips up and sometimes makes mistakes in copying (called—wait for it—mutations). Therefore, looking for strict equality would rule out too many sequences that are almost certainly equivalent. Instead, we must perform an alignment step to find these equivalent sequences. As you might have guessed, this process is very much a process of computing an edit distance, and using some threshold to determine whether the edit is small enough. To be precise, we are performing local sequence alignment. This algorithm is named, after its creators, Smith-Waterman, and because it is essentially identical, has the same complexity as the Levenshtein algorithm.

The only difference between traditional presentations of Levenshtein and Smith-Waterman is something we alluded to earlier: why is every edit given a distance of one? Instead, in the Smith-Waterman presentation, we assume that we have a function that gives us the gap score, i.e., the value to assign every character's alignment, i.e., scores for both matches and edits, with scores driven by biological considerations. Of course, as we have already noted, this need is not peculiar to biology; we could just as well use a “gap score” to reflect the likelihood of a substitution based on keyboard characteristics.

21.4 Dynamic Programming We have used memoization as our canonical means of saving the values of past computations to reuse later. There is another popular technique for doing this called dynamic programming. This technique is closely related to memoization; indeed, it can be viewed as the dual method for achieving the same end. First we will see dynamic programming at work, then discuss how it differs from memoization.

Dynamic programming also proceeds by building up a memory of answers, and looking them up instead of recomputing them. As such, it too is a process for turning a computation's shape from a tree to a DAG of actual calls. The key difference is that instead of starting with the largest computation and recurring to smaller ones, it starts with the smallest computations and builds outward to larger ones.

We will revisit our previous examples in light of this approach.

21.4.1 Catalan Numbers with Dynamic Programming To begin with, we need to define a data structure to hold answers. Following convention, we will use an array. What happens when we run out of space? We can use the doubling technique we studied for Halloween Analysis.

```
MAX-CAT = 11
```

```
answers :: Array<Option<Number>> = array-of(none, MAX-CAT + 1)
```

Then, the `catalan` function simply looks up the answer in this array:

```
fun catalan(n):
  cases (Option) array-get-now(answers, n):
    | none => raise("looking at uninitialized value")
    | some(v) => v
  end
end
```

But how do we fill the array? We initialize the one known value, and use the formula to compute the rest in incremental order. Because we have multiple things to do in the body, we use `block`:

```
fun fill-catalan(upper) block:
    array-set-now(answers, 0, some(1))
    when upper > 0:
        for each(n from range(1, upper + 1)):
            block:
                cat-at-n =
                    for fold(acc from 0, k from range(0, n)):
                        acc + (catalan(k) * catalan(n - 1 - k))
                    end
                array-set-now(answers, n, some(cat-at-n))
            end
        end
    end
end

fill-catalan(MAX-CAT)
```

The resulting program obeys the tests in `<catalan-tests>`.

Notice that we have had to undo the natural recursive definition—which proceeds from bigger values to smaller ones—to instead use a loop that goes from smaller values to larger ones. In principle, the program has the danger that when we apply `catalan` to some value, that index of `answers` will have not yet been initialized, resulting in an error. In fact, however, we know that because we fill all smaller indices in `answers` before computing the next larger one, we will never actually encounter this error. Note that this requires careful reasoning about our program, which we did not need to perform when using memoization because there we made precisely the recursive call we needed, which either looked up the value or computed it afresh.

21.4.2 Levenshtein Distance and Dynamic Programming Now let's take on rewriting the Levenshtein distance computation:

```
<levenshtein-dp> ::=
fun levenshtein(s1 :: List<String>, s2 :: List<String>) block:
    <levenshtein-dp/1>
end
```

We will use a table representing the edit distance for each prefix of each word. That is, we will have a two-dimensional table with as many rows as the length of `s1` and as many columns as the length of `s2`. At each position, we will record the edit distance for the prefixes of `s1` and `s2` up to the indices represented by that position in the table.

Note that index arithmetic will be a constant burden: if a word is of length $\backslash(n\backslash)$, we have to record the edit distance to its $\backslash(n + 1\backslash)$ positions, the extra one corresponding to the empty word. This will hold for both words:

```
<levenshtein-dp/1> ::=
s1-len = s1.length()
s2-len = s2.length()
answers = array2d(s1-len + 1, s2-len + 1, none)
<levenshtein-dp/2>
```

Observe that by creating `answers` inside `levenshtein`, we can determine the exact size it needs to be based on the inputs, rather than having to over-allocate or dynamically grow the array.

Exercise

Define the functions

```
array2d :: Number, Number, A -> Array<A>
set-answer :: Array<A>, Number, Number, A -> Nothing
get-answer :: Array<A>, Number, Number -> A
```

We have initialized the table with `none`, so we will get an error if we accidentally try to use an uninitialized entry. Which proved to be necessary when writing and debugging this code! It will therefore be convenient to create helper functions that let us pretend the table contains only numbers:

```
<levenshtein-dp/2> ::=
fun put(s1-idx :: Number, s2-idx :: Number, n :: Number):
    set-answer(answers, s1-idx, s2-idx, some(n))
end
fun lookup(s1-idx :: Number, s2-idx :: Number) -> Number block:
    a = get-answer(answers, s1-idx, s2-idx)
    cases (Option) a:
        | none => raise("looking at uninitialized value")
        | some(v) => v
    end
end
```

Now we have to populate the array. First, we initialize the row representing the edit distances when `s2` is empty, and the column where `s1` is empty. At $\langle((0, 0)\rangle$, the edit distance is zero; at every position thereafter, it is the distance of that position from zero, because that many characters must be added to one or deleted from the other word for the two to coincide:

```
<levenshtein-dp/3> ::=
for each(s1i from range(0, s1-len + 1)):
    put(s1i, 0, s1i)
end
for each(s2i from range(0, s2-len + 1)):
    put(0, s2i, s2i)
end
<levenshtein-dp/4>
```

Now we finally get to the heart of the computation. We need to iterate over every character in each word. These characters are at indices 0 to `s1-len - 1` and `s2-len - 1`, which are precisely the ranges of values produced by `range(0, s1-len)` and `range(0, s2-len)`.

```
<levenshtein-dp/4> ::=
for each(s1i from range(0, s1-len)):
    for each(s2i from range(0, s2-len)):
        <levenshtein-dp/compute-dist>
    end
end
<levenshtein-dp/get-result>
```

Note that we're building our way "out" from small cases to large ones, rather than starting with the large input and working our way "down", recursively, to small ones.

Do Now!

Is this strictly true?

No, it isn't. We did first fill in values for the "borders" of the table. This is because doing so in the midst of `<levenshtein-dp/compute-dist>` would be much more annoying. By initializing all the known values, we keep the core computation cleaner. But it does mean the order in which we fill in the table is fairly complex.

Now, let's return to computing the distance. For each pair of positions, we want the edit distance between the pair of words up to and including those positions. This distance is given by checking whether the characters at the pair of positions are identical. If they are, then the distance is the same as it was for the previous pair of prefixes; otherwise we have to try the three different kinds of edits:

```
<levenshtein-dp/compute-dist> ::=
dist =
```

```

if get(s1, s1i) == get(s2, s2i):
    lookup(s1i, s2i)
else:
    min3(
        1 + lookup(s1i, s2i + 1),
        1 + lookup(s1i + 1, s2i),
        1 + lookup(s1i, s2i))
end
put(s1i + 1, s2i + 1, dist)

```

As an aside, this sort of “off-by-one” coordinate arithmetic is traditional when using tabular representations, because we write code in terms of elements that are not inherently present, and therefore have to create a padded table to hold values for the boundary conditions. The alternative would be to allow the table to begin its addressing from -1 so that the main computation looks traditional.

At any rate, when this computation is done, the entire table has been filled with values. We still have to read out the answer, which lies at the end of the table:

```

<levenshtein-dp/get-result> ::=

lookup(s1-len, s2-len)

```

Even putting aside the helper functions we wrote to satiate our paranoia about using undefined values, we end up with: As of this writing, the current version of the Wikipedia page on the Levenshtein distance features a dynamic programming version that is very similar to the code above. By writing in pseudocode, it avoids address arithmetic issues (observe how the words are indexed starting from 1 instead of 0, which enables the body of the code to look more “normal”), and by initializing all elements to zero it permits subtle bugs because an uninitialized table element is indistinguishable from a legitimate entry with edit distance of zero. The page also shows the recursive solution and alludes to memoization, but does not show it in code.

```

fun levenshtein(s1 :: List<String>, s2 :: List<String>) block:
    s1-len = s1.length()
    s2-len = s2.length()
    answers = array2d(s1-len + 1, s2-len + 1, none)

    fun put(s1-idx :: Number, s2-idx :: Number, n :: Number):
        set-answer(answers, s1-idx, s2-idx, some(n))
    end

    fun lookup(s1-idx :: Number, s2-idx :: Number) -> Number block:
        a = get-answer(answers, s1-idx, s2-idx)
        cases (Option) a:
            | none => raise("looking at uninitialized value")
            | some(v) => v
        end
    end

    for each(s1i from range(0, s1-len + 1)):
        put(s1i, 0, s1i)
    end

    for each(s2i from range(0, s2-len + 1)):
        put(0, s2i, s2i)
    end

    for each(s1i from range(0, s1-len)):
        for each(s2i from range(0, s2-len)):
            dist =
                if get(s1, s1i) == get(s2, s2i):
                    lookup(s1i, s2i)
                else:

```

```

min3(
    1 + lookup(s1i, s2i + 1),
    1 + lookup(s1i + 1, s2i),
    1 + lookup(s1i, s2i))
end
put(s1i + 1, s2i + 1, dist)
end
end

lookup(s1-len, s2-len)
end

```

which is worth contrasting with the memoized version (<levenshtein-memo>). For more examples of canonical dynamic programming problems, see this page and think about how each can be expressed as a direct recursion.

21.5 Contrasting Memoization and Dynamic Programming Now that we've seen two very different techniques for avoiding recomputation, it's worth contrasting them. The important thing to note is that memoization is a much simpler technique: write the natural recursive definition; determine its time complexity; decide whether this is problematic enough to warrant a space-time trade-off; and if it is, apply memoization. The code remains clean, and subsequent readers and maintainers will be grateful for that. In contrast, dynamic programming requires a reorganization of the algorithm to work bottom-up, which can often make the code harder to follow and full of subtle invariants about boundary conditions and computation order.

That said, the dynamic programming solution can sometimes be more computationally efficient. For instance, in the Levenshtein case, observe that at each table element, we (at most) only ever use the ones that are from the previous row and column. That means we never need to store the entire table; we can retain just the fringe of the table, which reduces space to being proportional to the sum, rather than product, of the length of the words. In a computational biology setting (when using Smith-Waterman), for instance, this saving can be substantial. This optimization is essentially impossible for memoization.

In more detail, here's the contrast:

Memoization

Dynamic Programming

Top-down

Bottom-up

Depth-first

Breadth-first

Black-box

Requires code reorganization

All stored calls are necessary

May do unnecessary computation

Cannot easily get rid of unnecessary data

Can more easily get rid of unnecessary data

Can never accidentally use an uninitialized answer

Can accidentally use an uninitialized answer

Needs to check for the presence of an answer

Can be designed to not need to check for the presence of an answer

As this table should make clear, these are essentially dual approaches. What is perhaps left unstated in most dynamic programming descriptions is that it, too, is predicated on the computation always producing the same answer for a given input—i.e., being a pure function.

From a software design perspective, there are two more considerations.

First, the performance of a memoized solution can trail that of dynamic programming when the memoized solution uses a generic data structure to store the memo table, whereas a dynamic programming solution will invariably use a custom data structure (since the code needs to be rewritten against it anyway). Therefore, before switching to dynamic programming for performance reasons, it makes sense to try to create a custom memoizer for the problem: the same knowledge embodied in the dynamic programming version can often be encoded in this custom memoizer (e.g., using an array instead of list to improve access times). This way, the program can enjoy speed comparable to that of dynamic programming while retaining readability and maintainability.

Second, suppose space is an important consideration and the dynamic programming version can make use of significantly less space. Then it does make sense to employ dynamic programming instead. Does this mean the memoized version is useless?

Do Now!

What do you think? Do we still have use for the memoized version?

Yes, of course we do! It can serve as an oracle [Oracles for Testing] for the dynamic programming version, since the two are supposed to produce identical answers anyway—and the memoized version would be a much more efficient oracle than the purely recursive implementation, and can therefore be used to test the dynamic programming version on much larger inputs.

In short, always first produce the memoized version. If you need more performance, consider customizing the memoizer's data structure. If you need to also save space, and can arrive at a more space-efficient dynamic programming solution, then keep both versions around, using the former to test the latter (the person who inherits your code and needs to alter it will thank you!).

Exercise

We have characterized the fundamental difference between memoization and dynamic programming as that between top-down, depth-first and bottom-up, breadth-first computation. This should naturally raise the question, what about:

- top-down, breadth-first
- bottom-up, depth-first

orders of computation. Do they also have special names that we just happen to not know?

Are they uninteresting? Or do they not get discussed for a reason?

contents ← prev up next →

22 Partial Domains

22 Partial Domains

Sometimes, we cannot precisely capture the domain of a function with the precision we would like. In mathematics, if a function cannot accept all values in its domain, it is called partial. This is a problem we encounter more often than we might like in programming, so we need to know how to handle it. There are actually several programming strategies that we can use, with different benefits and weaknesses. Here, we will examine some of them.

Consider some functions on lists of numbers, such as computing the median or the average. In both cases, these functions don't work on all lists of numbers: there is no median for the empty list, and we can't compute its average either, because there are no elements (so trying to compute the average would result in a division-by-zero error). Thus, while it is a convenient fiction to write

```
average :: List<Number> -> Number
```

it is just that: a (bit of a) fiction. The function is only defined on non-empty lists.

We will now see how to handle this from a software engineering perspective. We'll specifically work through `average` because the function is simple enough that we can focus on the software structure without getting lost in the solution details. There are at least four solutions, and one non-solution.

22.1 A Non-Solution We will start with a strategy that has often been used by programmers in the past, but that we reject as a non-solution. This strategy is to make the above contract absolutely correct by returning a value in the erroneous case; this value is often called a sentinel. For instance, the sentinel might be 0. Here is the full program:

```
type LoN = List<Number>

fun sum(l :: LoN) -> Number:
    fold({(a, b): a + b}, 0, l)
end
```

```
avg0 :: LoN -> Number
```

```
fun avg0(l):
    cases (List) l:
        | empty => 0
        | link(_, _) =>
            s = sum(l)
            c = l.length()
            s / c
    end
end
```

and here are a few tests:

```
check:
    avg0([list: 1]) is 1
    avg0([list: 1, 2, 3]) is 2
    avg0([list: 1, 2, 3, 10]) is 4
end
```

Is there a test missing here? Yes, for the empty list! Should we add it?

```
check:  
  avg0(empty) is 0  
end
```

The question is, should we be happy with this “solution”? There are two problems with it.

First, every single use of `avg0` needs to check for whether it got back 0 or not. If it did not, then the answer is legitimate, and it can proceed with the computation. But if it did, then it has to assume that the input may have been illegitimate, and cannot use the answer.

Second, even that’s not quite true. To understand why, we need to write a few more tests:

```
check:  
  avg0([list: -1, 0, 1]) is 0  
  avg0([list: -5, 4, 1]) is 0  
end
```

So the problem is that when `avg0` returns 0, we don’t know whether that’s a legitimate answer or a “fake” answer that stands for “this is not a valid input”. So even our strategy of “check everywhere” fails!

Ah, but maybe the problem is the use of 0! Perhaps we could use a different number that would work. How about 1? Or -1? The question is: Is there any number that reasonably can’t be the average of an actual input? (And in general, for all problems, can you be sure of this?) Well, of course not.

That’s why this is a non-solution. It has created several problems:

- We can’t tell from the output whether the input was invalid.
- That means every caller needs to check.
- A caller that forgets to check may compute with nonsense.
- Compositionality is ruined: any function passed `average` needs to know to check the output (and there is nothing in the contract to warn it!).

Indeed, decades of experience tells us that some of the world’s most sophisticated programmers have not been able to handle this issue even when it matters most, resulting in numerous, pernicious security problems. Therefore, we should now regard this as a flawed approach to software construction, and never do it ourselves.

Let’s instead look at four actual solutions.

22.2 Exceptions One technique that many languages, including Pyret, provide is called the exception. An exception is a special programming construct that effectively halts the computation because the program cannot figure out how to continue computing with the data it has. There are more sophisticated forms of exceptions in some languages, but here we focus simply on using them as a strategy for handling partiality.

Here is the average program written using an exception (we reuse `sum` from before):

```
avg1 :: LoN -> Number  
  
fun avg1(l):  
  cases (List) l:  
    | empty => raise("no average for empty list")  
    | link(_, _) =>  
      s = sum(l)  
      c = l.length()  
      s / c  
  end  
end  
  
check:  
  avg1([list: 1]) is 1
```

```

avg1([list: 1, 2, 3]) is 2
avg1([list: 1, 2, 3, 10]) is 4
end

```

The way `raise` works is that it terminates everything that is waiting to happen. For instance, if we were to write `1 + avg1(empty)`

the `1 + ...` part never happens: the whole computation ends. `raise` creates exceptions.

Again, we're missing a test. How do we write it? `check: avg1(empty) raises "no average for empty list"`
`end` The `raises` form takes a string that it matches against that provided to `raise`. In fact, for convenience, any sub-string of the original string is permitted: we can, for instance, also write `check: avg1(empty) raises "no average" avg1(empty) raises "empty list" end`

In many programming languages, the use of exceptions is the standard way of dealing with partiality. It is certainly a pragmatic solution. Observe that we got to reuse `sum` from earlier; the contract looks clean; and we only needed to use `raise` at the spot where we didn't know what to do. What's not to like?

There are two main problems with exceptions:

1. In real systems, exceptions halt a program's execution in unpredictable ways. A caller to `avg1` may be half-way through doing something else (e.g., it may have opened a file that it intends to close), but the exception causes the call to not finish cleanly, causing the remaining computation to not run, leaving the system in a messy state.
2. Relatedly, what we presented as a feature should actually be treated as a problem: the contract lies! There's no indication at all in the contract that an exception might occur. A programmer has to read the whole implementation—which could change at any time—instead of being able to rely on its published contract, when the whole point of contracts was that they saved us from having to read the whole implementation!

Indeed, some modern programming languages designed for large-scale programming (such as Go and Rust) no longer have exception constructs. Therefore, you should not assume that this will continue to be the “standard” way of doing things in the future.

Observe that there are two kinds of exceptions that can occur. One is as we've written above. The other is when we completely ignore (or forget to even think about) the empty list case, and end up getting an error from Pyret, which is also a kind of exception. If Pyret will raise an exception anyway, does it make sense for us to go through the trouble of doing it ourselves?

Yes it does! For several reasons:

1. First, you get to control where the exception occurs and what it says.
2. You can document that the exception will occur.
3. You are less dependent on the behavior of Pyret or whatever underlying programming language, which can change in subtle ways.
4. You can create an exception that is unique to you, so it can't be confused with other division-by-zero errors that may lurk in your program.

For these reasons, it's better to check and raise an exception explicitly than letting it “fall through” to the programming language. Instead, the real problems with this solution are subtler: the lying contract, and the impact on program execution.

22.3 The Option Type Let's revisit `avg0`. The problem with it was that it returned a value that was not distinguishable from an actual answer. So perhaps another approach is to return a value that is guaranteed to be distinguishable! For this, a growing number of languages (including Pyret) have something like this type:

```

data Option<T>:
| none
| some(value :: T)
end

```

This is a type we use when we aren't sure we will have an answer: `none` means we don't have an answer, whereas `some` means we do and `value` is that answer.

Here's how our program now looks:

```
avg2 :: LoN -> Option<Number>
```

```
fun avg2(l):
  cases (List) l:
    | empty => none
    | link(_, _) =>
      s = sum(l)
      c = l.length()
      some(s / c)
  end
end
```

Now our tests look a bit different:

```
check:
  avg2([list: 1]) is some(1)
  avg2([list: 1, 2, 3]) is some(2)
  avg2([list: 1, 2, 3, 10]) is some(4)
end

check:
  avg2(empty) is none
end
```

The good news is, the contract is now truthful. Just by looking at it, we are reminded that `avg0` may not always be able to compute an answer.

Unfortunately, this imposes some cost on every user: they have to use `cases` to check return values and only use them if they are legitimate. However, this is the same thing we expected in `avg0`—except we lacked a discipline for making sure we didn't abuse that value! So this is `avg0` done in a principled way.

22.4 Total Domains, Dynamically All these problems arise because we said that `average` (like `median`) is partial. However, it's only partial if we give the domain as `List<Number>`; it's actually a total function on the non-empty list of numbers. But how do we represent that?

In some languages, like Pyret, we can actually express this directly:

```
type NeLoND = List<Number>%{is-link}
```

This says that we're refining numeric lists to always have a `link`, i.e., to be non-empty. In Pyret, currently, this check is only done at run-time; in some other programming languages, this can be done by the type-checker itself.

This refinement lets us pretend that we're dealing with regular lists and reuse all existing list code, while knowing for sure we will never get a divide-by-zero error:

```
avg3 :: NeLoND -> Number

fun avg3(l):
  s = sum(l)
  c = l.length()
  s / c
end

check:
  avg3([list: 1]) is 1
  avg3([list: 1, 2, 3]) is 2
  avg3([list: 1, 2, 3, 10]) is 4
```

```
end
```

If we do try passing an empty list, we get an internal exception:

```
check:  
    avg3(empty) raises ""  
end
```

This is a pretty interesting solution. Our function's code is clean. We don't deal with nonsensical values. The interface is truthful! (However, it does require a careful reading to observe that there's an exception lurking underneath the domain.) And it lets us reuse existing code.

There are two main weaknesses:

1. Dynamic refinements aren't found in most languages, so we'd have to do more manual work to obtain the same solution.
2. We don't get a static guarantee (i.e., before even running the program) that we'll never get an exception.

22.5 Total Domains, Statically How do we make the function `total` with a static guarantee? That would require that we ensure that we can never construct an empty list! Obviously, this is not possible with the existing lists in Pyret. However, we can construct a new list-like datatype that "bottoms out" not at empty lists but at lists of one element:

```
data NeLoN:  
    | one(n :: Number)  
    | more(n :: Number, r :: NeLoN)  
end
```

Observe that there is simply no way to make an empty list: the smallest list has one element in it. Furthermore, our type checker enforces this for us.

Of course, this is an entirely different datatype than a list of numbers. We can't, for instance, use the existing `sum` or `length` code on it. However, one option is to convert a `NeLoN` into a `LoN`, which is always safe, and reuse that code:

```
fun nelon-to-lon(nl :: NeLoN):  
    cases (NeLoN) nl:  
        | one(n) => [list: n]  
        | more(n, r) => link(n, nelon-to-lon(r))  
    end  
end  
  
fun nl-sum(nl :: NeLoN) -> Number:  
    sum(nelon-to-lon(nl))  
end  
  
fun nl-len(nl :: NeLoN) -> Number:  
    nelon-to-lon(nl).length()  
end
```

Now we can write the average in an interesting way:

```
fun avg4(nl :: NeLoN) -> Number:  
    s = nl-sum(nl)  
    c = nl-len(nl)  
    s / c  
end
```

Once again, we don't have to have any logic for dealing with errors. However, it's not because we're sloppy or letting Pyret deal with it or getting it checked at runtime or anything else: it's because there is no way for an empty list to arise. Thus we have both the simplest body and the most truthful interface! But it comes at a cost: we need to do some work to reuse existing functions.

This problem extends to writing tests, which is now more painful:

```
check:
  nl1 = one(1)
  nl2 = more(1, more(2, one(3)))
  nl3 = more(1, more(2, more(3, one(10)))))

  avg4(nl1) is 1
  avg4(nl2) is 2
  avg4(nl3) is 4
end
```

That is, we've lost our convenient way of writing lists. We can recover that by writing a helper that creates NeLoNs:

```
fun lon-to-nelon(l :: LoN) -> NeLoN:
  cases (List) l:
    | empty => raise("can't make an empty NeLoN")
    | link(f, r) =>
      cases (List) r:
        | empty => one(f)
        | else => more(f, lon-to-nelon(r))
      end
    end
  end
end

check:
  avg4(lon-to-nelon([list: 1])) is 1
  avg4(lon-to-nelon([list: 1, 2, 3])) is 2
  avg4(lon-to-nelon([list: 1, 2, 3, 10])) is 4
end
```

Notice that if we try to use an empty list, we get an exception:

```
check:
  avg4(lon-to-nelon(empty)) raises ""
end
```

However, it's very important to understand where the error is coming from: the exception is not from `avg4`, it's coming from `lon-to-nelon`, i.e., from the "interface" function. The bad datum never makes it as far as `avg4`! We can verify this:

```
check:
  lon-to-nelon(empty) raises ""
end
```

Remember, there's no way to send an empty list to `avg4`! Nevertheless, this suggests a trade-off: we can either use `NeLoN` explicitly but with more notational pain, or we can use `list` but run the risk of some confusion about exceptions. This is a trade-off in general, but there are better options in some languages (A Note on Notation).

So this is actually a very powerful technique: building a datatype that reflects exactly what we want, thereby turning a partial function into a total one. Programmers call this principle making illegal states unrepresentable. It may require writing some procedures to convert to and from other convenient representations for code reuse. Somewhere in those procedures there must be checks that reflect the partiality.

22.6 Summary

In general, there is one non-solution:

- Return a sentinel value. Do not ever do this unless you've first fixed all the security bugs lurking in C programs from the past several decades.

and there are four solutions:

- Use `raise`. This is not very good for software engineering in general because exceptions are clunky, semantically complicated, and not compositional.
- Use a dynamic refinement. Dynamic refinements aren't in most languages. Also, it's less good than each of the other solutions, but it's a decent compromise in many settings.
- Define a datatype to make illegal states unrepresentable. A bit of work. Pretty sophisticated, invaluable in some places, but not always worth the effort.
- Use `Option`. Often the ideal option, because:
 - The type tells us to expect funny business. (`raise` hides that.)
 - We can't accidentally misuse the value. (Sentinels hide that.)
 - It's compositional: we can create functions to help us handle it.
 - It's much lower overhead than the static totality solution.
 - It's more statically robust than the dynamic totality solution.
 - It generalizes: in practice, instead of just `none` and `some`, a real program will have `some` for the "normal" case, and a bunch of variants describing the different kinds of errors that are possible, with extra information in each case. For concrete examples of this, see Picking Elements from Sets on sets Combining Answers on queues.

22.7 A Note on Notation When we wrote above that we can't get the convenience of writing, say, `[list: 1, 2, 3]` when using NeLoNs, we were speaking in general. In some languages, we can actually make similar convenient constructors. In Pyret, for instance, there is a protocol for defining custom constructors; in fact, seemingly built-in constructors like `list` and `set` are built using this protocol. The code for doing this is a bit ungainly (in part because it's optimized to save some space and time by making the constructor-writer's life a little harder), but it only needs to be written once. Here's a `nelon` constructor for NeLoNs:

```
fun ra-to-nelon(r :: RawArray<Number>) -> NeLoN:
  len = raw-array-length(r)
  fun make-from-index(n :: Number):
    v = raw-array-get(r, n)
    if n == (len - 1):
      one(v)
    else:
      more(v, make-from-index(n + 1))
    end
  end
  make-from-index(0)
end

nelon = {
  make0: {(): raise("can't make an empty NeLoN")},
  make1: {(a1): one(a1)},
  make2: {(a1, a2): more(a1, one(a2))},
  make3: {(a1, a2, a3): more(a1, more(a2, one(a3)))},
  make4: {(a1, a2, a3, a4): more(a1, more(a2, more(a3, one(a4))))},
  make5: {(a1, a2, a3, a4, a5): more(a1, more(a2, more(a3, more(a4, one(a5)))))},
  make: {args :: RawArray<Number>: ra-to-nelon(args)} }
```

These tests show that this constructor works very much like the built-in `list`:

```
check:
[nelon: ] raises "empty"
[nelon: 1] is one(1)
[nelon: 1, 2] is more(1, one(2))
[nelon: 1, 2, 3] is more(1, more(2, one(3)))
```

```
[nelon: 1, 2, 3, 4] is more(1, more(2, more(3, one(4))))  
[nelon: 1, 2, 3, 4, 5] is more(1, more(2, more(3, more(4, one(5)))))  
[nelon: 1, 2, 3, 4, 5, 6] is  
more(1, more(2, more(3, more(4, more(5, one(6))))))  
[nelon: 1, 2, 3, 4, 5, 6, 7] is  
more(1, more(2, more(3, more(4, more(5, more(6, one(7)))))))  
end
```

With this, we can rewrite the tests from Total Domains, Statically very conveniently:

```
check:  
  avg4([nelon: 1]) is 1  
  avg4([nelon: 1, 2, 3]) is 2  
  avg4([nelon: 1, 2, 3, 10]) is 4  
end
```

thereby having our cake and eating it too!

contents ← prev up next →

23 Staging

23.1 Problem Definition Earlier, we saw a detailed development of binary trees representing ancestry [Creating a Datatype for Ancestor Trees]. In what follows we don't need a lot of detail, so we will give ourselves a simplified version of essentially the same data definition:

```
data ABT:
  | unknown
  | person(name :: String, bm :: ABT, bf :: ABT)
end
```

We can then write functions over such as this:

```
fun abt-size(p :: ABT):
  doc: "Compute the number of known people in the ancestor tree"
  cases (ABT) p:
    | unknown => 0
    | person(n, p1, p2) => 1 + abt-size(p1) + abt-size(p2)
  end
end
```

Now let's think about a slightly different function: `how-many-named`, which tells us how many people in a family have a particular name. Not only can more than one person have the same name, in some cultures it's not uncommon to use the same name across generations, either in successive generations or skipping one.

Do Now!

What is the contract for `how-many-named`? The contract for this function will be crucial, so make sure you do this step!

Here is one meaningful contract:

```
how-many-named :: ABT, String -> Number
```

It takes a tree in which to search, a name to search for, and returns a count.

Do Now!

Define `how-many-named`.

23.2 Initial Solution Presumably you ended up with something like this:

```
fun how-many-named(p :: ABT, looking-for :: String):
  cases (ABT) p:
    | unknown => 0
    | person(n, p1, p2) =>
      if n == looking-for:
        1 + how-many-named(p1, looking-for) + how-many-named(p2, looking-for)
      else:
        how-many-named(p1, looking-for) + how-many-named(p2, looking-for)
    end
  end
end
```

Let's say you have defined this person:

```

p =
  person("A",
    person("B",
      person("A", unknown, unknown),
      person("C",
        person("A", unknown, unknown),
        person("B", unknown, unknown))),
      person("C", unknown, unknown))

```

With that, we can write a test like

```

check:
  how-many-named(p, "A") is 3
end

```

23.3 Refactoring

Now let's apply some transformations, sometimes called code refactorings, to this function.

First, notice the repeated expression. What the whole conditional is essentially saying is that we want to know how much this person is contributing to the overall count; the rest of the count stays the same regardless. We can therefore write this more concisely (and, if we know how to read such code, more meaningfully) as the following instead:

```

fun how-many-named(p :: ABT, looking-for :: String):
  cases (ABT) p:
    | unknown => 0
    | person(n, p1, p2) =>
      (if n == looking-for: 1 else: 0 end)
      +
      how-many-named(p1, looking-for) + how-many-named(p2, looking-for)
  end
end

```

If you have prior programming experience, this may look a bit odd to you, but `if` is in fact an expression, which has a value; in this case the value is either 0 or 1. This value can then be used in an addition.

Now let's look at this code even more closely. Notice something interesting. We keep passing two parameters to `how-many-named`; however, only one of those parameters (`p`) is actually changing. The name we are looking for does not change, as we would expect: we are looking for the same name in the entire tree. How can we reflect this in the code?

First, we'll do something that looks a little useless, but it's also an innocent change, so it shouldn't irk us too much: we'll change the order of the arguments.

```

fun how-many-named(looking-for :: String, p :: ABT):
  cases (ABT) p:
    | unknown => 0
    | person(n, p1, p2) =>
      (if n == looking-for: 1 else: 0 end)
      +
      how-many-named(looking-for, p1) + how-many-named(looking-for, p2)
  end
end

```

What we have now done is put the “constant” argument first, and the “varying” argument second. That is, our contract has changed from

`how-many-named :: ABT, String -> Number`

to

`how-many-named :: String, ABT -> Number`

so the way we call it also has to change: `how-many-named("A", p)` instead. Observe that the function calls inside `how-many-named` have also been altered in the same way.

23.4 Separating Parameters This sets us up for the next stage. The parameters of functions are meant to indicate what might vary in a function. Because the name we're looking for is a constant once we initially have it, we'd like the actual search function to take only one argument: where in the tree we're searching. That is, we want its contract to be `(ABT → Number)`. To achieve that, we need another function that will take the `String` part.

We achieve this by breaking up the parameter list:

```
fun how-many-named(looking-for :: String) -> (ABT -> Number):
  lam(p :: ABT) -> Number:
    cases (ABT) p:
      | unknown => 0
      | person(n, p1, p2) =>
        (if n == looking-for: 1 else: 0 end)
        +
        how-many-named(looking-for, p1) + how-many-named(looking-for, p2)
    end
  end
end
```

Thus, the contract has become

```
how-many-named :: String -> (ABT -> Number)
```

where `how-many-named` consumes a name and returns a function that will consume the actual tree to check.

However, this function body is not okay: the Pyret type-checker will give us type errors. That's because `how-many-named` takes one parameter, not two, as in the two recursive calls.

How do we fix this? Remember, the whole point of this change is we don't want to change the name, only the tree. That means we want to recur on the inner function. We currently can't do this because it doesn't have a name! So we have to give it a name:

```
fun how-many-named(looking-for :: String) -> (ABT -> Number):
  fun search-in(p :: ABT) -> Number:
    cases (ABT) p:
      | unknown => 0
      | person(n, p1, p2) =>
        (if n == looking-for: 1 else: 0 end)
        +
        search-in(p1) + search-in(p2)
    end
  end
end
```

This now lets us recur on just the part that should vary, leaving the name we're looking for unchanged (and hence, fixed for the duration of the search). However, the above body has a syntax error, because `how-many-named` does not seem to return any kind of value.

What should it return? Once we provide the function with a name, we should get back a function that searches for that name in a tree. But we already have exactly such a function: `search-in`. Therefore, `how-many-named` should just ... return `search-in`.

```
fun how-many-named(looking-for :: String) -> (ABT -> Number):
  fun search-in(p :: ABT) -> Number:
    cases (ABT) p:
      | unknown => 0
      | person(n, p1, p2) =>
        (if n == looking-for: 1 else: 0 end)
        +

```

```

    search-in(p1) + search-in(p2)
end
end
search-in
end

```

And we're set! We would call this function as

```
how-many-named("A")(p) is 3
```

The outer function application returns a function value (bound to `search-in`), is then applied to the particular tree.

23.5 Context The transformation we just applied is generally called currying, in honor of Haskell Curry, who was one of the early people to describe it, though it was earlier discovered by Moses Schönfinkel and even earlier by Gottlob Frege. The particular use of currying here, where we move more “static” arguments earlier and more “dynamic” ones later, and split on the static-dynamic divide, is called staging. It’s a very useful programming technique, and furthermore, one that enables some compilers to produce more time-efficient programs.

Even more subtly but importantly, the staged computation tells a different story than the unstaged one, and we can read this off just from the contract:

```

how-many-named :: (String, ABT -> Number)
how-many-named :: (String -> (ABT -> Number))

```

The first one says the string could co-vary with the person. The second one rules out that interpretation.

Do Now!

Is the former useful? When might we have the name also changing?

Imagine a slightly different problem: we want to know how often a child has the same name as a parent. Then, as we traverse the tree, as the name of the person (potentially) keeps changing, the name we’re looking for in the parent also changes.

Exercise

Write this function.

In contrast, the staged type rules out that interpretation and that behavior. In that way, it sends a signal to the reader about how the computation might behave just from the type. In the same way, the unstaged type can be read as giving the reader a hint that the behavior could depend on both parameters changing, therefore accommodating a much broader range of behaviors (e.g., checking for parent-child or grandparent-child name reuse).

There’s another very nice example of staging here: A Little Calculus.

Finally, it’s worth knowing that some languages, like Haskell and OCaml, do this transformation automatically. In fact, they don’t even have multiple-parameter functions: what look like multiple arguments are actually a sequence of staged functions. This can, in extremis, lead to a very elegant and powerful programming style. Pyret chose to not do this because, while this is a powerful tool in the hands of advanced programmers, for less experienced programmers, finding out about a mismatch in the number of parameters and arguments is very useful.

contents ← prev up next →

24 Factoring Numbers

24 Factoring Numbers

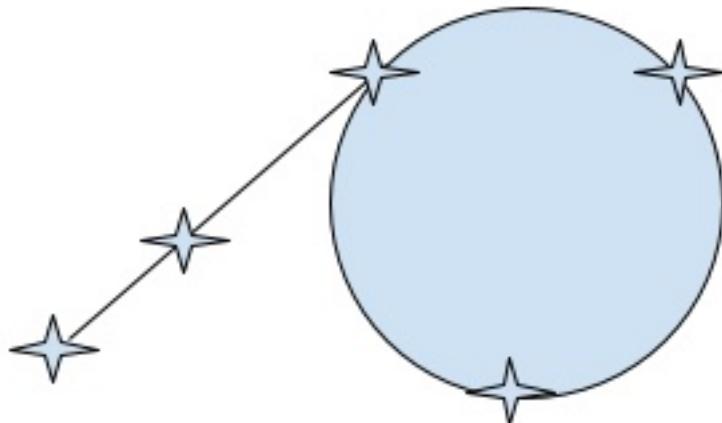
Much of modern cryptography is founded on the difficulty of factoring numbers. Suppose we want to factorize \sqrt{n} . We can just check whether any of the numbers from $\sqrt{2}$ to $\sqrt{n-1}$ (indeed, up to $\sqrt{\sqrt{n}}$) divides \sqrt{n} : if it does, then it's a factor, and we recursively factor what's left. So that just takes a linear amount of time! Why is this hard?

The problem is it's linear in the “wrong” thing: the value of the number. However, the value of a number is, in a place notation, in the worst case exponential in its size. So we'd have to iterate until at least the square root of the exponential of the size, which is size divided by 2, which is in the same big-O class, i.e., exponential in the value. In general we don't really know how to improve the worst-case performance of factorization, which is why contemporary cryptography works. (We discuss numbers elsewhere too [The Complexity of Numbers].)

In practice, it is useful to have factorization algorithms that terminate quickly. They obviously cannot be perfect; we have to compromise instead on accuracy in one way or another: reporting a non-prime as a prime, reporting a non-factor as a factor, etc.

One well-known algorithm is called Pollard's rho algorithm. It will attempt to find a factor for a number. If it succeeds, we are guaranteed that what it found is indeed a factor. If it fails, however, we cannot be sure that the number is actually prime: there may be other factors lurking.

The algorithm gets its name from a picture that should be familiar from Detecting Cycles:



If you rotate that a little bit, you get the Greek letter ρ . The similarity, as we will see in a moment, is not a coincidence.

Explaining the algorithm requires more number theory than we can cover here: if you're interested, read more about it on Wikipedia. Instead, we will focus on the code.

First, we need a helper function that can compute the greatest common denominator:

```
fun gcd(a, b):
    if b == 0:
        a
    else:
```

```

    gcd(b, num-modulo(a, b))
end
end

fun pr(n):
  fun g(x): num-modulo((x * x) + 1, n) end
  fun loop(x, y, d):
    new-x = g(x)
    new-y = g(g(y))
    new-d = gcd(num-abs(new-x - new-y), n)
    ask:
      | new-d == 1 then:
        loop(new-x, new-y, new-d)
      | new-d == n then:
        none
      | otherwise:
        some(new-d)
    end
  end
  loop(2, 2, 1)
end

```

The key step is the computation $g(x)$ versus $g(g(x))$. We can imagine x is the tortoise, so $g(x)$ is the tortoise's update, while y is the hare, so $g(g(y))$ is the hare's update.

Try to run the above on the following values and see what it produces:

```

pr(6)
pr(14)
pr(35)
pr(37)
pr(41)
pr(8)
pr(44)

```

In general, we can check the first few numbers and see how closely they match our intuition:

```

for map(n from range(2, 100)):
  cases (Option) pr(n):
    | none => num-to-string(n) + " may be prime"
    | some(v) => num-to-string(n) + " has factor " + num-to-string(v)
  end
end

```

Exercise

Do you see any patterns in the above output? Does it help you make any conjectures about the algorithm? Can you mathematically prove your conjectures?

contents ← prev up next →

25 Deconstructing Loops

25.1 Setup: Two Functions Let's look at two functions we wrote earlier in Factoring Numbers:

```
fun gcd(a, b):
    if b == 0:
        a
    else:
        gcd(b, num-modulo(a, b))
    end
end

fun pr(n):
    fun g(x): num-modulo((x * x) + 1, n) end
    fun iter(x, y, d):
        new-x = g(x)
        new-y = g(g(y))
        new-d = gcd(num-abs(new-x - new-y), n)
        ask:
            | new-d == 1 then:
                iter(new-x, new-y, new-d)
            | new-d == n then:
                none
            | otherwise:
                some(new-d)
        end
    end
    iter(2, 2, 1)
end
```

We've written both recursively: `gcd` by calling itself and `pr` with recursion on its inner function. But if you've programmed before, you've probably written similar programs with loops.

Exercise

Because we don't have loops in Pyret, the best we can do is to use a higher-order function; which ones would you use?

But let's see if we can do something "better", i.e., get closer to a traditional-looking program.

Before we start changing any code, let's make sure we have some tests for `gcd`:

```
check:
    gcd(4, 5) is 1
    gcd(5, 7) is 1
    gcd(21, 21) is 21
    gcd(12, 24) is 12
    gcd(12, 9) is 3
end
```

25.2 Abstracting a Loop Now let's think about how we can create a loop. At each iteration, a loop has a status: whether it's done or whether it should continue. Since we have two parameters here, let's record two parameters for

continuing:

```
data LoopStatus:  
| done(final-value)  
| next-2(new-arg-1, new-arg-2)  
end
```

Now we can write a function that does the actual iteration:

```
fun loop-2(f, arg-1, arg-2):  
    r = f(arg-1, arg-2)  
    cases (LoopStatus) r:  
        | done(v) => v  
        | next-2(new-arg-1, new-arg-2) => loop-2(f, new-arg-1, new-arg-2)  
    end  
end
```

Note that this is completely generic: it has nothing to do with `gcd`. (It is generic in the same way that higher-order functions like `map` and `filter` are generic.) It just repeats if `f` says to repeat, stops if `f` says to stop. This is the essence of a loop.

Exercise

Observe also that we could, if we wanted, stage [Staging] `loop-2`, because `f` never changes.
Rewrite it that way.

With `loop-2`, we can rewrite `gcd`:

```
fun gcd(p, q):  
    loop-2(  
        {(a, b):  
            if b == 0:  
                done(a)  
            else:  
                next-2(b, num-modulo(a, b))  
            end},  
        p,  
        q)  
end
```

Now it might seem to you we haven't done anything useful at all. In fact, this looks like a significant step backward. At least before we just had simple, clean recursion, the way Euclid intended it. Now we have a higher-order function and we're passing it the erstwhile `gcd` code as a function and there's this `LoopStatus` datatype and...everything's gotten much more complicated.

But, not really. The reason we put it in this form is because we're about to exploit a feature of Pyret. The `for` construct in Pyret actually rewrites as follows:

```
for F(a from a_i, b from b_i, ...): BODY end
```

gets rewritten to

```
F({(a, b, ...): BODY}, a_i, b_i, ...)
```

For example, if we write

```
for map(i from range(0, 10)): i + 1 end
```

this becomes

```
map({(i): i + 1}, range(0, 10))
```

Now you may see why we rewrote `gcd`. Going in reverse, we can rewrite

```
F({(a, b, ...): BODY}, a_i, b_i, ...)
```

as

```
for F(a from a_i, b from b_i, ...): BODY end
```

so the function becomes just

```
fun gcd(p, q):
    for loop-2(a from p, b from q):
        if b == 0:
            done(a)
        else:
            next-2(b, num-modulo(a, b))
    end
end
```

and now closely resembles a traditional “loop” program.

25.3 Is It Really a Loop? This whole section should be considered an aside for people with more advanced computing knowledge.

If you know something about language implementation, you may know that loops have the property that the iteration does not consume extra space (beyond what the program already needs), and the repetition takes place very quickly (a “jump instruction”). In principle, our `loop-2` function does not have this property: every iteration is a function call, which is more expensive and builds additional stack context. However, one or both of these does not actually occur in practice.

In terms of space, the recursive call to `loop-2` is the last thing that a call to `loop-2` does. Furthermore, nothing in `loop-2` consumes and manipulates the return from that recursive call. This is therefore called a tail call. Pyret—like some other languages—causes tail calls to not take any extra stack space. In principle, Pyret can also turn some tail calls into jumps. Therefore, this version has close to the same performance as a traditional loop.

25.4 Re-Examining `for` The definition of `for` given above should make you suspicious: Where’s the loop?!? In fact, Pyret’s `for` does not do any looping at all: it’s simply a fancy way of writing `lam`. Any “looping” behavior is in the function written after `for`. To see that, let’s use `for` with a non-looping function.

Recall that

```
for F(a from a_i, b from b_i, ...): BODY end
```

gets rewritten to

```
F({(a, b, ...): BODY}, a_i, b_i, ...)
```

Thus, suppose we have this function (from Functions as Data):

```
delta-x = 0.0001
fun d-dx-at(f, x):
    (f(x + delta-x) - f(x)) / delta-x
end
```

We can call it like this to get approximately 20:

```
d-dx-at({(n): n * n}, 10)
```

That means we can also call it like this:

```
for d-dx-at(n from 10): n * n end
```

Indeed:

```
check:
    for d-dx-at(n from 10): n * n end
    is
        d-dx-at({(n): n * n}, 10)
```

```
end
```

Since `d-dx-at` has no iterative behavior, no iteration occurs. The looping behavior is given entirely by the function specified after `for`, such as `map`, `filter`, or `loop-2` above.

25.5 Rewriting Pollard-Rho Now let's tackle Pollard-rho. Notice that it's a three-parameter function, so we can't use the `loop-2` we had before: that's only a suitable loop when we have two arguments that change on each iteration (often the iteration variable and an accumulator). It would be easy to design a 3-argument version of `loop`, say `loop-3`, but we could also have a more general solution, using a tuple:

```
data LoopStatus:  
| done(v)  
| next-2(new-x, new-y)  
| next-n(new-t)  
end  
  
fun loop-n(f, t):  
  r = f(t)  
  cases (LoopStatus) r:  
  | done(v) => v  
  | next-n(new-t) => loop-n(f, new-t)  
  end  
end
```

where `t` is a tuple.

So now we can rewrite `pr`. Let's first rename the old `pr` function as `pr-old` so we can keep it around for testing. Now we can define a “loop”-based `pr`:

```
fun pr(n):  
  fun g(x): num-modulo((x * x) + 1, n) end  
  for loop-n({x; y; d} from {2; 2; 1}):  
    new-x = g(x)  
    new-y = g(g(y))  
    new-d = gcd(num-abs(new-x - new-y), n)  
    ask:  
    | new-d == 1 then:  
      next-n({new-x; new-y; new-d})  
    | new-d == n then:  
      done(None)  
    | otherwise:  
      done(Some(new-d))  
    end  
  end  
end
```

Indeed, we can test that the two behave in exactly the same way:

```
check:  
  ns = range(2, 100)  
  l1 = map(pr-old, ns)  
  l2 = map(pr, ns)  
  l1 is l2  
end
```

25.6 Nested Loops We can also write a nested loop this way. Suppose we have a list like

```
lol = [list: [list: 1, 2], [list: 3], [list:], [list: 4, 5, 6]]
```

and we want to sum the whole thing by summing each sub-list. Here it is:

```

for loop-2(l1 from lol, sum from 0):
  cases (List) l1:
    | empty => done(sum)
    | link(l, rl) =>
      l-sum =
        for loop-2(es from l, sub-sum from 0):
          cases (List) es:
            | empty => done(sub-sum)
            | link(e, r) => next-2(r, e + sub-sum)
          end
        end
      next-2(rl, sum + l-sum)
    end
  end
end

```

We can simplify this by writing it as two functions:

```

fun sum-a-lon(lon :: List<Number>):
  for loop-2(es from lon, sum from 0):
    cases (List) es:
      | empty => done(sum)
      | link(e, r) =>
        next-2(r, e + sum)
    end
  end
end

fun sum-a-lolon(lolon :: List<List<Number>>):
  for loop-2(l from lolon, sum from 0):
    cases (List) l:
      | empty => done(sum)
      | link(lon, r) =>
        next-2(r, sum-a-lon(lon) + sum)
    end
  end
end

check:
  sum-a-lolon(lol) is 21
end

```

Notice that the two functions are remarkably similar. This suggests an abstraction:

```

fun sum-a-list(f, L):
  for loop-2(e from L, sum from 0):
    cases (List) e:
      | empty => done(sum)
      | link(elt, r) =>
        next-2(r, f(elt) + sum)
    end
  end
end

```

Using this, we can rewrite the two previous functions as:

```

fun sum-a-lon(lon :: List<Number>):
  sum-a-list({(e): e}, lon)
end

```

```

fun sum-a-lolon(lolon :: List<List<Number>>):
    sum-a-list(sum-a-lon, lolon)
end

check:
  sum-a-lolon(lol) is 21
end

```

With the annotations, it becomes clear what each function does. In `sum-a-lon`, each element is a number, so it “contributes itself” to the overall sum. In `sum-a-lolon`, each element is a list of numbers, so it “contributes its `sum-a-lon`” to the overall sum.

Finally, to bring this full circle, we can rewrite the above the functions as follows:

```

fun sum-a-lon(lon :: List<Number>):
    for sum-a-list(e :: Number from lon): e end
end

fun sum-a-lolon(lolon :: List<List<Number>>):
    for sum-a-list(l :: List<Number> from lolon): sum-a-lon(l) end
end

```

Arguably this makes even clearer what each element contributes. In `sum-a-lon` each element is a number, so it contributes just that number. In `sum-a-lolon`, each element is a list of numbers, so it must contribute `sum-a-lon` of that list.

25.7 Loops, Values, and Customization Observe two important ways in which the loops above differ from traditional loops:

1. Every loop produces a value. This is consistent with the rest of the language, where—as much as possible—computations try to produce answers. We don’t have to produce a value; for instance, the following program, reminiscent of looping programs in many other languages, will work just fine in Pyret:

```
for each(i from range(0, 10)): print(i) end
```

However, this is the unusual case. In general, we want expressions to produce values so that we can compose them together.

2. Many languages have strong opinions on exactly how many looping constructs there should be: two? three? four? In Pyret, there are no built-in looping constructs at all; there’s just a syntax (`for`) that serves as a proxy for creating a specific `lam`. With it, we can reuse existing iterative functions (like `map` and `filter`), but also define new ones. Some can be very generic, like `loop-2` or `loop-n`, but others can be very specific, like `sum-a-list`. The language designers don’t prevent you from writing a loop that is useful to your situation, and sometimes the loop can be very expressive, as we see from rewriting `sum-a-lon` and `sum-a-lolon` atop `for` and `sum-a-list`.

contents ← prev up next →

26 Interactive Games as Reactive Systems

26 Interactive Games as Reactive Systems

In this tutorial we're going to write a little interactive game. The game won't be sophisticated, but it'll have all the elements you need to build much richer games of your own.



Albuquerque Balloon Fiesta

Imagine we have an airplane coming in to land. It's unfortunately trying to do so amidst a hot-air balloon festival, so it naturally wants to avoid colliding with any (moving) balloons. In addition, there is both land and water, and the airplane needs to alight on land. We might also equip it with limited amounts of fuel to complete its task. Here are some animations of the game:

- The airplane comes in to land successfully.
- Uh oh—the airplane collides with a balloon!
- Uh oh—the airplane lands in the water!

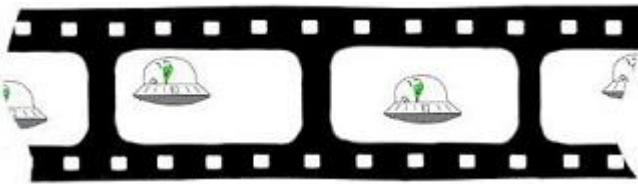
By the end, you will have written all the relevant portions of this program. Your program will:

- animate the airplane to move autonomously;
- detect keystrokes and adjust the airplane accordingly;
- have multiple moving balloons;
- detect collisions between the airplane and balloons;
- check for landing on water and land; and
- account for the use of fuel.

Phew: that's a lot going on! Therefore, we won't write it all at once; instead, we'll build it up bit-by-bit. But we'll get there by the end.

26.1 About Reactive Animations We are writing a program with two important interactive elements: it is an animation, meaning it gives the impression of motion, and it is reactive, meaning it responds to user input. Both of these can be challenging to program, but Pyret provides a simple mechanism that accommodates both and integrates well with other programming principles such as testing. We will learn about this as we go along.

The key to creating an animation is the Movie Principle. Even in the most sophisticated movie you can watch, there is no motion (indeed, the very term “movie”—short for “moving picture”—is a clever bit of false advertising). Rather, there is just a sequence of still images shown in rapid succession, relying on the human brain to create the impression of motion:



We are going to exploit the same idea: our animations will consist of a sequence of individual images, and we will ask Pyret to show these in rapid succession. We will then see how reactivity folds into the same process.

26.2 Preliminaries To begin with, we should inform Pyret that we plan to make use of both images and animations. We load the libraries as follows:

```
import image as I
import reactors as R
```

This tells Pyret to load these two libraries and bind the results to the corresponding names, `I` and `R`. Thus, all image operations are obtained from `I` and animation operations from `R`.

26.3 Version: Airplane Moving Across the Screen We will start with the simplest version: one in which the airplane moves horizontally across the screen. Watch this video.

First, here's an image of an airplane: Have fun finding your preferred airplane image! But don't spend too long on it, because we've still got a lot of work to do.

<http://world.cs.brown.edu/1/clipart/airplane-small.png>

We can tell Pyret to load this image and give it a name as follows:

```
AIRPLANE-URL =
"http://world.cs.brown.edu/1/clipart/airplane-small.png"
AIRPLANE = I.image-url(AIRPLANE-URL)
```

Henceforth, when we refer to `AIRPLANE`, it will always refer to this image. (Try it out in the interactions area!)

Now look at the video again. Watch what happens at different points in time. What stays the same, and what changes? What's common is the water and land, which stay the same. What changes is the (horizontal) position of the airplane.

Note

The World State consists of everything that changes. Things that stay the same do not need to get recorded in the World State.

We can now define our first World State:

World Definition

The World State is a number, representing the x-position of the airplane.

Observe something important above:

Note

When we record a World State, we don't capture only the type of the values, but also their intended meaning.

Now we have a representation of the core data, but to generate the above animation, we still have to do several things:

1. Ask to be notified of the passage of time.

2. As time passes, correspondingly update the World State.
3. Given an updated World State, produce the corresponding visual display.

This sounds like a lot! Fortunately, Pyret makes this much easier than it sounds. We'll do these in a slightly different order than listed above.

26.3.1 Updating the World State As we've noted, the airplane doesn't actually "move". Rather, we can ask Pyret to notify us every time a clock ticks. If on each tick we place the airplane in an appropriately different position, and the ticks happen often enough, we will get the impression of motion.

Because the World State consists of just the airplane's x-position, to move it to the right, we simply increment its value. Let's first give this constant distance a name:

```
AIRPLANE-X-MOVE = 10
```

We will need to write a function that reflects this movement. Let's first write some test cases:

```
check:
  move-airplane-x-on-tick(50) is 50 + AIRPLANE-X-MOVE
  move-airplane-x-on-tick(0) is 0 + AIRPLANE-X-MOVE
  move-airplane-x-on-tick(100) is 100 + AIRPLANE-X-MOVE
end
```

The function's definition is now clear:

```
fun move-airplane-x-on-tick(w):
  w + AIRPLANE-X-MOVE
end
```

And sure enough, Pyret will confirm that this function passes all of its tests.

Note

If you have prior experience programming animations and reactive programs, you will immediately notice an important difference: it's easy to test parts of your program in Pyret!

26.3.2 Displaying the World State Now we're ready to draw the game's visual output. We produce an image that consists of all the necessary components. It first helps to define some constants representing the visual output:

```
WIDTH = 800
HEIGHT = 500
```

```
BASE-HEIGHT = 50
WATER-WIDTH = 500
```

Using these, we can create a blank canvas, and overlay rectangles representing water and land:

```
BLANK-SCENE = I.empty-scene(WIDTH, HEIGHT)

WATER = I.rectangle(WATER-WIDTH, BASE-HEIGHT, "solid", "blue")
LAND = I.rectangle(WIDTH - WATER-WIDTH, BASE-HEIGHT, "solid", "brown")

BASE = I.beside(WATER, LAND)

BACKGROUND =
  I.place-image(BASE,
    WIDTH / 2, HEIGHT - (BASE-HEIGHT / 2),
    BLANK-SCENE)
```

Examine the value of BACKGROUND in the interactions area to confirm that it looks right.

Do Now!

The reason we divide by two when placing `BASE` is because Pyret puts the middle of the image at the given location. Remove the division and see what happens to the resulting image.

Now that we know how to get our background, we're ready to place the airplane on it. The expression to do so looks roughly like this:

```
I.place-image(AIRPLANE,  
  # some x position,  
  50,  
  BACKGROUND)
```

but what `x` position do we use? Actually, that's just what the World State represents! So we create a function out of this expression:

```
fun place-airplane-x(w):  
  I.place-image(AIRPLANE,  
    w,  
    50,  
    BACKGROUND)  
end
```

26.3.3 Observing Time (and Combining the Pieces)

Finally, we're ready to put these pieces together.

We create a special kind of Pyret value called a reactor, which creates animations. We'll start by creating a fairly simple kind of reactor, then grow it as the program gets more sophisticated.

The following code creates a reactor named `anim`:

```
anim = reactor:  
  init: 0,  
  on-tick: move-airplane-x-on-tick,  
  to-draw: place-airplane-x  
end
```

A reactor needs to be given an initial World State as well as handlers that tell it how to react. Specifying `on-tick` tells Pyret to run a clock and, every time the clock ticks (roughly thirty times a second), invoke the associated handler. The `to-draw` handler is used by Pyret to refresh the visual display.

Having defined this reactor, we can run it in several ways that are useful for finding errors, running scientific experiments, and so on. Our needs here are simple; we ask Pyret to just run the program on the screen interactively:

```
R.interact(anim)
```

This creates a running program where the airplane flies across the background!

That's it! We've created our first animation. Now that we've gotten all the preliminaries out of the way, we can go about enhancing it.

Exercise

If you want the airplane to appear to move faster, what can you change?

26.4 Version: Wrapping Around When you run the preceding program, you'll notice that after a while, the airplane just disappears. This is because it has gone past the right edge of the screen; it is still being "drawn", but in a location that you cannot see. That's not very useful! Also, after a long while you might get an error because the computer is being asked to draw the airplane at a location beyond what the graphics system can manage. Instead, when the airplane is about to go past the right edge of the screen, we'd like it to reappear on the left by a corresponding amount: "wrapping around", as it were.

Here's the video for this version.

Do Now!

What needs to change?

Clearly, we need to modify the function that updates the airplane's location, since this must now reflect our decision to wrap around. But the task of how to draw the airplane doesn't need to change at all! Similarly, the definition of the World State does not need to change, either.

Therefore, we only need to modify `move-airplane-x-on-tick`. The function `num-modulo` does exactly what we need. That is, we want the x-location to always be modulo the width of the scene:

```
fun move-airplane-wrapping-x-on-tick(x):
    num-modulo(x + AIRPLANE-X-MOVE, WIDTH)
end
```

Notice that, instead of copying the content of the previous definition we can simply reuse it:

```
fun move-airplane-wrapping-x-on-tick(x):
    num-modulo(move-airplane-x-on-tick(x), WIDTH)
end
```

which makes our intent clearer: compute whatever position we would have had before, but adapt the coordinate to remain within the scene's width.

Well, that's a proposed re-definition. Be sure to test this function thoroughly: it's trickier than you might think! Have you thought about all the cases? For instance, what happens if the airplane is half-way off the right edge of the screen?

Exercise

Define quality tests for `move-airplane-wrapping-x-on-tick`.

Note

It is possible to leave `move-airplane-x-on-tick` unchanged and perform the modular arithmetic in `place-airplane-x` instead. We choose not to do that for the following reason. In this version, we really do think of the airplane as circling around and starting again from the left edge (imagine the world is a cylinder...). Thus, the airplane's x-position really does keep going back down. If instead we allowed the World State to increase monotonically, then it would really be representing the total distance traveled, contradicting our definition of the World State.

Do Now!

After adding this function, run your program again. Did you see any change in behavior?

If you didn't...did you remember to update your reactor to use the new airplane-moving function?

26.5 Version: Descending Of course, we need our airplane to move in more than just one dimension: to get to the final game, it must both ascend and descend as well. For now, we'll focus on the simplest version of this, which is an airplane that continuously descends. Here's a video.

Let's again consider individual frames of this video. What's staying the same? Once again, the water and the land. What's changing? The position of the airplane. But, whereas before the airplane moved only in the x-dimension, now it moves in both x and y. That immediately tells us that our definition of the World State is inadequate, and must be modified.

We therefore define a new structure to hold this pair of data:

```
data Posn:
    | posn(x, y)
end
```

Given this, we can revise our definition:

World Definition

The World State is a `posn`, representing the x-position and y-position of the airplane on the screen.

26.5.1 Moving the Airplane First, let's consider `move-airplane-wrapping-x-on-tick`. Previously our airplane moved only in the x-direction; now we want it to descend as well, which means we must add something to the current y value:

```
AIRPLANE-Y-MOVE = 3
```

Let's write some test cases for the new function. Here's one:

```
check:  
  move-airplane-xy-on-tick(posn(10, 10)) is posn(20, 13)  
end
```

Another way to write the test would be:

```
check:  
  p = posn(10, 10)  
  move-airplane-xy-on-tick(p) is  
    posn(move-airplane-wrapping-x-on-tick(p.x),  
          move-airplane-y-on-tick(p.y))  
end
```

Note

Which method of writing tests is better? Both! They each offer different advantages:

- The former method has the benefit of being very concrete: there's no question what you expect, and it demonstrates that you really can compute the desired answer from first principles.
- The latter method has the advantage that, if you change the constants in your program (such as the rate of descent), seemingly correct tests do not suddenly fail. That is, this form of testing is more about the relationships between things rather than their precise values.

There is one more choice available, which often combines the best of both worlds: write the answer as concretely as possible (the former style), but using constants to compute the answer (the advantage of the latter style). For instance:

```
check:  
  p = posn(10, 10)  
  move-airplane-xy-on-tick(p) is  
    posn(num-modulo(p.x + AIRPLANE-X-MOVE, WIDTH),  
         p.y + AIRPLANE-Y-MOVE)  
end
```

Exercise

Before you proceed, have you written enough test cases? Are you sure? Have you, for instance, tested what should happen when the airplane is near the edge of the screen in either or both dimensions? We thought not—go back and write more tests before you proceed!

Using the design recipe, now define `move-airplane-xy-on-tick`. You should end up with something like this:

```
fun move-airplane-xy-on-tick(w):  
  posn(move-airplane-wrapping-x-on-tick(w.x),  
        move-airplane-y-on-tick(w.y))  
end
```

Note that we have reused the existing function for the x-dimension and, correspondingly, created a helper for the y dimension:

```
fun move-airplane-y-on-tick(y):  
  y + AIRPLANE-Y-MOVE  
end
```

This may be slight overkill for now, but it does lead to a cleaner separation of concerns, and makes it possible for the complexity of movement in each dimension to evolve independently while keeping the code relatively readable.

26.5.2 Drawing the Scene We have to also examine and update `place-airplane-x`. Our earlier definition placed the airplane at an arbitrary y-coordinate; now we have to take the y-coordinate from the World State:

```
fun place-airplane-xy(w):
    I.place-image(AIRPLANE,
        w.x,
        w.y,
        BACKGROUND)
end
```

Notice that we can't really reuse the previous definition because it hard-coded the y-position, which we must now make a parameter.

26.5.3 Finishing Touches Are we done? It would seem so: we've examined all the procedures that consume and produce World State and updated them appropriately. Actually, we're forgetting one small thing: the initial World State given to `big-bang!` If we've changed the definition of World State, then we need to reconsider this parameter, too. (We also need to pass the new handlers rather than the old ones.)

```
INIT-POS = posn(0, 0)

anim = reactor:
    init: INIT-POS,
    on-tick: move-airplane-xy-on-tick,
    to-draw: place-airplane-xy
end

R.interact(anim)
```

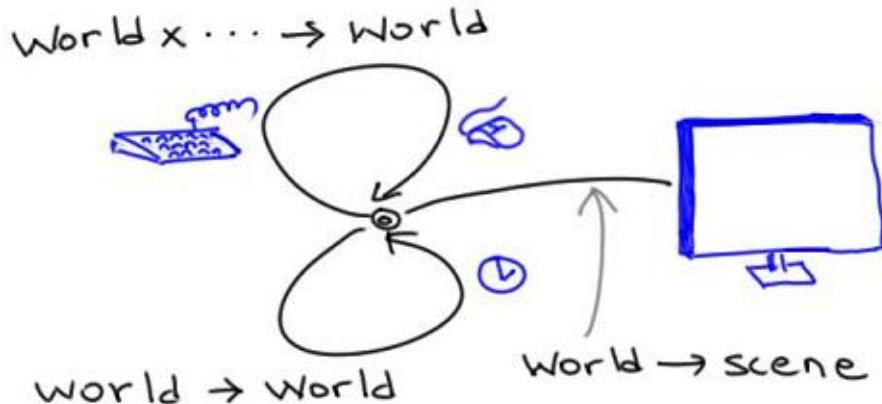
Exercise

It's a little unsatisfactory to have the airplane truncated by the screen. You can use `I.image-width` and `I.image-height` to obtain the dimensions of an image, such as the airplane. Use these to ensure the airplane fits entirely within the screen for the initial scene, and similarly in `move-airplane-xy-on-tick`.

26.6 Version: Responding to Keystrokes Now that we have the airplane descending, there's no reason it can't ascend as well. Here's a video.

We'll use the keyboard to control its motion: specifically, the up-key will make it move up, while the down-key will make it descend even faster. This is easy to support using what we already know: we just need to provide one more handler using `on-key`. This handler takes two arguments: the first is the current value of the world, while the second is a representation of which key was pressed. For the purposes of this program, the only key values we care about are "up" and "down".

This gives us a fairly comprehensive view of the core capabilities of reactors:



We just define a group of functions to perform all our desired actions, and the reactor strings them together. Some functions update world values (sometimes taking additional information about a stimulus, such as the key pressed), while others transform them into output (such as what we see on the screen).

Returning to our program, let's define a constant representing how much distance a key represents:

```
KEY-DISTANCE = 10
```

Now we can define a function that alter's the airplane's position by that distance depending on which key is pressed:

```
fun alter-airplane-y-on-key(w, key):
    ask:
        | key == "up"    then: posn(w.x, w.y - KEY-DISTANCE)
        | key == "down" then: posn(w.x, w.y + KEY-DISTANCE)
        | otherwise: w
    end
end
```

Do Now!

Why does this function definition contain

```
| otherwise: w
```

as its last condition?

Notice that if we receive any key other than the two we expect, we leave the World State as it was; from the user's perspective, this has the effect of just ignoring the keystroke. Remove this last clause, press some other key, and watch what happens!

No matter what you choose, be sure to test this! Can the airplane drift off the top of the screen? How about off the screen at the bottom? Can it overlap with the land or water?

Once we've written and thoroughly tested this function, we simply need to ask Pyret to use it to handle keystrokes:

```
anim = reactor:
    init: INIT-POS,
    on-tick: move-airplane-xy-on-tick,
    on-key: alter-airplane-y-on-key,
    to-draw: place-airplane-xy
end
```

Now your airplane moves not only with the passage of time but also in response to your keystrokes. You can keep it up in the air forever!

26.7 Version: Landing Remember that the objective of our game is to land the airplane, not to keep it airborne indefinitely. That means we need to detect when the airplane reaches the land or water level and, when it does, terminate the animation.

First, let's try to characterize when the animation should halt. This means writing a function that consumes the current World State and produces a boolean value: `true` if the animation should halt, `false` otherwise. This requires a little arithmetic based on the airplane's size:

```
fun is-on-land-or-water(w):
    w.y >= (HEIGHT - BASE-HEIGHT)
end
```

We just need to inform Pyret to use this predicate to automatically halt the reactor:

```
anim = reactor:
    init: INIT-POS,
    on-tick: move-airplane-xy-on-tick,
    on-key: alter-airplane-y-on-key,
    to-draw: place-airplane-xy,
    stop-when: is-on-land-or-water
```

```
end
```

Exercise

When you test this, you'll see it isn't quite right because it doesn't take account of the size of the airplane's image. As a result, the airplane only halts when it's half-way into the land or water, not when it first touches down. Adjust the formula so that it halts upon first contact.

Exercise

Extend this so that the airplane rolls for a while upon touching land, decelerating according to the laws of physics.

Exercise

Suppose the airplane is actually landing at a secret subterranean airbase. The actual landing strip is actually below ground level, and opens up only when the airplane comes in to land. That means, after landing, only the parts of the airplane that stick above ground level would be visible. Implement this. As a hint, consider modifying `place-airplane-xy`.

26.8 Version: A Fixed Balloon

Now let's add a balloon to the scene. Here's a video of the action.

Notice that while the airplane moves, everything else—including the balloon—stays immobile. Therefore, we do not need to alter the World State to record the balloon's position. All we need to do is alter the conditions under which the program halts: effectively, there is one more situation under which it terminates, and that is a collision with the balloon.

When does the game halt? There are now two circumstances: one is contact with land or water, and the other is contact with the balloon. The former remains unchanged from what it was before, so we can focus on the latter.

Where is the balloon, and how do we represent where it is? The latter is easy to answer: that's what `posns` are good for. As for the former, we can decide where it is:

```
BALLOON-LOC = posn(600, 300)
```

or we can let Pyret pick a random position:

```
BALLOON-LOC = posn(random(WIDTH), random(HEIGHT))
```

Exercise

Improve the random placement of the balloon so that it is in credible spaces (e.g., not submerged).

Given a position for the balloon, we just need to detect collision. One simple way is as follows: determine whether the distance between the airplane and the balloon is within some threshold:

```
fun are-overlapping(airplane-posn, balloon-posn):
  distance(airplane-posn, balloon-posn)
    < COLLISION-THRESHOLD
end
```

where `COLLISION-THRESHOLD` is some suitable constant computed based on the sizes of the airplane and balloon images. (For these particular images, 75 works pretty well.)

What is `distance`? It consumes two `posns` and determines the Euclidean distance between them:

```
fun distance(p1, p2):
  fun square(n): n * n end
  num-sqrt(square(p1.x - p2.x) + square(p1.y - p2.y))
end
```

Finally, we have to weave together the two termination conditions:

```
fun game-ends(w):
  ask:
    | is-on-land-or-water(w)           then: true
```

```

| are-overlapping(w, BALLOON-LOC) then: true
| otherwise: false
end
end

```

and use it instead:

```

anim = reactor:
  init: INIT-POS,
  on-tick: move-airplane-xy-on-tick,
  on-key: alter-airplane-y-on-key,
  to-draw: place-airplane-xy,
  stop-when: game-ends
end

```

Do Now!

Were you surprised by anything? Did the game look as you expected?

Odds are you didn't see a balloon on the screen! That's because we didn't update our display.

You will need to define the balloon's image:

```

BALLOON-URL =
  "http://world.cs.brown.edu/1/clipart/balloon-small.png"
BALLOON = I.image-url(BALLOON-URL)

```

and also update the drawing function:

```

BACKGROUND =
  I.place-image(BASE,
    WIDTH / 2, HEIGHT - (BASE-HEIGHT / 2),
  I.place-image(BALLOON,
    BALLOON-LOC.x, BALLOON-LOC.y,
    BLANK-SCENE))

```

Do Now!

Do you see how to write `game-ends` more concisely?

Here's another version:

```

fun game-ends(w):
  is-on-land-or-water(w) or are-overlapping(w, BALLOON-LOC)
end

```

26.9 Version: Keep Your Eye on the Tank Now we'll introduce the idea of fuel. In our simplified world, fuel isn't necessary to descend—gravity does that automatically—but it is needed to climb. We'll assume that fuel is counted in whole number units, and every ascension consumes one unit of fuel. When you run out of fuel, the program no longer responds to the up-arrow, so you can no longer avoid either the balloon or water.

In the past, we've looked at still images of the game video to determine what is changing and what isn't. For this version, we could easily place a little gauge on the screen to show the quantity of fuel left. However, we don't do this on purpose, to illustrate a principle.

Note

You can't always determine what is fixed and what is changing just by looking at the image. You have to also read the problem statement carefully, and think about it in depth.

It's clear from our description that there are two things changing: the position of the airplane and the quantity of fuel left. Therefore, the World State must capture the current values of both of these. The fuel is best represented as a single number. However, we do need to create a new structure to represent the combination of these two.

World Definition

The World State is a structure representing the airplane's current position and the quantity of fuel left.

Concretely, we will use this structure:

```
data World:  
    | world(p, f)  
end
```

Exercise

We could have also defined the World to be a structure consisting of three components: the airplane's x-position, the airplane's y-position, and the quantity of fuel. Why do we choose to use the representation above?

We can again look at each of the parts of the program to determine what can stay the same and what changes. Concretely, we must focus on the functions that consume and produce *Worlds*.

On each tick, we consume a world and compute one. The passage of time does not consume any fuel, so this code can remain unchanged, other than having to create a structure containing the current amount of fuel. Concretely:

```
fun move-airplane-xy-on-tick(w :: World):  
    world(  
        posn(  
            move-airplane-wrapping-x-on-tick(w.p.x),  
            move-airplane-y-on-tick(w.p.y)),  
        w.f)  
end
```

Similarly, the function that responds to keystrokes clearly needs to take into account how much fuel is left:

```
fun alter-airplane-y-on-key(w, key):  
    ask:  
        | key == "up" then:  
            if w.f > 0:  
                world(posn(w.p.x, w.p.y - KEY-DISTANCE), w.f - 1)  
            else:  
                w # there's no fuel, so ignore the keystroke  
            end  
        | key == "down" then:  
            world(posn(w.p.x, w.p.y + KEY-DISTANCE), w.f)  
        | otherwise: w  
    end  
end
```

Exercise

Updating the function that renders a scene. Recall that the world has two fields; one of them corresponds to what we used to draw before, and the other isn't being drawn in the output.

Do Now!

What else do you need to change to get a working program?

You should have noticed that your initial world value is also incorrect because it doesn't account for fuel. What are interesting fuel values to try?

Exercise

Extend your program to draw a fuel gauge.

26.10 Version: The Balloon Moves, Too Until now we've left our balloon immobile. Let's now make the game more interesting by letting the balloon move, as this video shows.

Obviously, the balloon's location needs to also become part of the World State.

World Definition

The World State is a structure representing the plane's current position, the balloon's current position, and the quantity of fuel left.

Here is a representation of the world state. As these states become more complex, it's important to add annotations so we can keep track of what's what.

```
data World:  
    | world(p :: Posn, b :: Posn, f :: Number)  
end
```

With this definition, we obviously need to re-write all our previous definitions. Most of this is quite routine relative to what we've seen before. The only detail we haven't really specified is how the balloon is supposed to move: in what direction, at what speed, and what to do at the edges. We'll let you use your imagination for this one! (Remember that the closer the balloon is to land, the harder it is to safely land the plane.)

We thus have to modify:

- The background image (to remove the static balloon).
- The drawing handler (to draw the balloon at its position).
- The timer handler (to move the balloon as well as the airplane).
- The key handler (to construct world data that leaves the balloon unchanged).
- The termination condition (to account for the balloon's dynamic location).

Exercise

Modify each of the above functions, along with their test cases.

26.11 Version: One, Two, ..., Ninety-Nine Luftballons! Finally, there's no need to limit ourselves to only one balloon. How many is right? Two? Three? Ten? ... Why fix any one number? It could be a balloon festival!

Similarly, many games have levels that become progressively harder; we could do the same, letting the number of balloons be part of what changes across levels. However, there is conceptually no big difference between having two balloons and five; the code to control each balloon is essentially the same.

We need to represent a collection of balloons. We can use a list to represent them. Thus:

World Definition

The World State is a structure representing the plane's current position, a list of balloon positions, and the quantity of fuel left.

You should now use the design recipe for lists of structures to rewrite the functions. Notice that you've already written the function to move one balloon. What's left?

1. Apply the same function to each balloon in the list.
2. Determine what to do if two balloons collide.

For now, you can avoid the latter problem by placing each balloon sufficiently spread apart along the x-dimension and letting them move only up and down.

Exercise

Introduce a concept of wind, which affects balloons but not the airplane. After random periods of time, the wind blows with random speed and direction, causing the balloons to move laterally.

27 Appendices

27.1 Pyret for Racketeers and Schemers If you've programmed before in a language like Scheme or the student levels of Racket (or the WeScheme programming environment), or for that matter even in certain parts of OCaml, Haskell, Scala, Erlang, Clojure, or other languages, you will find many parts of Pyret very familiar. This chapter is specifically written to help you make the transition from (student) Racket/Scheme/WeScheme (abbreviated "RSW") to Pyret by showing you how to convert the syntax. Most of what we say applies to all these languages, though in some cases we will refer specifically to Racket (and WeScheme) features not found in Scheme.

In every example below, the two programs will produce the same results.

27.1.1 Numbers, Strings, and Booleans Numbers are very similar between the two. Like Scheme, Pyret implements arbitrary-precision numbers and rationals. Some of the more exotic numeric systems of Scheme (such as complex numbers) aren't in Pyret; Pyret also treats imprecise numbers slightly differently.

RSW

Pyret

1

1

RSW

Pyret

1/2

1/2

RSW

Pyret

#i3.14

~3.14

Strings are also very similar, though Pyret allows you to use single-quotes as well.

RSW

Pyret

"Hello, world!"

"Hello, world!"

RSW

Pyret

"\"Hello\\", he said"

"\"Hello\\", he said"

RSW

Pyret

```
\"Hello\", he said"
```

```
'"Hello", he said'
```

Booleans have the same names:

```
RSW
```

```
Pyret
```

```
true
```

```
true
```

```
RSW
```

```
Pyret
```

```
false
```

```
false
```

27.1.2 Infix Expressions Pyret uses an infix syntax, reminiscent of many other textual programming languages:

```
RSW
```

```
Pyret
```

```
(+ 1 2)
```

```
1 + 2
```

```
RSW
```

```
Pyret
```

```
(* (- 4 2) 5)
```

```
(4 - 2) * 5
```

Note that Pyret does not have rules about orders of precedence between operators, so when you mix operators, you have to parenthesize the expression to make your intent clear. When you chain the same operator you don't need to parenthesize; chaining associates to the left in both languages:

```
RSW
```

```
Pyret
```

```
(/ 1 2 3 4)
```

```
1 / 2 / 3 / 4
```

These both evaluate to 1/24.

27.1.3 Function Definition and Application Function definition and application in Pyret have an infix syntax, more reminiscent of many other textual programming languages. Application uses a syntax familiar from conventional algebra books:

```
RSW
```

```
Pyret
```

```
(dist 3 4)
```

```
dist(3, 4)
```

Application correspondingly uses a similar syntax in function headers, and infix in the body:

```
RSW
```

```
Pyret
```

```

(define (dist x y)
  (sqrt (+ (* x x)
            (* y y))))
fun dist(x, y):
  num-sqrt((x * x) +
            (y * y))
end

```

27.1.4 Tests There are essentially three different ways of writing the equivalent of Racket's check-expect tests. They can be translated into check blocks:

RSW

Pyret

```
(check-expect 1 1)
```

```
check:
  1 is 1
end
```

Note that multiple tests can be put into a single block:

RSW

Pyret

```
(check-expect 1 1)
```

```
(check-expect 2 2)
```

```
check:
  1 is 1
  2 is 2
end
```

The second way is this: as an alias for `check` we can also write `examples`. The two are functionally identical, but they capture the human difference between examples (which explore the problem, and are written before attempting a solution) and tests (which try to find bugs in the solution, and are written to probe its design).

The third way is to write a `where` block to accompany a function definition. For instance:

```

fun double(n):
  n + n
where:
  double(0) is 0
  double(10) is 20
  double(-1) is -2
end

```

These can even be written for internal functions (i.e., functions contained inside other functions), which isn't true for check-expect.

In Pyret, unlike in Racket, a testing block can contain a documentation string. This is used by Pyret when reporting test successes and failures. For instance, try to run and see what you get:

```

check "squaring always produces non-negatives":
  (0 * 0) is 0
  (-2 * -2) is 4
  (3 * 3) is 9
end

```

This is useful for documenting the purpose of a testing block.

Just as in Racket, there are many testing operators in Pyret (in addition to `is`). See the documentation.

27.1.5 Variable Names Both languages have a fairly permissive system for naming variables. While you can use CamelCase and under_scores in both, it is conventional to instead use what is known as kebab-case. This name is inaccurate. The word “kebab” just means “meat”. The skewer is the “shish”. Therefore, it ought to at least be called “shish kebab case”. Thus:

RSW

Pyret

this-is-a-name

this-is-a-name

Even though Pyret has infix subtraction, the language can unambiguously tell apart `this-name` (a variable) from `this - name` (a subtraction expression) because the `-` in the latter must be surrounded by spaces.

Despite this spacing convention, Pyret does not permit some of the more exotic names permitted by Scheme. For instance, one can write

```
(define e^i*pi -1)
```

in Scheme but that is not a valid variable name in Pyret.

27.1.6 Data Definitions Pyret diverges from Racket (and even more so from Scheme) in its handling of data definitions. First, we will see how to define a structure:

RSW

Pyret

```
(define-struct pt (x y))
```

```
data Point:
```

```
  | pt(x, y)
```

```
end
```

This might seem like a fair bit of overkill, but we’ll see in a moment why it’s useful. Meanwhile, it’s worth observing that when you have only a single kind of datum in a data definition, it feels unwieldy to take up so many lines. Writing it on one line is valid, but now it feels ugly to have the `|` in the middle:

```
data Point: | pt(x, y) end
```

Therefore, Pyret permits you to drop the initial `|`, resulting in the more readable

```
data Point: pt(x, y) end
```

Now suppose we have two kinds of points. In the student languages of Racket, we would describe this with a comment:

```
; ; A Point is either
```

```
; ; - (pt number number), or
```

```
; ; - (pt3d number number number)
```

In Pyret, we can express this directly:

```
data Point:  
  | pt(x, y)  
  | pt3d(x, y, z)  
end
```

In short, Racket optimizes for the single-variant case, whereas Pyret optimizes for the multi-variant case. As a result, it is difficult to clearly express the multi-variant case in Racket, while it is unwieldy to express the single-variant case in Pyret.

For structures, both Racket and Pyret expose constructors, selectors, and predicates. Constructors are just functions:

RSW

Pyret

(pt 1 2)

pt(1, 2)

Predicates are also functions with a particular naming scheme:

RSW

Pyret

(pt? x)

is-pt(x)

and they behave the same way (returning true if the argument was constructed by that constructor, and false otherwise). In contrast, selection is different in the two languages (and we will see more about selection below, with `cases`):

RSW

Pyret

(pt-x v)

v.x

Note that in the Racket case, `pt-x` checks that the parameter was constructed by `pt` before extracting the value of the `x` field. Thus, `pt-x` and `pt3d-x` are two different functions and neither one can be used in place of the other. In contrast, in Pyret, `.x` extracts an `x` field of any value that has such a field, without attention to how it was constructed. Thus, we can use `.x` on a value whether it was constructed by `pt` or `pt3d` (or indeed anything else with that field). In contrast, `cases` does pay attention to this distinction.

27.1.7 Conditionals There are several kinds of conditionals in Pyret, one more than in the Racket student languages.

General conditionals can be written using `if`, corresponding to Racket's `if` but with more syntax.

RSW

Pyret

```
(if full-moon  
    "howl"  
    "meow")
```

```
if full-moon:  
    "howl"  
else:  
    "meow"  
end
```

RSW

Pyret

```
(if full-moon
```

```

"howl"
(if new-moon
    "bark"
    "meow"))
if full-moon:
    "howl"
else if new-moon:
    "bark"
else:
    "meow"
end

```

Note that `if` includes `else if`, which makes it possible to list a collection of questions at the same level of indentation, which `if` in Racket does not have. The corresponding code in Racket would be written

```

(cond
[full-moon "howl"]
[new-moon "bark"]
[else "meow"])

```

to restore the indentation. There is a similar construct in Pyret called `ask`, designed to parallel `cond`:

```

ask:
| full-moon then: "howl"
| new-moon then: "bark"
| otherwise:      "meow"
end

```

In Racket, we also use `cond` to dispatch on a datatype:

```

(cond
[(pt? v)  (+ (pt-x v) (pt-y v))]
[(pt3d? v) (+ (pt-x v) (pt-z v))])

```

We could write this in close parallel in Pyret:

```

ask:
| is-pt(v)   then: v.x + v.y
| is-pt3d(v) then: v.x + v.z
end

```

or even as:

```

if is-pt(v):
    v.x + v.y
else if is-pt3d(v):
    v.x + v.z
end

```

(As in Racket student languages, the Pyret versions will signal an error if no branch of the conditional matched.)

However, Pyret provides a special syntax just for data definitions:

```

cases (Point) v:
| pt(x, y)      => x + y
| pt3d(x, y, z) => x + z
end

```

This checks that `v` is a `Point`, provides a clean syntactic way of identifying the different branches, and makes it possible to give a concise local name to each field position instead of having to use selectors like `.x`. In general, in Pyret we prefer to use `cases` to process data definitions. However, there are times when, for instance, there many variants of data but a function processes only very few of them. In such situations, it makes more sense to explicitly use predicates and selectors.

27.1.8 Lists In Racket, depending on the language level, lists are created using either `cons` or `list`, with `empty` for the empty list. The corresponding notions in Pyret are called `link`, `list`, and `empty`, respectively. `link` is a two-argument function, just as in Racket:

RSW

Pyret

```
(cons 1 empty)
```

```
link(1, empty)
```

RSW

Pyret

```
(list 1 2 3)
```

```
[list: 1, 2, 3]
```

Note that the syntax `[1, 2, 3]`, which represents lists in many languages, is not legal in Pyret: lists are not privileged with their own syntax. Rather, we must use an explicit constructor: just as `[list: 1, 2, 3]` constructs a list, `[set: 1, 2, 3]` constructs a set instead of a list. In fact, we can create our own constructors and use them with this syntax.

Exercise

Try typing `[1, 2, 3]` and see the error message.

This shows us how to construct lists. To take them apart, we use `cases`. There are two variants, `empty` and `link` (which we used to construct the lists):

RSW

Pyret

```
(cond
  [(empty? l) 0]
  [(cons? l)
   (+ (first l)
      (g (rest l)))]
  cases (List) l:
  | empty      => 0
  | link(f, r) => f + g(r)
end
```

It is conventional to call the fields `f` and `r` (for “first” and “rest”). Of course, this convention does not work if there are other things by the same name; in particular, when writing a nested destructuring of a list, we conventionally write `fr` and `rr` (for “first of the rest” and “rest of the rest”).

27.1.9 First-Class Functions The equivalent of Racket’s `lambda` is Pyret’s `lam`:

RSW

Pyret

```
(lambda (x y) (+ x y))
```

```
lam(x, y): x + y end
```

27.1.10 Annotations

In student Racket languages, annotations are usually written as comments:

```
; square: Number -> Number  
; sort-nums: List<Number> -> List<Number>  
; sort: List<T> * (T * T -> Boolean) -> List<T>
```

In Pyret, we can write the annotations directly on the parameters and return values. Pyret will check them to a limited extent dynamically, and can check them statically with its type checker. The corresponding annotations to those above would be written as

```
fun square(n :: Number) -> Number: ...  
  
fun sort-nums(l :: List<Number>) -> List<Number>: ...  
  
fun sort<T>(l :: List<T>, cmp :: (T, T -> Boolean)) -> List<T>: ...
```

Though Pyret does have a notation for writing annotations by themselves (analogous to the commented syntax in Racket), they aren't currently enforced by the language, so we don't include it here.

27.1.11 What Else?

If there are other parts of Scheme or Racket syntax that you would like to see translated, please let us know.

27.2 Pyret vs. Python

For the curious, we offer a few examples here to justify our frustration with Python for early programming.

Python

Pyret

Python exposes machine arithmetic by default. Thus, by default, $0.1 + 0.2$ is not the same as 0.3 . (We hope you're not surprised to hear this.) Why this is the case is a fascinating subject of study, but we consistently find it a distraction for first-time programmers writing programs with arithmetic. And if we handwave the details of floating point aside, are we taking our claims of program reliability seriously?

Pyret implements exact arithmetic, including rationals, by default. In Pyret, $0.1 + 0.2$ really is equal to 0.3 . Where a computation must return an inexact number, Pyret does it explicitly: a key requirement in a curriculum built on reliability.

Python

Pyret

Understanding the difference between creating a variable and updating its value is a key learning outcome, along with understanding variables' scopes. Python explicitly conflates declaration with update, and has a tangled history with scope.

Pyret is statically scoped, and goes to great lengths—e.g., in the design of a query language for tables—to maintain it. There is no ambiguity in Pyret's syntax for working with variables.

Python

Pyret

Python has a weakly-defined, optional mechanism of annotations that was added late in the language's design, which conflates values and types.

Drawing on lessons learned from our several prior research projects on adding types to languages after-the-fact, Pyret was designed with typability from the start, with several subtle design choices to enable this. Pyret also has support (currently dynamic) for refinement-type annotations.

Python

Pyret

Python’s annotation mechanism has no notion of refinements.

To prepare students for modern programming languages with rich type systems, Pyret’s annotation syntax supports refinements. However, these are checked dynamically, so that students do not need to satisfy the vagaries of any particular proof assistant.

Python

Pyret

Python has weak built-in support for testing. While it has extensive professional libraries to test software, these impose a non-trivial burden on learners, as a result of which most introductory curricula do not use them.

First, a curriculum that proclaims reliability must put testing at its heart. Second, our pedagogy places heavy emphasis on the use of examples, and in particular the building-up of abstractions from concrete instances. For both these reasons, Pyret has extensive support in the language itself—not through optional, external libraries—for writing examples and tests, and provides direct language support for many of the interesting and tricky issues that arise when doing so.

Python

Pyret

Modern testing goes well beyond unit-tests. Furthermore, property-based testing is a very useful gateway to thinking about formal properties. In Python, this is only available through libraries.

Pyret has convenient language features—such as the use of **satisfies** rather than **is** in tests—to expose students to these ideas in lightweight ways.

Python

Pyret

State is ubiquitous in libraries.

State is an important but also complicated part of programming. Pyret nudges students to program without state while still permitting the full range of stateful programming. This comes with safeguards both linguistic (e.g., variables are immutable unless declared otherwise) and in output (e.g., mutable fields are displayed to alert the student that the value may change or may even have already changed).

Python

Pyret

Equality comparison is simplistic and in line with most other professional languages.

Equality is in fact subtle, and useful as a pedagogic device. Therefore, Pyret has a carefully-designed family of equality operators that are not only of practical value but also have pedagogic use.

Python

Pyret

Images are not values in the language. You can write a program to produce an image, but you can’t just view it in your programming environment.

Images are values. Pyret can print an image just like it can a string or a number (and why not?). Images are fun values, but they aren’t frivolous: they are especially useful for demystifying and explaining important but abstract issues like function composition.

Python

Pyret

The language doesn’t have a built-in notion of reactive programs.

Reactivity is a core concept in the language, and the subject of both design and implementation research.

Python

Pyret

Python's error messages are not added with novices as a primary audience.

Novices make many errors. They can be especially intimidated by error reports, and can feel discouraged about causing errors. Thus, Pyret's error messages are the result of nearly a decade of research. In fact, some educators have created pedagogic techniques that explicitly rely on the nature and presentation of information in Pyret's errors.

Python

Pyret

Python has begun to suffer from complexity creep that we believe serves professionals at the expense of novices. For example, the result of map in Python is actually a special generator value. This can lead to outcomes requiring extra explanation, like map(str, [1, 2, 3]) producing <map object at 0x1045f4940>. Type hints (discussed above) are another example.

Since Pyret's target audience is novice programmers programming in the style of this book, our primary goal when adding any feature is to preserve the early experience and avoid surprises.

Python

Pyret

Data definitions are central to computer science, but Python over-relies on built-in data structures (especially dictionaries) and makes user-defined ones unwieldy to create.

Pyret borrows from the rich tradition of languages like Standard ML, OCaml, and Haskell to provide algebraic datatypes, whose absence often forces programmers to engage in unwieldy (and inefficient) encoding tricks.

Python

Pyret

Python has several more rough corners that can lead to unexpected and undesirable outcomes. For instance, = sometimes introduces new variables and sometimes rebinds them. A function where a student forgot to return a value doesn't result in an error but silently returns None. Python has a complicated table that describes which values are true and which are false. And so on.

Pyret is designed from the ground-up to avoid all these problems.

27.3 Comparing This Book to HtDP This book (DCIC) is often compared to How to Design Programs (HtDP), from which it draws enormous inspiration. Here we briefly describe how the two books compare.

At a high level they are very similar:

- Both are built around the centrality of data structure. Both want to provide methods for designing programs. Both start with functional programming but transition to (and take very seriously) stateful imperative programming.
- Both are built around languages carefully designed with education in mind. The languages provide special support for writing examples and tests; error reporting designed for beginners; built-in images and reactivity. The languages eschew weird gotchas (in a way that Python does not: see Pyret vs. Python or, if you want to read much more, this paper).

and so on. To call these “similarities” is, however, a disservice. DCIC copied these ideas from HtDP; in some cases, HtDP even pioneered them.

Now for the differences. Note that they are differences now. Some ideas from DCIC are going to HtDP, and over time more may intermingle.

- The most obvious is that DCIC is in Pyret. HtDP has tons of good ideas, all ignored because it uses Racket, whose syntax some people (especially some educators) dislike. We built Pyret to embody good ideas we'd learned from the Racket student languages and other good ideas of our own, but package them in a familiar

syntax. But as you can see, the two languages are not actually that far apart: Pyret for Racketeers and Schemers.

- The next most obvious thing is that DCIC also includes Python. HtDP has a (not formally published) follow-up that teaches program design in Java. In contrast, we wanted to integrate the transition to Python into DCIC itself. There's much to be learned from the contrast! In particular, Pyret and its environment were carefully designed around pedagogic ideas for teaching state. Python was not, despite the ubiquity and difficulty of state! So there's a lot to be gained, when introducing state, to contrast them.
- Next, DCIC has a lot algorithmic content, whereas HtDP has almost none. DCIC covers, for instance, Big-O analysis [Predicting Growth]. It even has a section on amortized analysis [Halloween Analysis]. It goes up through some graph algorithms. This is far more advanced material than HtDP covers.

Those are most of the differences. They're visible (some even evident) from glancing through the table of contents. However, there is one very deep difference that will not be apparent to most readers, which we discuss below.

HtDP is built around a beautiful idea: the data structures shown grow in complexity in set-theoretic terms. Therefore it begins with atomic data, then has fixed-size data (structures), then unbounded collections (lists) of atomic data, pairs of lists, lists of structures, and so on. All built up, systematically, in a neat progression.

However, this has a downside. You have to imagine what the data represent (this number is an age, that string is a name, that list is of GDPs), but they're idealized. In a way the most real data are actually images! After that (which come early), all the data are "virtualized" and imaginary.

Our view is that the most interesting data are lists of structures. (Remember those? They're complicated and come some ways down the progression.) You might find this surprising; if so, we give you another name for them: tables. Tables are ubiquitous. Even companies process and publish them; even primary school students recognize and use them. They are perhaps our most important universal form of structured data.

Even better, lots of real-world data are provided as tables. You don't have to imagine things or make up fake GDPs like 1, 2, and 3. You can get actual GDPs or populations or movie revenues or sports standings or whatever interests you. (Ideally, cleansed and curated.) We believe that just about every student—even every child—is a nascent data scientist (at least when it's convenient to them). Even a child who says "I hate math" will often gladly use statistics to argue for their favorite actor or sportsperson or whatever. We just have to find what motivates them.

Buut there's a big catch! A key feature of HtDP is that for every level of datatype, it provides a Design Recipe for programming over that datatype. Lists-of-structs are complex. So is their programming recipe. And we want to put them near the beginning! Furthermore, the Design Recipe is dangerous to ignore. Students struggle with blank pages and often fill them up with bad code, which they then get attached to. The Design Recipe provides structure, scaffolding, reviewability, and much more. It's cognitively grounded in schemas.

So over the past few years, we've been working on different program design methods that address the same ends through different means. A lot of our recent education research has been putting new foundations in place. It's very much work in progress. And DCIC is the distillation of those efforts. As we have new results, we'll be weaving them into DCIC (and probably HtDP too). Stay tuned!

27.4 Release Notes This is a summary of updates made with each release of the book (excluding typos and other minor fixes).

Version 2023-02-21

This version has a sweeping set of changes:

- The book has been broken down into a collection of booklets, to give it a clearer structure and organization. See Organization of the Material for details.
- Several huge chapters in the earlier version have been broken down into smaller chapters and split across booklets.
- The Introduction to Programming material has been substantially revised and expanded.
- The ordering of materials has changed. The material on Algorithm Analysis has been moved to after the Python material.

- The From Pyret to Python transition has been improved substantially.
- There is now a chapter on using Pandas that builds off the corresponding Pyret example on tables.
- Python dictionaries have moved into From Pyret to Python. Even though we don't cover Pyret's dictionaries, having the dictionaries material come before state is a more natural flow with regards to students learning Python. We then develop the implementation of dictionaries using state in Hashes, Sets, and Key-Values.
- There is now a whole new unified approach to Programming With State that shows the Pyret and Python versions side-by-side. Seeing this material in two different languages helps focus on similarities and differences and can improve transfer between languages.
- Material that depends on algorithm analysis has now been separated from material that does not. Furthermore, in keeping with the book's theme, the focus is (again) on data structures. The result, in Data Structures with Analysis, refactors a lot of prior material to present it much more cleanly.
- Some material has further been refactored into Advanced Topics. In addition, there are several more new advanced goodies!

Version 2022-08-28

Besides numerous small improvements, we added some new bonus material.

Version 2022-01-25

- Consistently renamed the definitions and interactions window to the definitions and interactions pane.
- Moved the material on working with variables out of the intro to Python section and into the Programming with State section. Mutation of structured data moved before variable mutation within the Programming with State section.
- Added a comparison to How to Design Programs.
- The include line for the DCIC libraries at this version is

```
include shared-gdrive(
  "dcic-2021",
  "1wyQZj_L0qqV9Ekgr9au6RX2iqt2Ga8Ep")
```

Version 2021-08-21

The original release! Based on the prior book Programming and Programming Languages.

27.5 Glossary bandwidth

The bandwidth between two network nodes is the quantity of data that can be transferred in a unit of time between the nodes.

cache

A cache is an instance of a space-time tradeoff: it trades space for time by using the space to avoid recomputing an answer. The act of using a cache is called caching. The word “cache” is often used loosely; we use it only for information that can be perfectly reconstructed even if it were lost: this enables a program that needs to reverse the trade—i.e., use less space in return for more time—to do so safely, knowing it will lose no information and thus not sacrifice correctness.

coinduction

Coinduction is a proof principle for mathematical structures that are equipped with methods of observation rather than of construction. Conversely, functions over inductive data take them apart; functions over coinductive data construct them. The classic tutorial on the topic will be useful to mathematically sophisticated readers.

idempotence

An idempotent operator is one whose repeated application to any value in its domain yields the same result as a single application (note that this implies the range is a subset of the domain). Thus, a function

$\backslash(f\backslash)$ is idempotent if, for all $\backslash(x\backslash)$ in its domain, $\backslash(f(f(x))) = f(x)\backslash$) (and by induction this holds for additional applications of $\backslash(f\backslash)$).

invariants

Invariants are assertions about programs that are intended to always be true (“in-vary-ant”—never varying). For instance, a sorting routine may have as an invariant that the list it returns is sorted.

latency

The latency between two network nodes is the time it takes for packets to go between the nodes.

metasyntactic variable

A metasyntactic variable is one that lives outside the language, and ranges over a fragment of syntax. For instance, if we write “for expressions e_1 and e_2 , the sum $e_1 + e_2$ ”, we do not mean the programmer literally wrote “ e_1 ” in the program; rather we are using e_1 to refer to whatever the programmer might write on the left of the addition sign. Therefore, e_1 is metasyntax.

packed representation

At the machine level, a packed representation is one that ignores traditional alignment boundaries (in older or smaller machines, bytes; on most contemporary machines, words) to let multiple values fit inside or even spill over the boundary.

For instance, say we wish to store a vector of four values, each of which represents one of four options. A traditional representation would store one value per alignment boundary, thereby consuming four units of memory. A packed representation would recognize that each value requires two bits, and four of them can fit into eight bits, so a single byte can hold all four values. Suppose instead we wished to store four values representing five options each, therefore requiring three bits for each value. A byte- or word-aligned representation would not fundamentally change, but the packed representation would use two bytes to store the twelve bits, even permitting the third value’s three bits to be split across a byte boundary.

Of course, packed representations have a cost. Extracting the values requires more careful and complex operations. Thus, they represent a classic space-time tradeoff: using more time to shrink space consumption. More subtly, packed representations can confound certain run-time systems that may have expected data to be aligned.

parsing

Parsing is, very broadly speaking, the act of converting content in one kind of structured input into content in another. The structures could be very similar, but usually they are quite different. Often, the input format is simple while the output format is expected to capture rich information about the content of the input. For instance, the input might be a linear sequence of characters on an input stream, and the output might be expected to be rich and tree-structured according to some datatype: most program and natural-language parsers are faced with this task.

reduction

Reduction is a relationship between a pair of situations—problems, functions, data structures, etc.—where one is defined in terms of the other. A reduction R is a function from situations of the form P to ones of the form Q if, for every instance of P , R can construct an instance of Q such that it preserves the meaning of P . Note that the converse strictly does not need to hold.

space-time tradeoff

Suppose you have an expensive computation that always produces the same answer for a given set of inputs. Once you have computed the answer once, you now have a choice: store the answer so that you can simply look it up when you need it again, or throw it away and re-compute it the next time. The former uses more space, but saves time; the latter uses less space, but consumes more time. This, at its heart, is the space-time tradeoff. Memoization [Avoiding Recomputation by Remembering Answers] and using a cache are both instances of it.

type variable

Type variables are identifiers in the type language that (usually) range over actual types.

wire format

A notation used to transmit data across, as opposed to within, a closed platform (such as a virtual machine). These are usually expected to be relatively simple because they must be implemented in many languages and on weak processes. They are also expected to be unambiguous to aid simple, fast, and correct parsing. Popular examples include XML, JSON, and s-expressions.

contents ← prev up next →