Programming and Programming Languages

Shriram Krishnamurthi, Benjamin S. Lerner, Joe Gibbs Politz, Kathi Fisler

July 23, 2020

Contents

Chapter 1

Introduction

1.1 Our Philosophy

Many people would regard this as being two books in one. One book is an introduction to *programming*, teaching you basic concepts of organizing data and the programs that operate over them, ending in the investigation of universally useful algorithms. The other book is an introduction to *programming languages*: a study, from one level up, of the media by which we structure these data and programs.

Obviously, these are not unrelated topics. We learn programming through one or more languages, and the programs we write then become natural subjects of study to understand languages at large. Nevertheless, these are considered sufficiently different topics that they are approached separately. This is how we approached them, too.

The one noble exception to this separation is the best computer science book ever written, *The Structure and Interpretation of Computer Programs*.

We have come to realize that this separation is neither meaningful nor helpful. The topics are deeply intertwined and, by accepting that interleaving, the result is likely to be a much better book. This is my experiment with that format.

1.2 Predictability as a Theme

There are many ways to organize the study of programming and programming languages. My central theme is the concept of *predictability*.

Programs are typically static: they live on the moral equivalent of a paper, unmoving and unchanging. But when we run a program, it produces a complex, dynamic behavior that yields utility, pleasure, and (sometimes) frustration. Everyone who writes programs ultimately cares—whether they realize it or not—in *predicting* the latter from the former. Sometimes we even write programs to help us with this task (as we'll see in chapter 14, chapter 16, chapter 27, and elsewhere).

Predictability has a bad rap. Under the guise of "program reasoning", it came to be viewed simultaneously as both noble and mind-numbingly boring. It is certainly noble, but we will try to present it a way that will hopefully seem utterly natural, indeed entirely obvious (because we believe it is). Hopefully you'll come away from this study reasonably convinced about the central place of predictability in your own work, and as a metric for programming language design.

1.3 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.

At various points 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.

1.4 The Language of This Book

This book uses a new programming language called Pyret. Pyret is the outgrowth of our deep experience programming in and designing functional, object-oriented, and scripting languages, as well as their type systems, program analyses, and development environments.

The language's syntax is inspired by Python. It fits the niche missing in computer science education of a simple language that sheds both the strange corner-cases (of which there are many) of Python while adding important features that Python lacks for learning programming (such as algebraic datatypes, optional annotations on variables, design decisions that better enable the construction of development environments, and strong support for testing). Beginning programmers can 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.

Unlike Python, Pyret will enforce indentation rather than interpret it: that is, indentation will simply become another syntax well-formedness criterion. But that hasn't been implemented yet.

Chapter 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 Kleiman, Athyuttam Eleti, Benjamin S. Shapiro, Cheng Xie, Dave Lee, Ebube Chuba, Harrison Pincket, Igor Moreno Santos, Iuliu Balibanu, Jason Bennett, Jon Sailor, Josh Paley, Kelechi Ukadike, Kendrick Cole, Marc Smith, Raymond Plante, Samuel Ainsworth, Samuel Kortchmar, frodokomodo (on github).

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

> Dorai Sitaram, John Palmer, Kartik Singhal, Kathi Fisler, 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 the first half of this book, even if he wouldn't necessarily approve of what has come of his patient instruction.

The second half of this book is essentially a translation into Pyret of *Programming Languages: Application and Interpretation*, and owes thanks to all the people acknowledged there.

The chapter on chapter 13 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.

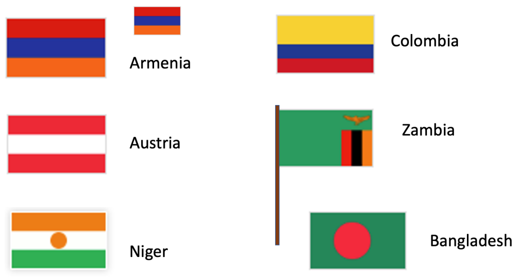
We thank cloudconvert for their free conversion tools.

Chapter 3

Getting Started

3.1 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!

3.2 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*.

If you are new to Pyret, go to code.pyret.org (which we'll henceforth refer to as "CPO").

For now, we will work only in the right Pyret window (the *interactions* window).

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 window 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.

3.3 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 window to change how the number is presented. Try it!

3.4 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**: *a 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*

3.5 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).

3.6 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).

Images are "optional", in the sense that some programs use them but many do not (where most programs use numbers and strings). When we want to use a feature that isn't common to most programs, we have to tell Pyret that we plan to use that feature (these are called "libraries"). For images, we do this by running the following at the prompt:

```
include image
```

Do Now!

Below the `include image`, write each of these Pyret expressions 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.

3.6.1 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.

3.6.2 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.

3.7 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.

3.7.1 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.

3.7.2 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.

3.7.3 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.

Chapter 4

Naming Values

4.1 The Definitions Window

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

The left window is called the *definitions window*. 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.

4.2 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 window is for: you can put expressions and programs in the definitions window, then use the "Run" button in CPO to make the definitions available in the interactions window.

Do Now!

Put the following in the definitions window:

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

Hit run, then enter `red-circ` in the interactions window. 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 window 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.

4.2.1 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. 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.

4.2.2 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. When you write a definition, you are telling Pyret to make an entry in its *directory* that associates names with values. Pyret evaluates the expression on the right side of the `=`, then stores the resulting value alongside the name. In contrast, if you write just an expression, you are asking Pyret to perform a computation and produce the result.

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 window, enter `blue-circ` at the prompt (you can re-enter than 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. When Pyret encounters a name in (or as) an expression, it goes to the directory and looks it up, returning the saved value.

This exercise highlights the difference between making a definition and using a defined name. These two tasks are akin to what happens with a human-language dictionary: at some point, someone made an entry in the dictionary for a word. When you use a dictionary, you are interested in retrieving the meaning or value of that word.

4.3 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 window contained only this most recent expression (and the `include image` statement). How many separate images would appear in the interactions window 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 section 3.6.2), 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.

Chapter 5

From Repeated Expressions to Functions

5.1 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:

```
# 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.

5.2 Defining Functions

In programming, a *function* takes one or more (configuration) *parameters* and uses them to produce a result. Specifically, the way we create a function is to

- Write down some examples of 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:

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

Do Now!

Why is there now 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.

- 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.

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 we won't even remember that we actually took step*s* to get from examples to the function: it'll become a single, natural *step*.

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 `#`.)

5.2.1 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 functiom (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`.

5.2.2 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-color :: String,
      mid-color :: String,
      bot-color :: String):
  frame(
    above(rectangle(120, 30, "solid", top-color),
      above(rectangle(120, 30, "solid", mid-color),
        rectangle(120, 30, "solid", bot-color))))
```

```
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-color :: String,
      mid-color :: String,
      bot-color :: String) -> Image:
  frame(
    above(rectangle(120, 30, "solid", top-color),
      above(rectangle(120, 30, "solid", mid-color),
        rectangle(120, 30, "solid", bot-color))))
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.

5.2.3 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 strip
  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.

5.3 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.

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
```

5.4 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 such examples—called *tests*—to make sure their programs are behaving as they expect.

5.5 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):
  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
```

For the time being, we won't worry about nonsensical situations like negative numbers of pens.

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?

Chapter 6

Conditionals and Booleans

6.1 Motivating Example: Shipping Costs

In section 5.5, 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.

6.2 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 section 6.3), 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.

6.3 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`. While there are an infinitely many values of type `Number`, there are only two of type `Boolean`: `true` and `false`.

Named for George Boole.

Exercise

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

Exercise

Why did we not enter `order-amt <= 10` at the interactions prompt to explore booleans?

6.3.1 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 [section 19.1].

```
››› 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 (almost; if you're on top of everything and want to read something more challenging, look at section 21.6.3); 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.

6.3.2 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
```

6.4 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 <Boolean expression to check>:
  <expression if first expression is true>
else if <another Boolean expression to check>:
  <expression if first expression is false and second expression is true>
else:
  <expression if both expressions are false>
end
```

A program can have multiple `if else` 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 a `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 window?

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 + 12
end
```

Both versons 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.

6.5 Evaluating by Reducing Expressions

In section 5.2.1, 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 first step is to *substitute* the `hours` with `45`.

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

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

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

=>  (40 * 10) + ((45 - 40) * 15)

=>  400 + (5 * 15)
=>  475
```

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

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

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

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).

6.6 Wrapping up: 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, "wow!"))
```

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

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

Both methods would produce the same answer.

Exercise

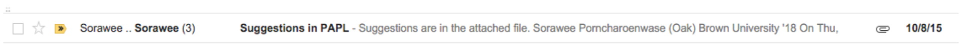
Manually evaluate each version. Where are the sequences of evaluation steps the same and where do they differ across these two programs?

Chapter 7

Introduction to Tabular Data

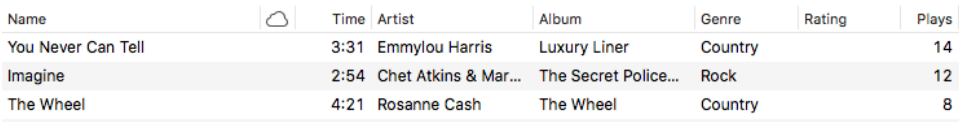
Many interesting data in computing are *tabular*—i.e., like a table—in form. First we'll see a few examples of them, before we try to identify what they have in common. Here are some of them:

- 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.



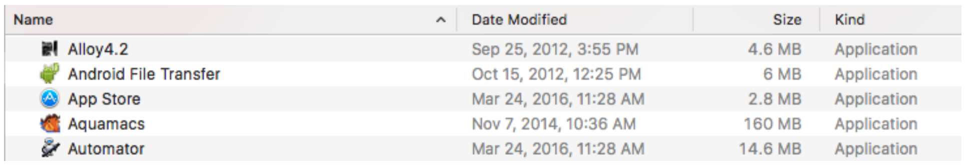
Sorawee .. **Sorawee** (3)	**Suggestions in PAPL** - Suggestions are in the attached file. Sorawee Porncharoenwase (Oak) Brown University '18 On Thu,		**10/8/15**

- 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	Plays
You Never Can Tell		3:31	Emmylou Harris	Luxury Liner	Country		14
Imagine		2:54	Chet Atkins & Mar...	The Secret Police...	Rock		12
The Wheel		4:21	Rosanne Cash	The Wheel	Country		8

- 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 consists of rows and columns. For instance, 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.
- 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 are usually 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.

7.1 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: "Alice", 30
  row: "Bob", 40
  row: "Carol", 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: "Alice", 30
    row: "Bob", 40
    row: "Carol", 25
  end
  is-not
  table: age, name
    row: 30, "Alice"
    row: 40, "Bob"
    row: 25, "Carol"
  end
end
```

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

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

```
people = table: name, age
  row: "Alice", 30
  row: "Bob", 40
  row: "Carol", 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, the language doesn't care where they came from.

7.2 Processing Rows

Let's now learn how we can actually process a table. Pyret offers a variety of built-in operations that make it quite easy to perform interesting computations over tables. In addition, as we will see later [chapter 8], if we don't find these sufficient, we can write our own. For now, we'll focus on the operations Pyret provides.

Let's think about some of the questions we might want to ask about our data:

- Which emails were sent by a particular user?
- Which songs were sung by a particular artist?
- Which are the most frequently played songs in a playlist?
- Which are the least frequently played songs in a playlist?

We see that some of these correspond to *keeping* some rows and some correspond to *ordering* them. Pyret provides tabular operations corresponding to these.

7.2.1 Keeping

Some of us have many more messages in our inbox!

Let's imagine we had a table that represented our inbox:

```
email = table: sender, recipient, subject
  row: 'Matthias Felleisen', 'Pedro Diaz', 'Introduction'
  row: 'Joe Politz', 'Pedro Diaz', 'Class on Friday'
  row: 'Matthias Felleisen', 'Pedro Diaz', 'Book comments'
  row: 'Mia Minnes', 'Pedro Diaz', 'CSE8A Midterm'
end
```

and we wanted to get a table of just the messages from Matthias.

We can keep rows from a table as follows:

```
sieve email using sender:
  sender == 'Matthias Felleisen'
end
```

says to use the `email` table, and specifically to employ the `sender` column. This operation processes every row of the table. In each row, `sender` refers to the value of the `sender` column of that row. The expression in the body (between `:` and `end`) must evaluate to a `Boolean`; if it is `true`, then Pyret keeps that row in the resulting table, otherwise it is discarded. The outcome of running this *query* is a fresh table with the same columns but only some (perhaps as few as none) of the rows; those rows that remain will be in the same order as in the original table.

In the same way, we can keep rows based on the artist:

```
sieve playlist using artist:
  (artist == 'Deep Purple') or (artist == 'Van Halen')
end
```

This shows that we can write complex expressions to select rows.

Exercise

Write a table for to use as `playlist` that works with the sieve expression above.

Exercise

Write a `sieve` expression on the `email` table above that would result in a table with zero rows.

7.2.2 Ordering

We can similarly order the rows of a table, which produces a new table that has the rows in the described order:

```
order playlist:
  play-count ascending
end
```

orders the rows with the `play-count` values in ascending order, so that the earliest rows in the table tell us which songs we've listened to least frequently.

Note that what goes between the `:` and `end` is *not* an expression. Therefore, we cannot write arbitrary code here. We can only name columns and indicate which way they should be ordered.

7.2.3 Combining Keeping and Ordering

Naturally, we are not limited to performing only one of these operations. Since each of them consumes a table and produces one, we can easily combine them. Let's first think of what we might want to do in English:

- Of the emails from a particular person, which is the oldest?
- Of the songs by a particular artist, which have we played the least often?

Do Now!

Take a moment to think about how you would write these with what you have seen so far.

Here is the first example:

```
mf-emails = sieve email using sender:
  sender == 'Matthias Felleisen'
end
order mf-emails:
  sent-date ascending
end
```

Note that in the `order` expression, we order not `email`, which is the table of all emails, but only `mf-email`, the table of just the emails from that one sender. Now, looking at the earliest rows in the result gives us the earliest emails from that one person.

Exercise

Write the second example as a composition of `keep` and `order` operations on a playlist table.

7.2.4 Extending

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:

```
extend employees using hourly-wage, hours-worked:
  total-wage: hourly-wage * hours-worked
end
```

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; as we will soon see, we can easily change the order of columns [section 7.2.6].

Naturally, we can combine extension with other table operations. For instance, we might have noticed that messages with short subject lines usually don't contain high-priority tasks. Therefore, we might first extend the email table with the length of the subject line:

```
ext-email = extend email using subject:
  subject-length: string-length(subject)
end
order ext-email:
  subject-length descending
end
```

This will create a table where the longest subject lines are at the top and the shortest subject lines are at the bottom.

7.2.5 Transforming, Cleansing, and Normalizing

There are times when a table is "almost right", but requires a little adjusting. For instance, we might have a table of customer requests for a free sample, and want to limit each customer to at most a certain number. We might get temperature readings from different countries in different formats, and want to convert them all to one single format. We might have a gradebook where different graders have used different levels of precision, and want to standardize all of them to have the same level of precision.

Because unit errors can be dangerous!

In all these cases, we want the resulting table to have the same "shape" as the original—the same columns, the same rows, in the same order—but with some of the column values transformed slightly. Pyret provides `transform` to do this. For instance, here is how we limit customer orders:

```
transform orders using count:
  count: num-min(count, 3)
end
```

Here's how we round the total grades:

```
transform gradebook using total-grade:
  total-grade: num-round(total-grade)
end
```

Of course, a transformation can involve columns other than the one being transformed, and can transform multiple columns:

```
transform weather using temp, unit:
  temp:
    if unit == "F":
      fahrenheit-to-celsius(temp)
    else:
      temp
    end,
  unit:
    if unit == "F":
      "C"
    else:
      unit
    end
end
```

This alters the table so that all temperatures are converted to celsius.

Do Now!

In this example, why do we also transform `unit`?

It's because we should keep the temperature and unit in sync. If we transform the temperature but not the unit, a later user of this table might assume the `unit` column is accurate, and accidentally treat the converted temperature as if it were still in Fahrenheit.

7.2.6 Selecting

Finally, for presentation purposes, it is sometimes useful to see just a few of the columns, especially in tables with many of them; it can also be helpful to change the order of columns so that items that are meant to be viewed together are made adjacent. Suppose our gradebook has numerous columns representing all the intermediate scores, at the end of which is the total; when we're done assigning grades, we want to see each student's name with just their final score:

```
select name, total-grade from gradebook end
```

Again, we can combine this operation with others. For instance, we may want to see just the artists and songs in our playlist, sorted in order by the artist's name:

```
ss = select artist, song from playlist end
order ss:
  artist ascending
end
```

7.2.7 Summary of Row-Wise Table Operations

We've seen a lot in a short span. Specifically, we have seen several operations that consume a table and produce a new one according to some criterion. It's worth summarizing the impact each of them has in terms of key table properties (where "-" means the entry is left unchanged):

Operation	**Cell contents**	**Row order**	**Number of rows**	**Column order**	**Number**
Keeping	-	-	*reduced*	-	-
Ordering	-	*changed*	-	-	-
Extending	existing unchanged, *new computed*	-	-	-	*augmente*
Transforming	*altered*	-	-	-	-
Selecting	-	-	-	*changed*	*reduced*

The italicized entries reflect how the new table may differ from the old. Note that an entry like "reduced" or "altered" should be read as *potentially* reduced or altered; depending on the specific operation and the content of the table, there may be no change at all. (For instance, if a table is already sorted according to the criterion given in an `order` expression, the row order will not change.) However, in general one should expect the kind of change described in the above grid.

Observe that both dimensions of this grid provide interesting information. Unsurprisingly, each row has at least some kind of impact on a table (otherwise the operation would be useless and would not exist). Likewise, each column also has at least one way of impacting it. Furthermore, observe that most entries leave the table unchanged: that means each operation has limited impact on the table, careful to not overstep the bounds of its mandate.

On the one hand, the decision to limit the impact of each operation means that to achieve complex tasks, we may have to compose several operations together. We have already seen examples of this earlier this chapter. However, there is also a much more subtle consequence: it also means that to achieve complex tasks, we *can* compose several operations and get exactly what we want. If we had fewer operations that each did more, then composing them might have various undesired or (worse) unintended consequences, making it very difficult for us to obtain exactly the answer we want. Instead, the operations above follow the *principle of orthogonality*: no operation shadows what any other operation does, so they can be composed freely.

As a result of having these operations, we can think of tables also algebrically. Concretely, when given a problem, we should again begin with concrete examples of what we're starting with and where we want to end. Then we can ask ourselves questions like, “Does the number of columns stay the same, grow, or shrink?”, “Does the number of rows stay the same or shrink?”, and so on. The grid above now provides us a toolkit by which we can start to decompose the task into individual operations. Of course, we still have to think: the order of operations matters, and sometimes we have to perform an operation mutiple times. Still, this grid is a useful guide to hint us towards the operations that might help solve our problem.

Chapter 8

From Tables to Lists

Previously [chapter 7] 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 commmon. 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 [chapter 3] 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.

8.1 Basic Statistical Questions

There are many more questions we might want to ask of our data. For instance:

- The most-played song in a playlist, which translates to the maximum value in a column of play counts.
- The largest file in a filesystem, which translates to the maximum value in a column of file sizes.
- The shortest person in a table of people, which translates to the smallest value in a column of heights.
- The number of songs in a playlist. (This is arguably a question about all the columns combined, not any one specific column, since they all have the same number of entries.)

- All the *distinct* entries in the play-counts column. (This, naturally, is a question about a specific column, because the number of distinct entries will differ depending on the column.)
- The *number of* distinct entries in the play-counts column.
- The average in a column of wages.
- Other statistics (the median, mode, standard deviation, etc.) in a column of heights.

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.

8.2 Extracting a Column from a Table

Hopefully you found `select` attractive, because it gives us a column in isolation: e.g.,

```
songs = table: title, artist, play-count
  row: "Harry Styles", "Adore You", 0
  row: "Blinding Lights", "The Weeknd", 5
  row: "Memories", "Maroon 5", 97
  row: "The Box", "Roddy Ricch", 25
end
select play-count from songs end
```

But in the end we're still stuck with a column *in a table*, and none of the other operations let us compute the answers we're looking for. Therefore, there is no (straightforward) way to express these questions at all, because they require us to be perform a computation looking at the values of a table *relative to one another*, rather than *in isolation*.

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 [chapter 10], 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 data in programs without the bother of having to create a table every single time.

Therefore, we introduce one more operation, `extract`, which takes a column name and gives just the content of that one column:

```
extract play-count from songs end
```

And now we can answer the critical question—what is the difference between `select` and `extract`—by saying that while `select` produces a table, `extract` produces a list.

8.3 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.

8.3.1 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).

8.3.2 Creating Literal Lists

We have already seen how we can create lists from a table, using `extract`. 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).

8.4 Operating on Lists

8.4.1 Built-In Operations on Lists

Pyret handily provides a useful set of operations we can already perform on lists. As you might have guessed, we can already compute most of the answers we've asked for above. First we need to include some libraries that contain useful functions:

```
include math
include statistics
```

We can then access several useful functions:

- `max` computes the maximum element of a list.

- `min` computes the minimum element of a list.
- `mean` computes the average of a list.
- `stdev` computes the standard deviation of the values in list.

Thus:

```
pcs = extract play-count from songs end
most-played-count = max(pcs)
least-played-count = min(pcs)
```

would compute the highest and lowest play count from a table of songs.

Similarly, we could get the tallest and shortest heights from a table of people:

```
hts = extract height from people end
tallest-height = max(hts)
shortest-height = min(hts)
```

Exercise

Design a table that has three people and would produce `78` and `42` for the `tallest-height` and `shortest-height` with the example code above.

8.4.2 Combining Lists and Tables

Note that the questions we originally asked were slightly different: we didn't ask for the tallest height but the tallest *person*, or likewise the most most-played *song*. Because we've stripped the heights and counts of their surrounding context, we can no longer tell which person or song these values correspond to. For that, we have to go back to the table.

Do Now!

Do you see how we can use the values above, like `most-played-count` or `shortest-height`, to obtain the corresponding songs or people?

The key is to write a query over the corresponding table that refers to this value. For instance:

```
most-played-songs = sieve songs using play-count:
  play-count == most-played-count
end
```

```
tallest-people = sieve people using height:
  height == tallest-height
end
```

There's a reason we are careful to always use the plural—*`people`*, *`songs`*—rather than the singular. This is because we cannot be sure there is only one person or one song with this height or play count. That is, there is a single biggest or smallest value in the list, because the value has no other information about it (so the same height coming from two different people, or the same play count coming from two different songs, looks the same in the list). But when put back in the context of the original table, the other values may be different.

In short, our overall answer is computed quite simply:

```
pcs = extract play-count from songs end
most-played-count = max(pcs)
sieve songs using play-count:
  play-count == most-played-count
end
```

and

```
hts = extract height from people end
tallest-height = max(hts)
sieve people using height:
  height == tallest-height
end
```

Exercise

Implement all the other statistical questions posed in section 8.1.

Until now we've only seen how to use built-in functions over lists. Next [chapter 9], 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!

Chapter 9

Processing Lists

We have already seen [chapter 8] 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. As we already briefly discussed earlier [section 8.3.1],

- 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! Our mission is to understand and internalize this process.

9.1 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 head is `1`, and the rest of it is `[list: 2, 3]`. For this second list, the head is `2` and the rest is `[list: 3]`.

Do Now!

Take apart this third list.

For the third list, the head is `3` and the rest is `[list: ]`, i.e., the empty list. In Pyret, we have another way of writing the empty list: `empty`.

Here, we've taken the lists apart manually. Naturally, Pyret has operations that let us do that. Lists are an instance of *structured data*, and in general there are two ways to take apart structured data: using `cases`, which we will see below, and using *accessors*. A list has two accessors: `first` and `rest`. We use an accessor by writing an expression, followed by a dot (`.`), followed by the accessor's name. Thus:

```
l1 = [list: 1, 2, 3]
h1 = l1.first
l2 = l1.rest
h2 = l2.first
l3 = l2.rest
h3 = l3.first
l4 = l3.rest

check:
  h1 is 1
  h2 is 2
  h3 is 3
  l2 is [list: 2, 3]
  l3 is [list: 3]
  l4 is empty
end
```

Thus, `.first` and `.rest` give us a way to take apart a list. Can we also put together a list piece-by-piece? This would be especially useful for building up a list. And indeed we can: the function (called a *constructor*) that makes lists is called `link`. It takes two arguments: a list element, and the rest of the list. Thus, `l1` above is equivalent to a series of `link`s followed by `empty`:

```
check:
  [list: 1, 2, 3] is link(1, link(2, link(3, empty)))
end
```

Obviously, writing the `link` form is not very convenient to humans. But it will prove very valuable to programs!

Observe, in summary, that broadly speaking we have two kinds 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, 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, . . .).

9.2 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. 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.

- Compute the length of a list:

  ```
  my-len :: List<Any> -> Number
  ```

- Compute the sum of a list (of numbers):

  ```
  my-sum :: List<Number> -> Number
  ```

- Compute the maximum of a list (of numbers or strings):

  ```
  my-max :: List<Any> -> Any
  ```

- Given a list of strings, convert each string to a number representing its length:

  ```
  my-str-len :: List<String> -> List<Number>
  ```

- Given a list of numbers, generate a list of its positive numbers:

  If you want to be pedantic: its positive numbers with the same count and in the same order.

  ```
  my-pos-nums :: List<Number> -> List<Number>
  ```

- Given a list of numbers, replace every element with the running sum, i.e., the sum of all the elements from the beginning of the list until that element (inclusive):

  ```
  my-running-sum :: List<Number> -> List<Number>
  ```

- Given a list, keep every alternate element in it, starting from the first:

  ```
  my-alternating :: List<Any> -> List<Any>
  ```

- Given a list of numbers, compute the average of the numbers:

  ```
  my-avg :: List<Number> -> List<Number>
  ```

To solve problems like this, there are two things we should do:

- Construct examples of the function's behavior.
- Employ the *template* that suggests possible solutions.

Both steps sound simple but have several nuances, which we will explore.

9.3 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.

9.3.1 `my-len`: Examples

We have'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
```

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. That is,

```
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
```

That is, we can use the result of computing `my-len` on the rest of the list to compute the answer for the entire list.

Double-check all these and make sure you understand the calculations. It'll prove central to how we write the program later!

9.3.2 `my-sum`: Examples

A similar logic applies to how we treat a function like `my-sum`. What do we want the sum of the empty list to be? Well, it may be entirely clear, so let's move on for a moment. What is the sum of the list `[list: 7, 8, 9]`? Well, clearly we intend for this to be `24`. Let's see how that works out.

Setting aside the empty list for a moment, here are sums we can agree upon:

```
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 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:     ])
```

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?

From this, we can see that the sum of the empty list must be `0`:

```
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.

9.3.3 From Examples to Code

Given these examples, we can now turn them into code. We introduce the construct `cases`, which lets us tell apart different kinds of lists, and use it to provide answers for each kind of list.

The grammar for `cases` for lists is as follows:

```
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`.

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`).

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!

9.4 Structural Problems with List Answers

Now let's tackle the functions that *produce* a list as the answer.

9.4.1 `my-str-len`: Examples and Code

As always, we'll begin with some examples. Given a list of strings, we want the lengths of each string (in the same order). 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 how 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
```

Note that for brevity we're written the answers of converting each string (`2`, `5`, and `6`), each of which we obtain by applying `string-length` to the first element of the list at each point. Therefore, we can formulate a solution from this:

```
fun my-str-len(l):
  cases (List) l:
    | empty => empty
    | link(f, r) =>
      link(string-length(f), my-str-len(r))
  end
end
```

9.4.2 `my-pos-nums`: Examples and Code

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: ]
```

That is, when the first element is positive we `link` it into the result of computing `my-pos-nums` on the rest of the list; when the first element is negative, the result is just that of computing `my-pos-nums` on the rest of the list. 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!

9.4.3 `my-alternating`: First Attempt

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, and we have more thinking to do. We'll return to this later [section 9.5.3 and also section 9.7.2].

9.4.4 `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 `link`ing 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.)

We'll return to this function later, too [section 9.7.1].

9.5 Structural Problems with Sub-Domains

9.5.1 `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).

9.5.2 `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 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 on:

```
fun my-max(l):
  cases (List) l:
    | empty      => raise("not defined for empty lists")
    | link(f, r) =>
      cases (List) r:
        | empty => f
        |...
      end
  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
```

And sure enough, this definition does the job!

9.5.3 `my-alternating`: Examples and Code

Looking back at section 9.4.3, we can see that every *alternate* example is one we want. The problem is, to get from one example to the one two below, we have to remove *two* elements, not just one. That is, we have to pretend our list has elements in pairs, not singles. In terms of examples, this would look as follows:

```
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:             5, 6]) is [list:
5]
```

```
my-alternating([list:                 ]) is [list:
]
```

Now it's pretty easy to see how to construct a program: keep the first element, skip the second, and repeat. Let's see how far we can get using the template:

```
fun my-alternating(l):
  cases (List) l:
    | empty => empty
    | link(f, r) =>
      link(f,...r..)
  end
end
```

Do Now!

Think about how to complete this definition.

Before we proceed, there is a small problem: our example is not good enough to cover all the cases we'll encounter. Specifically, to traverse by two we must *have* two elements, but we might not: the list might have only an odd number of elements. That is, we might instead have

```
my-alternating([list: 1, 2, 3, 4, 5]) is [list: 1, 3, 5]
my-alternating([list:       3, 4, 5]) is [list:    3, 5]
my-alternating([list:             5]) is [list:       5]
```

What this means is: *We won't always terminate with the empty list.* We have to be prepared to terminate with a list of one element. This suggests how we can complete the definition:

```
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
```

In section 9.7.2 we'll see another way of approaching this problem.

9.6 More Structural Problems with Scalar Answers

9.6.1 `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 [section 9.5], 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 may not 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.

9.7 Structural Problems with Accumulators

Now we are ready to tackle the problems we've left unfinished. They will require a new technique to solve.

9.7.1 `my-running-sum`: Examples and Code

Recall how we began in section 9.4.4. 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:

```
my-rs( 0, [list: 1, 2, 3, 4, 5]) is [list:  0 + 1] + my-rs( 0 + 1, [list: 2, 3
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. There are multiple reasons for this. [FILL]

9.7.2 `my-alternating`: Examples and Code

Recall our effort in section 9.4.3, which we tackled in section 9.5.3. There, we solved the problem by thinking of the list a little differently: we try, as much as possible, to skip two elements of the list at a time, so the first element we see is one we always want to keep as part of the answer. Here we will see another way to think about the same problem.

Return to the examples we've already seen [<*alternating-egs-1*>]. As we've already noted [section 9.5.3], in effect we want the output from every alternate example. One option was to traverse the list essentially two elements at a time. Another 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
```

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?

9.8 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.

9.8.1 `uniq`: Problem Setup

Consider the task of writing `uniq`: given a list of values, it produces a collection of the same elements while avoiding any duplicates (hence `uniq`, short for "unique").

`uniq` is the name of a Unix utility with similar behavior; hence the spelling of the name.

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.

9.8.2 `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 example we've worked out above.

9.8.3 `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.

9.8.4 `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-rec(r)
      if r.member(f):
        ur
      else:
        link(f, ur)
      end
  end
end
```

While it may seem that we have merely avoided repeating an expression, by moving the computation `uniq-rec(r)` to before the conditional, we have actually changed the program's behavior in a subtle way. We will discuss this later when we get to [REC tail calls].

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-rec(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 [chapter 16] 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 chapter 17.

9.8.5 `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 complicated even writing down examples (give it a try!).

9.8.6 `uniq`: Why Produce a List?

If you go back to the original statement of the `uniq` problem [section 9.8.1], 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 [chapter 17], 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).

9.9 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`?

Technically, elements that are also comparable.

Rather, what we mean is a list where all the elements are *of the same kind*, 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.

Chapter 10

Introduction to Structured Data

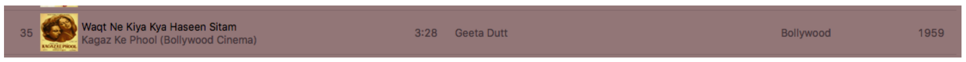
Earlier we had our first look at types. Until now, we have only seen the types that Pyret provides us, which is an interesting but nevertheless quite limited set. Most programs we write will contain many more kinds of data.

10.1 Understanding the Kinds of Compound Data

10.1.1 A First Peek at Structured Data

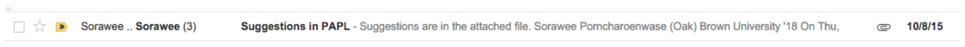
There are times when a datum has many *attributes*, or parts. We need to keep them all together, and sometimes take them apart. For instance:

- 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.



35 Waqt Ne Kiya Kya Haseen Sitam
Kagaz Ke Phool (Bollywood Cinema) 3:28 Geeta Dutt Bollywood 1959

- Your GMail application contains a bunch of information about a single message: its sender, the subject line, the conversation it's part of, the body, and quite a bit more.



Sorawee .. **Sorawee** (3) **Suggestions in PAPL** - Suggestions are in the attached file. Sorawee Porncharoenwase (Oak) Brown University '18 On Thu, **10/8/15**

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*.

10.1.2 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:

Yes, in some countries there are different or more colors and color-combinations.

- A traffic light can be in different states: red, yellow, or green. 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.

10.2 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.

10.2.1 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`. The way we actually make one of these data is by calling `song` with three parameters; for instance:

We follow a convention that types always begin with a capital letter.

<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)
```

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".

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)
```

10.2.2 Annotations for Structured Data

Recall that in [section 5.2.2] 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
```

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.

10.2.3 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
```

We can make examples of them as you would expect:

“In Texas, there ain’t nothin’ in the middle of the road except yellow stripes and a dead armadillo.”—Jim Hightower

```
b1 = boa("Alice", 10)
b2 = boa("Bob", 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. When we get to refinements [REF] we can recapture this distinction if we really want it.

```
b1 :: Animal = boa("Alice", 10)
b2 :: Animal = boa("Bob", 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.

10.3 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.

10.3.1 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(wnkkhs) is 57
end
```

10.3.2 Telling Apart Variants of Conditional Data

Now let's see how we tell apart variants. For this, we have to introduce another new piece of Pyret syntax: `cases`. A `cases` expression has several branches:

exactly as many as there are in the data definition. Each branch corresponds to one 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
```

Note that `cases` is followed by the name of the conditionally-defined type in parentheses (here, `TLColor`), and then an expression that computes a value of that type (in this case, `c` is already bound to such a value). Each variant is followed by `=>`, and then an expression that computes an answer for that variant.

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)`.

10.3.3 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, l) => ...
    | armadillo(n, l) => ...
  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, l) => n
    | armadillo(n, l) => n
  end
where:
  animal-name(b1) is "Alice"
  animal-name(b2) is "Bob"
  animal-name(a1) is "Glypto"
end
```

Chapter 11

Collections of Structured Data

As we were looking at structured data [chapter 10], 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: 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:

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*.

- 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 *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, may 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 [section 11.2].

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 [section 19.1] can involve more subtlety.

We have already seen tables [chapter 7], which are a form of collective, structured data. Now we will look at a few more, and how to program them.

11.1 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 [section 10.2.1], we can make a list of songs:

```
song-list = [list: lver, so, wnkkhs]
```

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 [chapter 9] 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, wnkkhs]) is lvar
oldest-song([list:       so, wnkkhs]) is wnkkhs
oldest-song([list:           wnkkhs]) is wnkkhs
oldest-song([list:                 ]) is ???
```

What do we write in the last case? Recall that we saw this problem earlier [section 9.5.1]: 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
```

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
```

11.2 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. Later [chapter 17] we will see how we can build sets for ourselves.

First, we can define sets just as easily as we can lists:

```
song-set = [set: lver, so, wnkkhs]
```

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 = [set: so, wnkkhs, 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 = [set: lver, so, wnkkhs, 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.

11.2.1 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 from the `pick` library). 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 :: Set):
  cases (Pick) s.pick():
    | pick-none => error("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?

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!

No you don't! 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.

11.2.2 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` [section 9.2]:

```
fun my-set-size(shadow 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.

11.3 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?

Chapter 12

Recursive Data

Sometimes, a data definition has a piece that refers back to itself. For example, a linked list of numbers:

```
data NumList:
  | nl-empty
  | nl-link(first :: Number, rest :: NumList)
end
```

Moving on to defining examples, we can talk about empty lists:

The `nl-` stands for `NumList`. This avoids clashing with Pyret's `empty`.

```
nl-empty
```

We can represent short lists, like a sequence of two `4`'s:

```
nl-link(4, nl-link(4, nl-empty))
```

Since these are created with constructors from `data`, we can use `cases` with them:

```
    cases (NumList) nl-empty:
      | nl-empty => "empty!"
      | nl-link(first, rest) => "not empty"
    end

=>  "empty!"

    cases (NumList) nl-link(1, nl-link(2, nl-empty)):
      | nl-empty => "empty!"
      | nl-link(first, rest) => first
    end
```

```
=>  1
```

This style of data definition permits 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))))
```

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)))))
  contains-3(nl-link(1, nl-link(2, nl-link(5, nl-link(4, nl-empty)))))
end
```

Next, we need to fill in the body with the template for a function over `NumList`s. We can start with the analogous template using `cases` we had before:

```
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
```

An empty list doesn't contain the number `3`, surely, so 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
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
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
```

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
```

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 positions.

The simple design recipe dictated this as the template:

```
#|
fun num-list-fun(nl :: NumList) -> ???:
  cases (NumList) nl:
    | nl-empty => ...
    | nl-link(first, rest) =>
      ... first ...
      ... rest ...
  end
end
|#
```

However, this template doesn't give much guidance with what to do with the `rest` field. We will extend the template with the following rule: each self-recursive position in the data definition becomes a self-recursive function call in the template. So the new template looks like:

```
#|
fun num-list-fun(nl :: NumList) -> ???:
  cases (NumList) nl:
    | nl-empty => ...
    | nl-link(first, rest) =>
      ... first ...
      ... num-list-fun(rest) ...
  end
end
|#
```

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.

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 `NumList`s, 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.

Chapter 13

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:

- http://world.cs.brown.edu/1/projects/flight-lander/v9-success.swf

  The airplane comes in to land succcessfully.
- http://world.cs.brown.edu/1/projects/flight-lander/v9-collide.swf

  Uh oh—the airplane collides with a balloon!
- http://world.cs.brown.edu/1/projects/flight-lander/v9-sink.swf

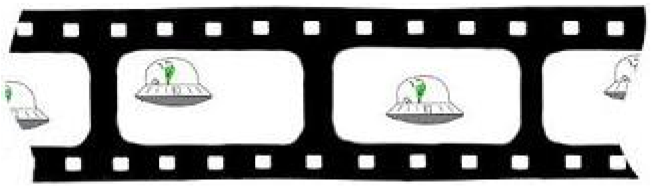
  Uh oh—the airplane lands in the water!

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.

13.1 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.

13.2 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 to 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`.

13.3 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:

http://world.cs.brown.edu/1/projects/flight-lander/v1.swf

First, here's an image of an airplane:

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:

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.

```
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.

13.3.1 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!

13.3.2 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
```

13.3.3 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?

13.4 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! Instead, when the airplane is about to go past the right edge of the screen, we'd

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.

like it to reappear on the left by a corresponding amount: “wrapping around”, as it were.

Here’s the video for this version:

http://world.cs.brown.edu/1/projects/flight-lander/v2.swf

Let’s think about what we need 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 tricker 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?

13.5 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 a airplane that continuously descends. Here's a video:

http://world.cs.brown.edu/1/projects/flight-lander/v3.swf

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.

13.5.1 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.

13.5.2 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.

13.5.3 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`.

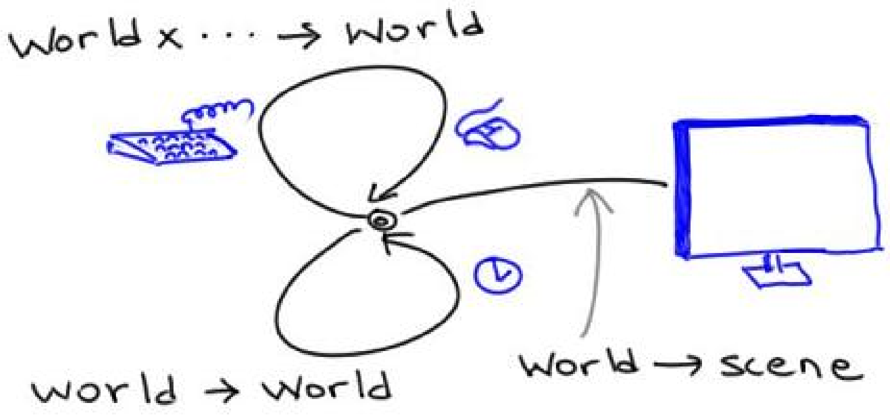
13.6 Version: Responding to Keystrokes

Now that we have the airplane descending, there's no reason it can't ascend as well. Here's a video:

http://world.cs.brown.edu/1/projects/flight-lander/v4.swf

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!

13.7 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:

http://world.cs.brown.edu/1/projects/flight-lander/v5.swf

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`.

13.8 Version: A Fixed Balloon

Now let's add a balloon to the scene. Here's a video of the action:
http://world.cs.brown.edu/1/projects/flight-lander/v6.swf

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 `posn`s 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 `posn`s 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
```

13.9 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 response 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 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.

13.10 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:

http://world.cs.brown.edu/1/projects/flight-lander/v8.swf

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.

13.11 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. Afer random periods of time, the wind blows with random speed and direction, causing the ballooons to move laterally.

Chapter 14

Examples, Testing, and Program Checking

When we think through a problem, it is often useful to write down *examples* of what we are trying to do. For example (see what we did there?), if we're asked to compute the [FILL]

There are, of course, 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.

14.1 From Examples to Tests

failure of tests can be due to

- the program being wrong - the example itself being wrong

when we find a bug, we

- find an example that captures the bug - add it to the program's test suite so that if we make the same mistake again, we will catch it right away

14.2 More Refined Comparisons

Sometimes, a direct comparison via `is` isn't enough for testing. We saw `raises` in the last section for testing errors. However, when doing some computations, especially involving math with approximations, we want to ask a different question. 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.

14.3 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 1.75^2 is clearly not 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.

14.4 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.

Consider the following problem: given a word (such as a name), we would like to spell it using the symbols of atoms (ignoring upper- and lower-case). This function, call it `elemental`, consumes a string and produces a list of strings such that

- each string in the output is an atomic symbol, and

- the concatenation of the strings in the output yields the input.

For instance, consider my name; it can be spelled as `[list: "S", "H", "Ri", "Ra", "M"]` (for [FILL], respectively). Thus we would write:

```
check:
  elemental("Shriram") is [list: "S", "H", "Ri", "Ra", "M"]
end
```

Now consider another example: [FILL]. We can clearly see that this breaks down as

```
check:
  elemental("...") is [list: ...]
end
```

14.5 Testing Erroneous Programs

- use RAISES to check erroneous code

Chapter 15

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.

15.1 A Little Calculus

If you've studied the differential calculus, you've come across curious sytactic statements such as this:

$$\frac{d}{dx} x^2 = 2x$$

Let's unpack what this means: the 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 square(x :: Number) -> Number: x * x end
fun double(x :: Number) -> Number: 2 * x end
```

and what we're really trying to say is that the d/dx (whatever that is) of `square` is `double`.

We're assuming functions of arity one in the variable that is changing.

So now let's unpack d/dx, starting with its type. As the above example illustrates, 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 [section 24.1].

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

$$\frac{f(x+\epsilon)-f(x)}{\epsilon}$$

as ϵ goes to zero in the limit. For now we'll give the infinitesimal a small but fixed value, and later [section 15.4] see how we can improve on this.

```
epsilon = 0.001
```

Let's now try to translate the above formula into Pyret:

```
fun d-dx(f :: (Number -> Number)) -> (Number -> Number):
  (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 :: (Number -> Number)) -> (Number -> Number):
  lam(x :: Number) -> Number:
    (f(x + epsilon) - f(x)) / epsilon
  end
end
```

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-square = d-dx(square)

check:
  ins = [list: 0, 1, 10, 100]
  for map(n from ins):
    num-floor(d-dx-square(n))
  end
  is
  for map(n from ins):
    num-floor(double(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 section 16.7,

$$\frac{d}{dx}[x \rightarrow x^2] = [x \rightarrow 2x]$$

Pity math textbooks for not wanting to tell us the truth!

15.2 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”.

15.3 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. In section 26.3.6 we see one direction of this: parameterized functions that are used immediately, so that we employ only abstraction and not delay. Below, we will see the other: delay without abstraction.

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)
```

To understand more about recursive definitions, see section 21.3.2. 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 [section 15.2] 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`, `filter`, and `fold` 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 [REF].

15.4 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.

Chapter 16

Predicting Growth

We will now commence the study of determining how long a computation takes. We'll begin with a little (true) story.

16.1 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.

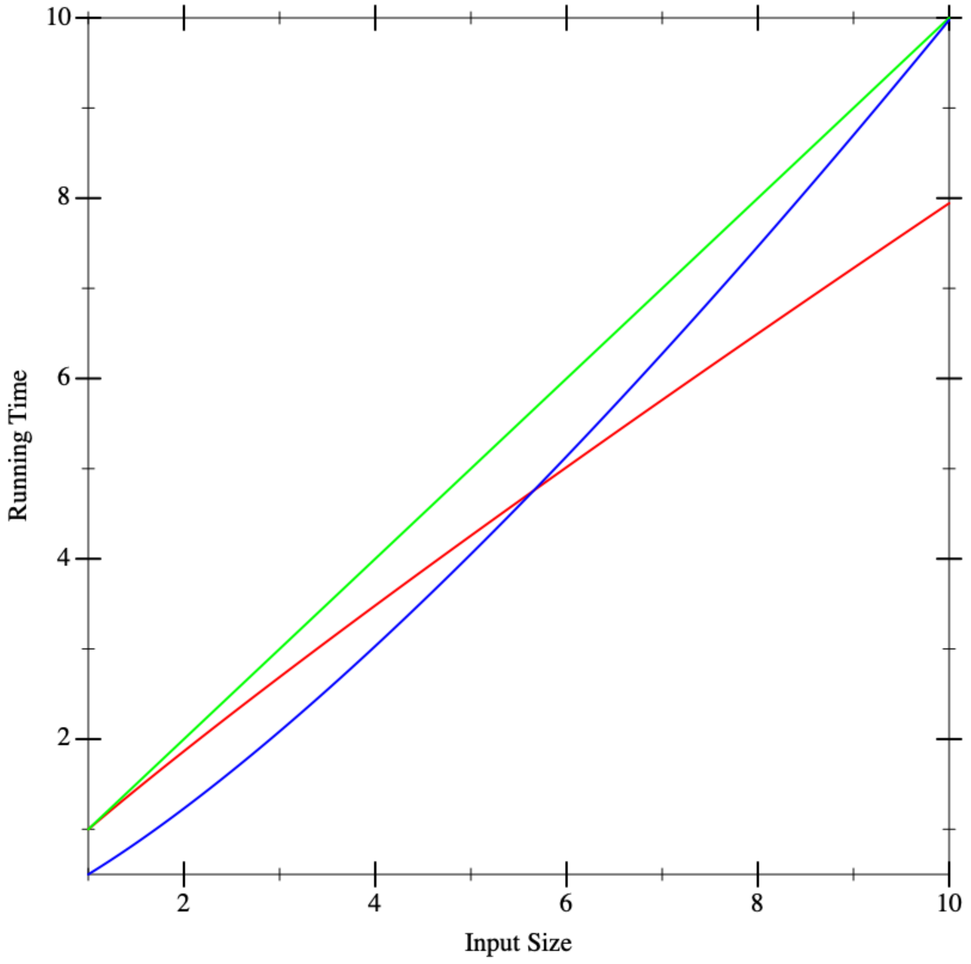
We therefore got only a sampling of their full data, and from this had to make some prediction on how long it would take to run the analytics on subsets (e.g., those corresponding to just one region) or all of their data set, both today and as it grew over time.

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.

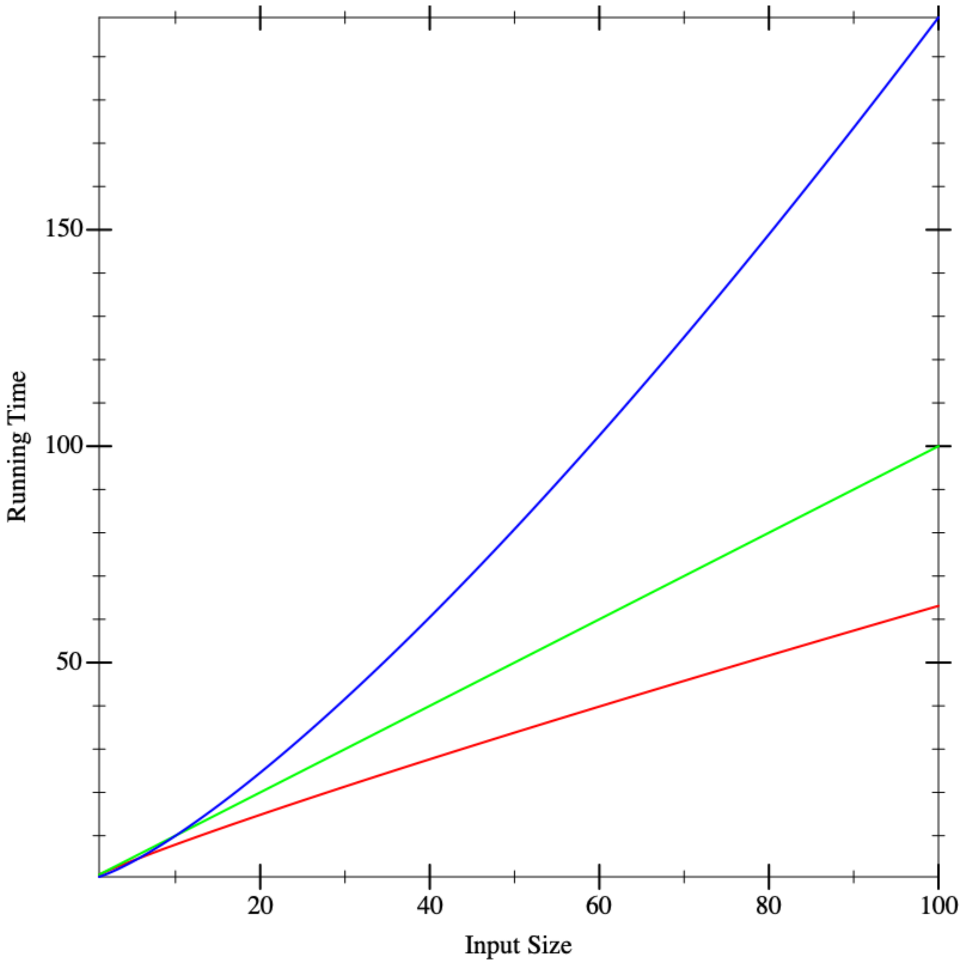
From this graph we have a good bet at guessing how long the tool would take on a dataset of 50,000. It's much harder, however, to be sure how long it would take on datasets of size 1.5 million or 3 million or 10 million. We've already explained why we couldn't get more data from the company. So what could we do?

These processes are respectively called *interpolation* and *extrapolation*.

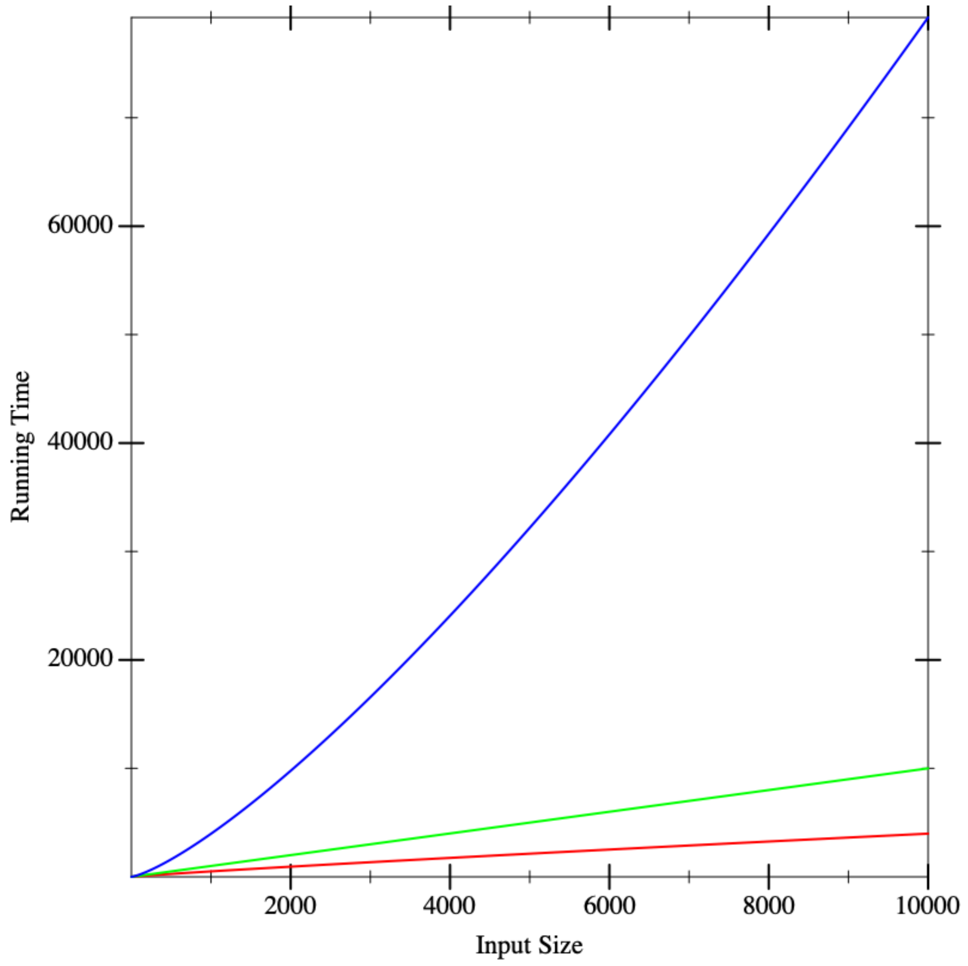
As another problem, suppose we have multiple implementations available. When we plot their running time, say the graphs look like this, with red, green, and blue each representing different implementations. On small inputs, suppose the running times look like this:



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 now do.

16.2 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. 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.

"Analytical" means applying algebraic and other mathematical methods to make predictive statements about a process without running it.

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 add up these costs for all the operations. This gives us a total cost for the program.

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.

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?

16.3 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 [section 16.8].

Of course, it is instructive—after carefully settting 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.

16.4 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*: n digits in base b can represent b^n numbers. Though irrelevant here, when numbers are central—e.g., when testing primality—the difference becomes critical! We will return to this briefly later [section 22.3.1.3].

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 `link`s in the list. We could also define it to be twice as large, to account for both the `link`s and the individual numbers (but as we'll see [section 16.8], 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 [section 9.9], 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*.

16.5 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 [section 16.6].

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. 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.

This idea is due to Prabhakar Ragde.

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. **TotQ**: 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. **TotA**: 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 section 16.6, 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 k (where we are only counting the number of `link`s 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., $k+1$ times. The total cost of the question is thus $3(k+1)$. 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 $3k+4$ units.

Now for the second row. The question again costs three units, and is evaluated k 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 `1 +`, one more to evaluate `len`...and *no more*, because we are ignoring the time spent in the recursive call itself. In short, it takes seven units of time (in addition to the recursion we've chosen to ignore).

In tabular form:

|Q|	#Q	TotQ	|A|	#A	TotA	Total
3	$k+1$	$3(k+1)$	1	1	1	$3k+4$
3	k	$3k$	7	k	$7k$	$10k$

Adding, we get $13k+4$. Thus running `len` on a k-element list takes $13k+4$ 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 k?

16.6 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, T, to compute an upper-bound of the time of `f`. T takes as many parameters as `f` does. The parameters to T represent the sizes of the corresponding arguments to `f`. Eventually we will want to arrive at a *closed form* solution to T, i.e., one that does not refer to T itself. But the easiest way to get there is to write a solution that is permitted to refer to T, called a *recurrence relation*, and then see how to eliminate the self-reference [section 16.10].

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 T.

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 T 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 T on the *size* of `a`.

For conditionals, we use only the $|\mathbf{Q}|$ and $|\mathbf{A}|$ 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 T. Thus, if we were doing this for `len` above, we would define $T(k)$—the time needed on an input of length k—in two parts: the value of $T(0)$ (when the list is empty) and the value for non-zero values of k. We know that $T(0) = 4$ (the cost of the first conditional and its corresponding answer). If the list is non-empty, the cost is $T(k) = 3 + 3 + 7 + T(k-1)$

(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:

$$T(k) = \begin{cases} 4 & \text{when } k = 0 \\ 13 + T(k-1) & \text{when } k > 0 \end{cases}$$

For a given list that is p elements long (note that $p \geq 0$), this would take 13 steps for the first element, 13 more steps for the second, 13 more for the third, and so on, until we run out of list elements and need 4 more steps: a total of $13p + 4$ steps. Notice this is precisely the same answer we obtained by the tabular method!

Exercise

Why can we assume that for a list p elements long, $p \geq 0$? 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 [section 16.10].

16.7 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): (13 * 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:

$$[k \rightarrow 13k + 4]$$

The brackets denote anonymous functions, with the parameters before the arrow and the body after.

16.8 Comparing Functions

Let's return to the running time of `len`. We've written down a function of great precision: 13! 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, $[k \rightarrow 13k + 4]$ and $[k \rightarrow 4k +$

$10]$. 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. This suggests a definition of this form:

$$\exists c. \forall n \in \mathbb{N}, f_1(n) \leq c \cdot f_2(n) \Rightarrow f_1 \leq f_2$$

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 [section 16.9].

Note carefully the order of identifiers. We must be able to pick the constant c up front for this relationship to hold.

Do Now!

Why this order and not the opposite order? What if we had swapped the two quantifiers?

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*.

This definition has more flexibility than we might initially think. For instance, consider our running example compared with $[k \rightarrow k^2]$. Clearly, the latter function eventually dominates the former: i.e.,

$$[k \rightarrow 13k + 4] \leq [k \rightarrow k^2]$$

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.

Observe that for a given function f, there are numerous functions that are less than it. We use the notation $O(\cdot)$ to describe this family of functions. Thus if

In computer science this is usually pronounced "big-Oh", though some prefer to call it the Bachmann-Landau notation after its originators.

$g \le f$, we can write $g \in O(f)$, which we can read as "f is an *upper-bound* for g". Thus, for instance,

$$[k \rightarrow 3k] \in O([k \rightarrow 4k + 12])$$
$$[k \rightarrow 4k + 12] \in O([k \rightarrow k^2])$$

and so on.

Pay especially close attention to our notation. We write $\in$ rather than $=$ or some other symbol, because $O(f)$ describes a family of functions of which g is a member. We also write f rather than $f(x)$ because we are comparing functions—f—rather than their values at particular points—$f(x)$—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

$$[k \rightarrow 3k] \in O([k \rightarrow 4k + 12])$$

This is not the only notion of function comparison that we can have. For instance, given the definition of $\le$ above, we can define a natural relation $<$. This then lets us ask, given a function f, what are all the functions g such that $g \le f$ but not $g < f$, i.e., those that are "equal" to f. 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 $\Theta(\cdot)$ to speak of this family of functions, so if g is equivalent to f by this notion, we can write $g \in \Theta(f)$ (and it would then also be true that $f \in \Theta(g)$).

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.

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 $f \le g$ and $g \le f$ then f and g are equal); and (c) transitive ($f \le g$ and $g \le h$ implies $f \le h$).

16.9 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 $O(F)$. Let's say we run it p times, for some given *constant*. The running time of the resulting code is then $p \times O(F)$. However, observe that this is really no different from $O(F)$: we can simply use a bigger constant for c in the definition of $O(\cdot)$—in particular, we can just use pc. Conversely, then, $O(pF)$ is equivalent to $O(F)$. This is the heart of the intution that "multiplicative constants don't matter".

- Suppose we have two functions, `f` in $O(F)$ and `g` in $O(G)$. 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., $O(F) + O(G)$. You should convince yourself that this is simply $O(F + G)$.

- Suppose we have two functions, `f` in $O(F)$ and `g` in $O(G)$. 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., $O(F) \times O(G)$. You should convince yourself that this is simply $O(F \times G)$.

These three operations—addition, multiplication by a constant, and multiplication by a function—cover just about all the cases. For instance, we can use this to reinterpret the tabular operations above (assuming everything is a function of k):

To ensure that the table fits in a reasonable width, we will abuse notation.

|Q|	**#Q**	**TotQ**	**|A|**	**#A**	**TotA**	**Total**
$O(1)$	$O(k)$	$O(k)$	$O(1)$	$O(1)$	$O(1)$	$O(k)$
$O(1)$	$O(k)$	$O(k)$	$O(1)$	$O(k)$	$O(k)$	$O(k)$

Because multiplication by constants doesn't matter, we can replace the 3 with 1. Because addition of a constant doesn't matter (run the addition rule in reverse), $k+1$ can become k. Adding this gives us $O(k)+O(k) = 2 \times O(k) \in O(k)$. This justifies claiming that running `len` on a k-element list takes time in $O([k \to k])$, which is a much simpler way of describing its bound than $O([k \to 13k + 4])$. 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.

16.10 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 [section 16.8].

- $$\begin{aligned} T(k) &= T(k-1) + c \\ &= T(k-2) + c + c \\ &= T(k-3) + c + c + c \\ &= ... \\ &= T(0) + c \times k \\ &= c_0 + c \times k \end{aligned}$$

  Thus $T \in O([k \rightarrow k])$. Intuitively, we do a constant amount of work (c) each time we throw away one element ($k-1$), so we do a linear amount of work overall.

- $$\begin{aligned} T(k) &= T(k-1) + k \\ &= T(k-2) + (k-1) + k \\ &= T(k-3) + (k-2) + (k-1) + k \\ &= ... \\ &= T(0) + (k-(k-1)) + (k-(k-2)) + \cdots + (k-2) + (k-1) + k \\ &= c_0 + 1 + 2 + \cdots + (k-2) + (k-1) + k \\ &= c_0 + \frac{k \cdot (k+1)}{2} \end{aligned}$$

  Thus $T \in O([k \rightarrow k^2])$. This follows from the solution to the sum of the first k numbers.

  We can also view this recurrence geometrically. Imagine each `x` below refers to a unit of work, and we start with k of them. Then the first row has k units of work:

  ```
  xxxxxxxx
  ```

  followed by the recurrence on $k-1$ of them:

  ```
  xxxxxxx
  ```

  which is followed by another recurrence on one smaller, and so on, until we fill end up with:

  ```
  xxxxxxxx
  xxxxxxx
  xxxxxx
  ```

```
xxxxx
xxxx
xxx
xx
x
```

The total work is then essentially the area of this triangle, whose base and height are both k: or, if you prefer, half of this $k \times k$ square:

```
xxxxxxxx
xxxxxxx.
xxxxxx..
xxxxx...
xxxx....
xxx.....
xx......
x.......
```

Similar geometric arguments can be made for all these recurrences.

- $$\begin{aligned} T(k) &= T(k/2) + c \\ &= T(k/4) + c + c \\ &= T(k/8) + c + c + c \\ &= \ldots \\ &= T(k/2^{\log_2 k}) + c \cdot \log_2 k \\ &= c_1 + c \cdot \log_2 k \end{aligned}$$

  Thus $T \in O([k \rightarrow \log k])$. Intuitively, we're able to do only constant work (c) at each level, then throw away half the input. In a logarithmic number of steps we will have exhausted the input, having done only constant work each time. Thus the overall complexity is logarithmic.

- $$\begin{aligned} T(k) &= T(k/2) + k \\ &= T(k/4) + k/2 + k \\ &= \ldots \\ &= T(1) + k/2^{\log_2 k} + \cdots + k/4 + k/2 + k \\ &= c_1 + k(1/2^{\log_2 k} + \cdots + 1/4 + 1/2 + 1) \\ &= c_1 + 2k \end{aligned}$$

  Thus $T \in O([k \rightarrow k])$. Intuitively, the first time your process looks at all the elements, the second time it looks at half of them, the third time a quarter, and so on. This kind of successive halving is equivalent to scanning all the elements in the input a second time. Hence this results in a linear process.

- $$\begin{aligned}T(k) &= 2T(k/2) + k\\ &= 2(2T(k/4) + k/2) + k\\ &= 4T(k/4) + k + k\\ &= 4(2T(k/8) + k/4) + k + k\\ &= 8T(k/8) + k + k + k\\ &= \ldots\\ &= 2^{\log_2 k}T(1) + k \cdot \log_2 k\\ &= k \cdot c_1 + k \cdot \log_2 k\end{aligned}$$

  Thus $T \in O([k \to k \cdot \log k])$. Intuitively, each time we're processing all the elements in each recursive call (the k) as well as decomposing into two half sub-problems. This decomposition gives us a recursion tree of logarithmic height, at each of which levels we're doing linear work.

- $$\begin{aligned}T(k) &= 2T(k-1) + c\\ &= 2T(k-1) + (2-1)c\\ &= 2(2T(k-2) + c) + (2-1)c\\ &= 4T(k-2) + 3c\\ &= 4T(k-2) + (4-1)c\\ &= 4(2T(k-3) + c) + (4-1)c\\ &= 8T(k-3) + 7c\\ &= 8T(k-3) + (8-1)c\\ &= \ldots\\ &= 2^kT(0) + (2^k - 1)c\end{aligned}$$

  Thus $T \in O([k \to 2^k])$. Disposing of each element requires doing a constant amount of work for it and then doubling the work done on the rest. This successive doubling leads to the exponential.

Exercise

Using induction, prove each of the above derivations.

Chapter 17

Sets Appeal

Earlier [section 11.2] we introduced sets. Recall that the elements of a set have no specific order, and ignore duplicates. 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.

If these ideas are not familiar, please read section 11.2, since they will be important when discussing the representation of sets.

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.

In principle, we want sets to obey the following interface:

Note that a type called `Set` is already built into Pyret, so we won't use that name below.

<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 [section 21.6.3]; and sometimes we might even want the elements to obey an ordering [section 17.2.1]. Numbers satisfy both characteristics.

17.1 Representing Sets by Lists

In what follows we will see multiple different representations of sets, so we will want names to tell them apart. We'll use `LSet` to stand for "sets represented as lists".

As a starting point, let's consider the implementation of sets using lists as the underlying representation. After all, a set appears to merely be a list wherein we ignore the order of elements.

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 [section 17.1.3]) 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`, `check`, and `size`. Suppose the size of the set is k (where, to avoid ambiguity, we let k 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 k. Similarly, `check` only needs to traverse the list once to determine whether or not an element is present, which also takes time linear in k. But `insert` needs to check whether an element is already present, which takes time linear in k, followed by at most a constant-time operation (`link`).

- If we `do` store duplicates, then `insert` is constant time: it simply `link`s on the new element without regard to whether it already is in the set representation. `check` traverses the list once, but the number of elements it needs to visit could be significantly greater than k, 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 k but the actual number of elements in `s` is d, where $d \geq k$. To compute the time to run `size` on d elements, $T(d)$, 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 $O([k \rightarrow d])$ operations. If it succeeds, we recur on something of size $T(d-1)$; else we do the same but perform a constant more operations. Thus $T(0)$ is a constant, while the recurrence (in big-Oh terms) is

$$T(d) = d + T(d-1)$$

Thus $T \in O([d \rightarrow d^2])$. 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	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 [chapter 18].

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? See [REF join lists] for a similar operation.

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.

17.2 Making Sets Grow on Trees

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 [section 16.10], one stands out: $T(k) = T(k/2) + c$. 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 k to $2k$, its logarithm—and hence resource usage—grows only by $\log 2k - \log k = \log 2$, 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.

17.2.1 Converting Values to Ordered Values

We have actually just made an extremely subtle assumption. When we check one element for membership and eliminate it, we have eliminated *only one element*. To eliminate *more than* one element, we need one element to "speak for" several. That is, eliminating that one value needs to have *safely* eliminated several others as well without their having to be consulted. In particular, then, we can no longer compare for mere equality, which compares one set element against another *element*; we need a comparison that compares against an element against a *set of elements*.

To do this, 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: e.g., below we assume its output is *partially ordered*.

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 A is less than another element B, then A is also less than all the other elements bigger than B. The same principle applies if the datatype is a string, using string inequality comparison. 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. In both cases, we will convert each character of the string into a number—e.g., by considering its ASCII encoding. Based on that, here are two 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). The second encoding is, of course, not invertible (e.g., simply permute the characters and, by commutativity, the sum will be the same).

Now let us consider more general datatypes. The principle of hashing will be similar. If we have a datatype with several variants, we can use a numeric tag to represent the variants: e.g., the primes will give us invertible tags. For each field of a record, we need an ordering of the fields (e.g., *lexicographic*, or "alphabetical" order), and must hash their contents recursively; having done so, we get in effect a string of numbers, which we have shown how to handle.

Now that we have understood how one can deterministically convert any arbitrary datum into a number, in what follows, we will assume that the trees representing sets are trees of numbers. However, it is worth considering what we really need out of a hash. In section 22.2, we will not need partial ordering. Invertibility is more tricky. In what follows below, we have assumed that finding a hash is tantamount to finding the set element itself, which is not true if multiple values can have the same hash. In that case, the easiest thing to do is to store alongside the hash all the values that hashed to it, and we must search through all of these values to find our desired element. Unfortunately, this does mean that in an especially perverse situation, the desired logarithmic complexity will actually be linear complexity after all!

In real systems, hashes of values are typically computed by the programming language implementation. This has the virtue that they can often be made unique. How does the system achieve this? Easy: it essentially uses the memory address of a value as its hash. (Well, not so fast! Sometimes the memory system can and does move values around [REF garbage collection]. In these cases computing a hash value is more complicated.)

17.2.2 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
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 candiate 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. 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:

We have used `<=` 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.

```
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.

But wait a minute. Are we actually done? Our recurrence takes the form $T(k) = T(k/2) + c$, 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 `insert`s 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.

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 -1, 0, or 1 (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. 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.

To implement a *bag* we count how many of each element are in it, which does not affect the tree's height.

Observe that because $<$ and $>$ are symmetric (likewise with $<=$ and $>=$), 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 [REF], 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.

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:

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.

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:

```
  p
 / \
  q   C
 / \
A   B
```

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.

Left-Left Case

Let's say the imbalance is in A, i.e., it has height $k+1$. Let's expand that tree:

```
      p
     / \
    q   C
   / \
  r   B
 / \
A1  A2
```

We know the following about the data in the sub-trees. We'll use the notation $T < a$ where T is a tree and a is a single value to mean every value in T is less than a.

- $A_1 < r$.
- $r < A_2 < q$.
- $q < B < p$.
- $p < C$.

Let's also remind ourselves of the sizes:

- The height of A_1 or of A_2 is k (the cause of imbalance).
- The height of the other A_i is $k - 1$ (see the exercise above).
- The height of C is k (initial assumption; k is arbitrary).
- The height of B must be $k - 1$ or 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:

```
     q
    / \
  r     p
 / \   / \
A1  A2 B  C
```

Observe that this preserves each of the ordering properties above. In addition, the A subtree has been brought one level closer to the root than earlier relative to B and C. This restores the balance (as you can see if you work out the heights of each of A_i, B, and C). Thus, we have also restored balance.

Left-Right Case

The imbalance might instead be in B. Expanding:

```
    p
   / \
  q   C
 / \
A   r
   / \
  B1  B2
```

Again, let's record what we know about data order:

- $A < q$.
- $q < B_1 < r$.
- $r < B_2 < p$.
- $p < C$.

and sizes:

- Suppose the height of C is k.
- The height of A must be $k-1$ or k.
- The height of B_1 or B_2 must be k, but not both (see the exercise above). The other must be $k-1$.

We therefore have to somehow bring B_1 and B_2 one level closer to the root of the tree. By using the above data ordering knowledge, we can construct this tree:

```
      p
     / \
    r   C
   / \
  q   B2
 / \
A   B1
```

Of course, if B_1 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:

```
      r
   /     \
  q       p
 / \     / \
A   B1 B2   C
```

Now observe that we have precisely maintained the data ordering constraints. Furthermore, from the root, A's lowest node is at height $k+1$ or $k+2$; so is B_1's; so is B_2's; and C's is at $k+2$.

Any Other Cases?

Were we a little too glib before? In the left-right case we said that only one of B_1 or B_2 could be of height k (after insertion); the other had to be of height $k-1$. Actually, all we can say for sure is that the other has to be *at most* height $k-2$.

Exercise

- Can the height of the other tree actually be $k-2$ instead of $k-1$?
- 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?

Chapter 18

Halloween Analysis

In chapter 16, we introduced the idea of big-Oh complexity to measure the worst-case time of a computation. As we saw in section 17.1.3, 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.

18.1 A First Example

Consider, for instance, a set that starts out empty, followed by a sequence of k insertions and then k membership tests, and suppose we are using the representation with*out* duplicates. Insertion time is proportional to the size of the set (and list); this is initially 0, then 1, and so on, until it reaches size k. Therefore, the total cost of the sequence of insertions is $k \cdot (k+1)/2$. The membership tests cost k each in the worst case, because we've inserted up to k distinct elements into the set. The total time is then

$$k^2/2 + k/2 + k^2$$

for a total of $2k$ operations, yielding an average of

$$\frac{3}{4}k + \frac{1}{4}$$

steps *per operation in the worst case*.

18.2 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*. 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 [section 18.3.5], can be quite a bit worse); it only says what their average is.

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.

In the above case, this new analysis did not yield any big surprises. We have found that on average we spend about k steps per operation; a big-Oh analysis would have told us that we're performing $2k$ operations with a cost of $O([k \rightarrow k])$ 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.

18.3 An Example: Queues from Lists

We have already seen lists [chapter 8] and sets [chapter 17]. Now let's consider another fundamental computer science data structure: the *queue*. A queue is a linear, ordered data structure, just like a list; however, the set of operations they offer is different. In a list, the traditional operations follow a last-in, first-out discipline: `.first` returns the element most recently `link`ed. In contrast, a queue follows a first-in, first-out discipline. That is, a list can be visualized as a stack, while a queue can be visualized as a conveyer belt.

18.3.1 List Representations

We can define queues using lists in the natural way: every *enqueue* is implemented with `link`, while every *dequeue* requires traversing the whole list until its end. Conversely, we could make enqueuing 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.

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, enqueuing 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.

18.3.2 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 k enqueues followed by k dequeues, then each of the enqueues takes one step; each of the last $k-1$ 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 k. Thus, the total cost of operations for this sequence is $k \cdot 1 + k + (k-1) \cdot 1 = 3k-1$ for a total of $2k$ operations, giving an *amortized* complexity of effectively *constant* time per operation!

18.3.3 More Liberal Sequences of Operations

In the process of this, however, I've quietly glossed over something you've probably noticed: 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 enqueuing 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
```

18.3.4 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 enqueuing. 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.

18.3.5 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.

18.4 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.

Chapter 19

Sharing and Equality

19.1 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
```

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 [section 21.2.2] and to the full range of Pyret's equality operations [section 21.6].

19.2 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 "*refer*ence") 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. 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.

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.

19.3 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("Haberman") ])
```

and a way to extract a particular person's page:

```
fun get-person(n): index(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 = index(site.sub, 0)
  systems = index(site.sub, 1)
  theory-dijkstra = index(theory.sub, 1)
  systems-dijkstra = index(systems.sub, 0)
  theory-dijkstra is systems-dijkstra
  theory-dijkstra is%(identical) systems-dijkstra
end
```

19.4 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` [section 15.3]. 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.

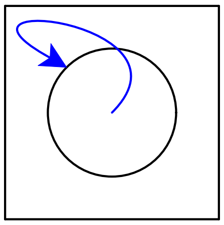
Suppose, however, that `web-colors` were written instead as a function definition to delay its creation:

```
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 `link`s.

- 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: section 21.3.

Chapter 20

Graphs

In section 19.4 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.

20.1 Understanding Graphs

Consider again the binary trees we saw earlier [section 19.1]. 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 [section 19.4], it is not completely straightforward to create such a structure, but what we saw earlier [section 15.3] 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. The support function it needs checks whether a given node has already been seen:

Note that this could just as well be a set; it doesn’t have to be a list.

```
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.

20.2 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 [section 20.3].

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.

Previously [chapter 17], we have explored 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:

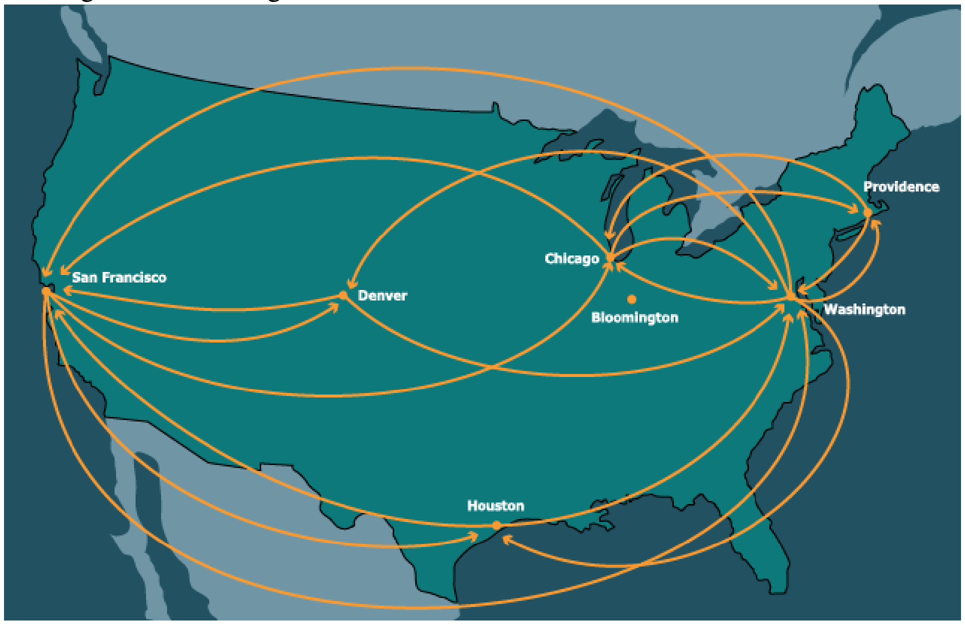
1. A way to construct graphs.

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:



20.2.1 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", "
  knORD = keyed-node("chi", "Chicago", [list: "was", "saf", "pvd"])
  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
```

20.2.2 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.

The prefix ix- stands for "indexed".

First, the datatype:

```
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.)

20.2.3 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.

20.2.4 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 `Key`s (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 [section 20.2.1].

20.3 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 k 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 k (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 k^2. 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* graph, 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.

A graph is connected if, from every node, we can traverse edges to get to every other node.

20.4 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.

20.4.1 Simple Recursion

A *path* is a sequence of zero or more linked edges.

At its simplest, reachability is easy. We want to know whether there exists a path 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 destintion 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?

20.4.2 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
```

20.4.3 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>
    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?

20.4.4 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.

20.5 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* [section 18.3].

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 N nodes and E edges, then a lower-bound on the complexity of traversal is $O([N, E \rightarrow N + E])$. 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: section 17.2). 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 N, when it visits a node P, count the number of edges taken to get to P. Breadth-first traversal guarantees that there cannot have been a shorter path to P: that is, it finds a *shortest* path to P.

Exercise

Why "a" rather than "the" shortest path?

Do Now!

Prove that breadth-first traversal finds a shortest path.

20.6 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?

20.7 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 [section 20.5] 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 desination 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, q, 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 q 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 $s \rightarrow d$ from source s to node d, 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 $s \rightarrow d$ 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* [REF] 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*.

20.8 Moravian Spanning Trees

At the turn of the milennium, 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.

20.8.1 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 academia 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.

20.8.2 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 [section 20.8.5]. 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.

20.8.3 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. 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!).

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.

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 $[e \rightarrow e \log e]$ 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.

20.8.4 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. 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.

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.

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.

20.8.5 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
```

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
```

Here are some tests to illustrate this working:

```
check:
  s0 = map(elt(_, none), [list: 0, 1, 2, 3, 4, 5, 6, 7])
  s1 = union(index(s0, 0), index(s0, 2), s0)
  s2 = union(index(s1, 0), index(s1, 3), s1)
  s3 = union(index(s2, 3), index(s2, 5), s2)
  print(s3)
  is-same-element(fynd(index(s0, 0), s3), fynd(index(s0, 5), s3)) is t
  is-same-element(fynd(index(s0, 2), s3), fynd(index(s0, 5), s3)) is t
  is-same-element(fynd(index(s0, 3), s3), fynd(index(s0, 5), s3)) is t
  is-same-element(fynd(index(s0, 5), s3), fynd(index(s0, 5), s3)) is t
  is-same-element(fynd(index(s0, 7), s3), fynd(index(s0, 7), s3)) is t
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. `fynd` 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.

The bottom line is that *pure functional programming is not a great fit with this problem*. We need a better implementation strategy: section 22.1.

Chapter 21

State, Change, and More Equality

21.1 A Canonical Mutable Structure

As we have motivated [section 20.8.5], sometimes it's nice to be able to *change* the value of a datum rather than merely construct a new one with an updated value. The main advantage to changing it is that every value that refers to it can now see this change. The main disadvantage to changing it is that every value that refers to it can now see this change. Using this power responsibly is therefore an important programming challenge.

To put this idea in the simplest light, let us consider the simplest kind of mutable datum: one that has only one field. We call this a *box*, and treat it as a fresh container type. Boxes will support just three operations:

1. `box` consumes a value and creates a mutable box containing that value.

2. `unbox` consumes a box and returns the value contained in the box.

3. `set-box` consumes a box, a new value, and *changes* the box to contain the value. All subsequent `unbox`es of that box will now return the new value.

In a typed language, we would require that the new value put in the box by `set-box` be type-consistent with what was there before. (Even in a language that isn't statically typed, we would presumably expect the same to keep the programming process sane.) You can thus think of a box as equivalent to a Java container class with parameterized type, which has a single member field with a getter and setter: `box` is the constructor, `unbox` is the getter, and `set-box` is the setter. (Because there is only one field, its name is irrelevant.)

```
class Box<T> {
    private T the_value;
    Box(T v) {
        this.the_value = v;
    }
    T get() {
        return this.the_value;
    }
    void set(T v) {
        this.the_value = v;
    }
}
```

Correspondingly, here is a definition of a box in Pyret:

```
data Box:
  | box(ref v)
where:
  n1 = box(1)
  n2 = box(2)
  n1!{v : 3}
  n2!{v : 4}
  n1!v is 3
  n2!v is 4
end
```

Notice that in Pyret, because values are immutable by default, we have to explicitly declare the `v` field to be mutable using `ref`; mutable fields must be accessed using `!` rather than `.`, the dot operator.

The reason for using a different syntax is to warn the programmer that the value obtained from this field may change over time, so they should not make assumptions about its longevity. Because Pyret is single-threaded, however, they can assume the value will stay unchanged until either the next mutation in the same procedure, or until the next call to another procedure or return from this procedure—whichever comes soonest.

Do Now!

Why do we say "type-consistent" above, rather than "the same type"?

The values could be related by subtyping [section 32.6.1].

Exercise

What does the comment about longevity mean? How does this apply to reusing values extracted from fields?

21.2 Equality and Mutation

We've already seen [section 19.1] that equality is subtle. It's about to become much subtler with the introduction of mutation!

First, let's make sure we understand how mutable fields operate with equality. As a running example, we'll work with:
<three-boxes> ::=
```
  b0 = box("a value")
  b1 = box("a value")
  b2 = b1
```
Observe that `b1` and `b2` are referring to the *same* box, while `b0` is referring to a different one. We can see this because the following tests pass:

```
check:
  b0!v is b1!v
  b1!v is b2!v

  b0 is-not%(identical) b1
  b1 is%(identical) b2
  b0 is-not%(identical) b2
end
```

In other words, `b1` and `b2` are *aliases* for each other: they are two different names for one and the same value. In contrast, neither name is an alias for the value referred to by `b0`. Since `identical` is transitive, it follows from the above two checks (and Pyret confirms for us) that `b0` is not `identical` to `b2`.

21.2.1 Observing Mutation

In the presence of mutation, the subtleties we've discussed about equality become even more significant. Specifically, if we change one value, all aliases to that value detect the change. Consider the following code:
<modify-b1> ::=
```
  b1!{v: "a different value"}
```

Observe that if you just copy these definitions and tests one after the other in your editor, tests that should succeed will fail. This is because Pyret moves all test blocks to the end of the program, so the checks are not running in the source location where they are written. Until now, this made not a whit of difference. Now that we have mutation, it makes a world of difference. Therefore, you will need to erase old tests as you add new code that modifies state. Because that is the point of state: statements that were previously true no longer are.

Do Now!

Which of the following tests do you expect to change?

- b0!v is b1!v
- b1!v is b2!v
- b0 is-not%(identical) b1
- b1 is%(identical) b2

Values that were not previously equal (by any measure) are not going to become so as a result of a change; however, values that appeared to be equal will no longer be: specifically, `b0!v is b1!v` will no longer be true.

Symmetrically, we can modify the value in `b2` and the change will be visible via `b1`, but not `b0`:

```
b2!{v: "yet another value"}

check:
  b0!v is-not b1!v
  b1!v is b2!v

  b0 is-not%(identical) b1
  b1 is%(identical) b2
  b0 is-not%(identical) b2
end
```

after which we can restore the value:

```
b1!{v: hold-b1-value}
```

Exercise

Modify the content of `b0` and see which tests break.

21.2.2 What it Means to be Identical

Before we modify the content of these boxes, we could hold on to their values:

```
hold-b1-value = b1!v
b1!{v: "a different value"}
```

And at the end of performing comparisons, we can restore them:

```
b1!{v: hold-b1-value}
```

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*implements the essence of `identical`*.

In practice, `identical` does not behave this way: it would be too disruptive—e.g., if this fake value was saved to persistent storage and then the system crashed—and it would anyway be observable if the system had multiple threads. It is also not the most efficient implementation possible. 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).

21.2.3 An Additional Challenge

If all this is a little confusing, don't fret: aliasing is a vexing problem in programming, and trips up even the most experienced programmers. Not properly understanding the aliasing behavior in code causes various errors (when it's underestimated) or loss of performance (when it's overestimated). One of the benefits of the functional style we've adopted until now is we don't have to worry about the impact of aliasing.

Nevertheless, if you're reading this, you're trying to become a more thorough programmer, which means you should get a grasp of this topic. Here's an extension to the code in section 21.1, before any mutations:

```
b4 = box(b1!v)
```

Here are a few tests, some or all of which may surprise you:

```
check:
  b1!v is b4!v
  b1!v is%(identical) b4!v
  b1 is-not%(identical) b4
end
```

Now suppose we add

```
b1!{v: "new value"}
```

What does this do to our tests? Why?

21.3 Recursion and Cycles from Mutation

Mutation can also help us make sense of recursion. Let us return to the example we tried to write earlier [section 19.4]:

```
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!—so that any definition can avail of it.

Returning to our example above, recall that we can't make up our list using `link`s, because we want the list to never terminate. Therefore, let us first define a new datatype to hold an cyclig list.

```
data CList: clink(v, r) end
```

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.)

21.3.1 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 CList: clink(v, ref r) end
```

Note that this datatype lacks a base case, which should remind you of definitions we saw in section 15.3.

Using this, we can define:

```
white-clink = clink("white", "dummy")
grey-clink = clink("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-clink!{r: grey-clink}
grey-clink!{r: white-clink}
```

Because we have ordained that our colors must alternate beginning with white, this rounds up our definition:

```
web-colors = white-clink
```

If we ask Pyret to inspect the value of `web-colors`, we notice that it employs an algorithm to prevent traversing infinite objects. We can define a helper function, `take`, a variation of which we saw earlier [section 15.3], to inspect a finite prefix of an infinite list:

```
fun take(n :: Number, il :: CList) -> List:
  if n == 0:
    empty
  else:
    link(il.v, take(n - 1, il!r))
  end
end
```

such that:

```
check:
  take(4, web-colors) is
  [list: "white", "grey", "white", "grey"]
end
```

21.3.2 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`.

21.3.3 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 `clink`s.

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.

21.3.4 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 using desugaring [section 24.4], 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.

21.4 From Identifiers to Variables

As we have seen, mutable values can be aliased, which means references can inadvertently have their values changed. Because these values can be passed around, it can be difficult to track all the aliases that might exist (because it would be infeasible for a value to retain "backward references").

Therefore, in Pyret as in most languages, there is another form of mutable, called a *variable*. A variable is an identifier whose binding can be changed (as opposed to re-bound in a new scope); in Pyret, the syntax for changing the value of a variable is `:=`. Furthermore, variables must be declared explicitly by preceding their declaration with `var`:

```
var x = 0
x := 1
```

The `var` keyword forces you to understand that the `=` is not a true equality: it's equal *for now*, but may not be in the future.

We've insisted on using the word "identifier" before because we wanted to reserve "variable" for what we're about to study. In Java, when we say (assuming `x` is locally bound, e.g., as a method parameter)

```
x = 1;
x = 3;
```

we're asking to *change* the value of `x`. After the first assignment, the value of `x` is `1`; after the second one, it's `3`. Thus, the value of `x` *varies* over the course of the execution of the method.

Now, we also use the term "variable" in mathematics to refer to function parameters. For instance, in *f(y) = y+3* we say that *y* is a "variable". That is called a variable because it varies *across invocations*; however, *within* each invocation, it has the same value in its scope. Our identifiers until now have corresponded to this mathematical notion of a variable. In contrast, programming variables can vary even *within* each invocation, like the Java `x` above.

If the identifier was bound to a box, then it remained bound to the same box value. It's the content of the box that changed, not which box the identifier was bound to.

Henceforth, we will use *variable* when we mean an identifier whose value can change within its scope, and *identifier* when this cannot happen. If in doubt, we might play it safe and use "variable"; if the difference doesn't really matter, we might use either one. It is less important to get caught up in these specific terms than to understand that they represent a distinction that matters [chapter 31].

21.5 Interaction of Mutation with Closures: Counters

Suppose we want to create a function that counts how many times it has been invoked. Further, let us assume we can create new counters on need. Thus, `mk-counter` creates a fresh counter each time, each of which maintains its own count history. A sample use might look like this:

```
check:
  l1 = mk-counter()
  l1() is 1
  l1() is 2
  l2 = mk-counter()
  l2() is 1
  l1() is 3
  l2() is 2
end
```

Notice that each invocation of `mk-counter` makes a *fresh* counter, so the counters created by two separate invocations do not interfere with one another.

We now see how we can implement this using both mutable structures (specifically, boxes) and variables.

21.5.1 Implementation Using Boxes

Here is the implementation:

```
fun mk-counter():
  ctr = box(0)
  lam():
    ctr!{v : (ctr!v + 1)}
    ctr!v
  end
end
```

Why does this work? It's because each invocation of `mk-counter` creates a box only *once*, which it binds to `ctr`. The closure closes over this one box. All subsequent mutations affect *the same box*. In contrast, swapping two lines makes a big difference:

```
fun mk-broken-counter():
  lam():
    ctr = box(0)
    ctr!{v : (ctr!v + 1)}
    ctr!v
  end
where:
  l1 = mk-broken-counter()
  l1() is 1
  l1() is 1
  l2 = mk-broken-counter()
  l2() is 1
  l1() is 1
  l2() is 1
end
```

In this case, a new box is allocated on every invocation, not of `mk-broken-counter` but of the function that it returns, so the answer each time is the same (despite the mutation inside the procedure). Our implementation of boxes should be certain to preserve this distinction.

The examples above hint at an implementation necessity. Clearly, whatever the environment closes over in the procedure returned by `mk-counter` must refer to the same box each time. Yet something also needs to make sure that the value in that box is different each time! Look at it more carefully: it must be *lexically* the same, but *dynamically* different. This distinction will be at the heart of a strategy for implementing state [chapter 31].

21.5.2 Implementation Using Variables

The implementation using variables is virtually identical:

```
fun mk-counter():
  var ctr = 0
  lam():
    ctr := ctr + 1
    ctr
  end
where:
  l1 = mk-counter()
  l1() is 1
  l1() is 2
  l2 = mk-counter()
  l2() is 1
  l1() is 3
  l2() is 2
end
```

And sure enough, if we swap the same critical two lines, we get the wrong behavior:

```
fun mk-broken-counter():
  lam():
    var ctr = 0
    ctr := ctr + 1
    ctr
  end
where:
  l1 = mk-broken-counter()
  l1() is 1
  l1() is 1
  l2 = mk-broken-counter()
  l2() is 1
  l1() is 1
  l2() is 1
end
```

21.6 A Family of Equality Predicates

Until now we have seen two notions of equality:

- The binary operator `==`, which is also used as the equality comparison by `in` when testing.
- `identical`, also written as `<=>`.

However, we haven't discussed exactly what `==` means, and it turns out there are two things it could mean, leaving us with *three* different notions of equality. To see this, refresh your memory with our three boxes.

We have already covered the meaning of `identical`. However, there is an intuitive level where it is unsatisfying: when our notion of equality is, "When printed (as output) or written (as input), would these two values look the same?", where `box("a value")` and `box("a value")` are the "same". That is, it would be nice to have an equality operator—call it `E1`—such that

```
check:
  E1(b0, b1) is true
  E1(b1, b2) is true
end
```

because all three look the same when written out as values.

However, as we just saw [section 21.2.2], these equivalences can be ephemeral. When we modify `b1` (see above), clearly these no longer print identically:

```
››› b0
```

```
box("a value")
```

```
››› b1
```

```
box("a different value")
```

```
››› b2
```

```
box("a different value")
```

so we would expect

```
check:
  E1(b0, b1) is false
  E1(b1, b2) is true
end
```

There is in fact such an operator in Pyret: it is called `equal-now`, and written as a binary operator as `=~` (the ∼ is meant to be suggestive of hand-waving, because the value is equal *now*, but you shouldn't assume it will be in the future). As the name and visual suggest, this is a fragile operator: you should not write programs that assume anything about the long-term equality of values that are equal at this moment, in case the values are mutable. However, it can still be useful to know whether, at this instant, two values will, in effect, "print the same".

Exercise

Confirm that `equal-now` does indeed have the properties ascribed to `E1` above.

However, this still does not enable us to distinguish between `==` and `identical`:

```
check:
  (b0 == b1) is false
  (b1 == b2) is true
  identical(b0, b1) is false
  identical(b1, b2) is true
end
```

For that, it helps to have a slightly richer structure:

```
b0 = box("a value")
b1 = box("a value")
b2 = b1
```

```
l0 = [list: b0]
l1 = [list: b1]
l2 = [list: b2]
```

Note that we are allocating three different lists, though two of them share the same mutable box. Now we find something interesting. Unsurprisingly, the following is true:

```
check:
  identical(l0, l1) is false
  identical(l1, l2) is false
end
```

while

```
check:
  equal-now(l0, l1) is true
  equal-now(l1, l2) is true
end
```

However:

```
check:
  (l0 == l1) is false
  (l1 == l2) is true
end
```

What might `==` represent that is interestingly different from both `identical` and `equal-now`? When it returns `true`, it is that the two values will "print the same" now *and forever*. How is this possible? It is because `==` recursively checks that the two arguments are structural *until* it gets to a mutable field; at that point, it checks that they are `identical`. If they are identical, then any change made to one will be reflected in the other (because they are in fact the same mutable field). That means their content, too, will always "print the same". Therefore, we can now reveal the name given to `==`: it is `equal-always`.

21.6.1 A Hierarchy of Equality

Observe that if two values `v1` and `v2` are `equal-now`, they are not necessarily `equal-always`; if they are `equal-always`, they are not necessarily identical. We have seen examples of both these cases above.

In contrast, if two values are `identical`, then they are certainly going to be `equal-always`. That is because their mutable fields reduce to `identical`, while the immutable parts—which will be traversed structurally—are guaranteed to yield equality, being in fact the same value. In turn, if they satisfy `equal-always` they are guaranteed to be `equal-now`, because the only difference is that `equal-now` structurally traverses the content of mutable fields, but if these are `identical` (as they must be, to be `equal-always`), they are certain to be structurally equal.

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.

21.6.2 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.

For some programs, the cost of checking equality may be considerable. There are two common strategies such a program can employ:

1. Use a quick check followed by a slower check only if necessary. For instance, suppose we want to speed up `equal-always`, and have reason to believe we will often compare `identical` elements and/or that the values being compared are very large. Then we might define:

   ```
   fun my-eq(v1, v2) -> Boolean:
     identical(v1, v2) or equal-always(v1, v2)
   end
   ```

   which has the following behavior:

   ```
   check:
   ```

```
    my-eq(b0, b1) is false
    my-eq(b1, b2) is true
    my-eq(l0, l1) is false
    my-eq(l1, l2) is true
  end
```

This is exactly the same as the behavior of `equal-always`, but faster when it can discharge the equality using `identical` without having to traverse the data. (Observe that this is a safe optimization because `identical` implies `equal-always`.)

2. Use a different equality strategy entirely, if possible: see section 22.2.

21.6.3 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 [REF].

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 [section 21.6.1]. 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 has to be written defensively. Because this is annoying, 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.

Chapter 22

Algorithms That Exploit State

22.1 Disjoint Sets Redux

Here's how we can use this to implement union-find afresh. We will try to keep things as similar to the previous version [section 20.8.5] as possible, to enhance comparison.

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
```

22.1.1 Optimizations

Look again at `fynd`. 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 `fynd` 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.

22.1.2 Analysis

This optimized union-find data structure has a remarkble 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.

22.2 Set Membership by Hashing Redux

We have already seen solutions to set membership. First we saw how to represent sets as lists [section 17.1], then as (balanced) binary trees [section 17.2.3]. With this we were able to reduce insertion and membership to logarithmic time in the number of elements. Along the way, we also learned that the essence of using these representations was to reduce any datatype to a comparable, ordered element—for efficiency, usually a number [section 17.2.1]—which we called *hashing*.

Don't confuse this with union-find, which is a different kind of problem on sets [section 22.1].

Let us now ask whether we can use these numbers in any other way. Suppose our set has only five elements, which map *densely* to the values between 0 and 4. We can then have a five element list of boolean values, where the boolean at each index of the list indicates whether the element corresponding to that position is in the set or not. Both membership and insertion, however, require traversing potentially the entire list, giving us solutions linear in the number of elements.

That's not all. Unless we can be *certain* that there will be only five elements, we can't be sure to bound the size of the representation. Also, we haven't yet shown how to actually hash in a way that makes the representation dense; barring that, our space consumption gets much worse, in turn affecting time.

There is, actually, a relatively simple solution to the problem of reducing numbers densely to a range: given the hash, we apply modular arithmetic. That is, if we want to use a list of five elements to represent the set, we simply compute the hash's modulo five. This gives us an easy solution to that problem.

Except, of course, not quite: two different hashes could easily have the same modulus. That is, suppose we need to record that the set contains the (hash) value 5; the resulting list would be

```
[list: true, false, false, false, false]
```

Now suppose we want to ask whether the value 15 is in the set; we cannot tell from this representation whether it's in the set or not, because we can't tell whether the `true` represents 5, 15, 25, or any other value whose modulus 5 is 0. Therefore, we have to record the actual elements in the set; for type-consistency, we should be using an `Option`:

```
[list: some(5), none, none, none, none]
```

Now we can tell that 5 is in the set while 4 is not. However, this now makes it impossible to have both 5 and 10 in the set; therefore, our real representation needs to be a *list* at each position:

```
[list: [list: 5], empty, empty, empty, empty]
```

If we also add 10 to the set, we get:

```
[list: [list: 5, 10], empty, empty, empty, empty]
```

and now we can tell that both 5 and 10 are in the set, but 15 is not. These sub-lists are known as *buckets*.

Good; now we have another way of representing sets so we can check for membership. However, in the worst case one of those lists is going to contain all elements in the set, and we may have to traverse the entire list to find an element in it, which means membership testing will take time linear in the number of elements.

Insertion, in turn, takes time proportional to the size of the modulus because we may have to traverse the entire outer list to get to the right sub-list.

Can we improve on this?

22.2.1 Improving Access Time

Given that we currently have no way of ensuring we won't get hash collisions, for now we're stuck with a list of elements at each position that could be the size of the set we are trying to represent. Therefore, we can't get around that (yet). But, we're currently paying time in the size of the outer list just to insert an element, and surely we can do better than that!

We can, but it requires a different data structure: the *array*. You can look up arrays in the Pyret documentation. The key characteristics of an array are:

There are other data structures that will also do better, but the one we're about to see is important and widely used.

- Accessing the *n*th element of an array takes *constant*, not linear, time in *n*. This is sometimes known as *random-access*, because it takes the same time to access any random element, as opposed to just a known element.

- Arrays are updated by mutation. Thus, a change to an array is seen by all references to the array.

The former property warrants some discussion: how can an array provide random access whereas a list requires time linear in the index of the element we're accessing? This is because of a trade-off: a list can be extended indefinitely as the program extends, but an array cannot. An array must declare its size up front, and cannot grow without copying all the elements into a larger array. Therefore, we should only use arrays when we have a clearly identifiable upper-bound on their size (and that bound is not too large, or else we may not even be able to find that much contiguous space in the system). But the problem we're working on has exactly this characteristic.

So let's try defining sets afresh. We start with an array of a fixed size, with each element an empty list:

```
SIZE = 19
v = array-of(empty, SIZE)
```

We need to use modular arithmetic to find the right bucket:

```
fun find-bucket(n): num-modulo(n, SIZE) end
```

With this, we can determine whether an element is in the set:

```
fun get-bucket(n): array-get-now(v, find-bucket(n)) end
fun is-in(n): get-bucket(n).member(n) end
```

To actually add an element to the set, we put it in the list associated with the appropriate bucket:

```
fun set-bucket(n, anew): array-set-now(v, find-bucket(n), anew) end
fun put(n):
  when not(is-in(n)):
    set-bucket(n, link(n, get-bucket(n)))
  end
end
```

Checking whether the element is already in the bucket is an important part of our complexity argument because we have implicitly assumed there won't be duplicate elements in buckets.

Exercise

What impact do duplicate elements have on the complexity of operations?

The data structure we have defined above is known as a *hash table* (which is a slightly confusing name, because it isn't really a table of hashes, but this is the name used conventionally in computer science).

22.2.2 Better Hashing

Using arrays therefore appears to address one issue: insertion. Finding the relevant bucket takes constant time, linking the new element takes constant time, and so the entire operation takes constant time...except, we have to also check whether the element is already in the bucket, to avoid storing duplicates. We have gotten rid of the traversal through the outer list representing the set, but the `member` operation on the inner list remains unchanged. In principle it won't, but in practice we can make it much better.

Note that collisions are virtually inevitable. If we have uniformly distributed data, then collisions show up sooner than we might expect. Therefore, it is wise to prepare for the possibility of collisions.

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!

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.

22.2.3 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 list (or array) of booleans, with modular arithmetic, to represent the set. When the set said 4 was not present, this was absolutely true; but when it said 5 and 10 are both present, only one of these was present. The advantage was a huge saving in space and time: we needed only one bit per bucket, and did not need to search through a list to answer for membership. The downside, of course, was a hugely inaccurate set data structure, and one with *correlated* failure tied to the modulus.

There is a simple way to improve this solution: instead of having just one array, have several (but a fixed number of them). 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.

22.3 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.

22.3.1 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?

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)`.

This process, of converting a function into a version that remembers its past answers, is called *memoization*.

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.

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 `896519947090131496687`
and computations on numbers can no longer be constant time, contrary to what we said earlier [section 16.4]. 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).

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* [chapter 32]. 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 [section 21.3.2].

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!

If in doubt about how state interacts with functions, read section 21.5.

Exercise

Why is sharing memoization tables a bad idea? Be concrete.

22.3.2 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. Given such an intended use, we would like at least the following to hold:

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.

- 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 [section 22.3.3].

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", "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", "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.I
  levenshtein(
    [list: "a", "c", "g", "t", "a", "c", "g", "t", "a", "c", "g", "t
    [list: "a", "c", "a", "t", "a", "c", "t", "t", "g", "t", "a", "c
    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 “kitten” 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 m and n is $O([m, n \rightarrow m \cdot n])$. (We will return to space consumption later [section 22.3.5].)

Exercise

Modify the above algorithm to produce an actual (optimal) sequence of edit operations. This is sometimes known as the *traceback*.

22.3.3 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. 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. This algorithm is named, after its creators, *Smith-Waterman*, and because it is essentially identical, has the same complexity as the Levenshtein algorithm.

This section may need to be skipped in some states and countries.

To be precise, we are performing *local* sequence alignment.

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.

22.3.4 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.

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 chapter 18.

```
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:

```
fun fill-catalan(upper):
  array-set-now(answers, 0, some(1))
  when upper > 0:
    for map(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, resultingin 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.

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>):
    <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 n, we have to record the edit distance to its $n + 1$ 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.

Which proved to be necessary when writing and debugging this code!

We have initialized the table with `none`, so we will get an error if we accidentally try to use an uninitialized entry. 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):
    answers.set(s1-idx, s2-idx, some(n))
  end
  fun lookup(s1-idx :: Number, s2-idx :: Number) -> Number:
    a = answers.get(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 $(0, 0)$, 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 index(s1, s1i) == index(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, with 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>):
  s1-len = s1.length()
  s2-len = s2.length()
  answers = array2d(s1-len + 1, s2-len + 1, none)

  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 index(s1, s1i) == index(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.

22.3.5 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 space 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 a

As this table should make clear, these are essentialy *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 [section 14.4] 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 implemenation, 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?

Chapter 23

Processing Programs: Parsing

23.1 Understanding Languages by Writing Programs About Them

We will understand the nature of languages by writing programs about them. These programs will implement many interesting features of languages from different perspectives, embodied in different actions:

- An *interpreter* will consume programs in a language and produce the answers they are expected to produce.
- A *type checker* will consume programs in a language and produce either true or false, depending on whether the program has consistent type annotations.
- A *pretty-printer* will consume programs in a language and print them, prettified in some way.
- A *verifier* will consume programs in a language and check whether they satisfy some stated property.
- A *transformer* will consume programs in a language and produce related but different programs in the same language.
- A transformer's first cousin, a *compiler*, will consume programs in a language and produce related programs in a different language (which in turn can be interpreted, type-checked, pretty-printed, verified, transformed, even compiled...).

Observe that in each of these cases, we have to begin by consuming (the representation of) a program. We will briefly discuss how we do this quickly and easily, so that in the rest of our study we can focus on the remainder of each of these actions.

23.2 Everything (We Will Say) About Parsing

Parsing is a very general actvity whose difficulty depends both on how complex or ambiguous the input might be, and how much stucture we expect of the parser's output. For our purposes, we would like the parser to be maximally helpful by providing later stages as much structure as possible. This forces us to either write a very complex parser or limit the forms of legal input. We will choose the latter.

A key problem of parsing is the management of *ambiguity*: when a given expression can be parsed in multiple different ways. For instance, the input

```
23 + 5 * 6
```

could parse in two different ways: either the multiplication should be done first followed by addition, or vice versa. Though simple disambiguation rules (that you probably remember from middle school) disambigiuate arithmetic, the problem is much harder for full-fledged programming languages.

Ultimately, we would like to get rid of ambiguity once-and-for-all at the very beginning of processing the program, rather than deal with it repeatedly in each of the ways we might want to process it. Thus, if we follow the standard rules of arithmetic, we would want the above program to turn into a tree that has a (representation of) addition at its root, a (representation of) 23 as its left child, multiplication as its right child, and so on. This is called an *abstract syntax tree*: it is "abstract" because it represents the intent of the program rather than its literal syntactic structure (spaces, indentation, etc.); it is "syntax" because it represents the program that was given; and it is usually a "tree" but not always.

As we have said, we could push the problem of disambiguation onto a parser. This is what most real languages do. Because parsing is not our concern, we are instead going to ask the program's author to use an unambiguous syntax. Indeed, we can exploit the decades of work that has been invested into *wire format* to represent programs. For instance, the above expression might be written—avoiding the ambiguity induced by not properly parenthesizing the program—as:

```
<plus>
  <args>
    <arg position="1">
      <number value="23"/>
    </arg>
    <arg position="2">
      <mult>
        <args>
          <arg position="1">
```

```
            <number value="5"/>
          </arg>
          <arg position="2">
            <number value="6"/>
          </arg>
        </args>
      </mult>
    </arg>
  </args>
</plus>
```

in XML, or as

```
{plus:
  [{number: "23"},
   {mult:
     [{number: "5"},
      {number: "6"}]}]}
```

in JSON.

23.2.1 A Lightweight, Built-In First Half of a Parser

These are both worthy notations. Instead, we will use a related, and arguably even simpler, wire format known as the *s-expression*:

The name comes from Lisp.

```
(+ 23 (* 5 6))
```

Pyret has built-in support for processing s-expressions, so you can use this syntax and get support from the language to process it.

Do Now!

Load the s-expression library with

```
import s-exp as S
```

and then try the following:

```
S.read-s-exp("(+ 23 (* 5 6))")
```

Make sure you understand the output it produced and why it produced that.

You should have seen the following output:

```
check:
  S.read-s-exp("(+ 23 (* 5 6))") is
    S.s-list([list:
      S.s-sym("+"),
      S.s-num(23),
      S.s-list([list:
        S.s-sym("*"),
        S.s-num(5),
        S.s-num(6)])])
end
```

In this book we will use s-expressions to represent concrete syntax. This is helpful because the syntax is so different from that of Pyret, we will virtually never be confused as to what language we are reading. Since we will be writing programs to process programs, it is especially helpful to keep apart the program *being processed* and that *doing the processing*. For us, the former will be written in s-expressions and the latter in Pyret.

23.2.2 Completing the Parser

In principle, we can think of `read-s-exp` as a complete parser. However, its output is generic: it represents the token structure without offering any comment on its intent. We would instead prefer to have a representation that tells us something about the *intended meaning* of the terms in our language, just as we wrote at the very beginning: "(representation of) multiplication", and so on.

To do this, first let's import the necessary libraries:

```
import s-exp as S
import lists as L
```

Now down to business. We must first introduce a datatype that captures this representation. We will separately discuss [section 24.1] how and why we obtained this datatype, but for now let's say it's given to us:

```
data ArithC:
  | numC(n :: Number)
  | plusC(l :: ArithC, r :: ArithC)
  | multC(l :: ArithC, r :: ArithC)
end
```

We then need a function that will convert s-expressions into instances of this datatype. This is the other half of our parser:

```
fun parse(s :: S.S-Exp) -> ArithC:
  cases (S.S-Exp) s:
    | s-num(n) => numC(n)
    | s-list(shadow s) =>
```

```
      cases (List) s:
        | empty => raise("parse: unexpected empty list")
        | link(op, args) =>
          argL = L.get(args, 0)
          argR = L.get(args, 1)
          if op.s == "+":
            plusC(parse(argL), parse(argR))
          else if op.s == "*":
            multC(parse(argL), parse(argR))
          end
      end
    | else =>
      raise("parse: not number or list")
  end
end
```

This obeys the following tests:

Note the use of a helper function inside the block of tests.

```
check:
  fun p(s): parse(S.read-s-exp(s)) end
  p("3") is numC(3)
  p("(+ 1 2)") is plusC(numC(1), numC(2))
  p("(* (+ 1 2) (* 2 5))") is
    multC(plusC(numC(1), numC(2)), multC(numC(2), numC(5)))
end
```

Congratulations! You have just completed your first *representation of a program*. From now on we can focus entirely on programs represented as recursive trees, ignoring the vagaries of surface syntax and how to get them into the tree form (though in practice, we will continue to use the s-expression notation because it's easier to type than all those constructors). We're finally ready to start studying programming languages!

Exercise

If the test

```
p("3") is numC(3)
```

is instead written as

```
p(3) is numC(3)
```

what happens? Why?

23.2.3 Coda

"Recursive functions of symbolic expressions and their computation by machine, Part I" by John McCarthy in *Communications of the ACM*.

The s-expression syntax dates back to 1960. This syntax is often controversial amongst programmers. Observe, however, something deeply valuable that it gives us. While parsing traditional languages can be very complex, parsing this syntax is virtually trivial. Given a sequence of tokens corresponding to the input, it is absolutely straightforward to turn parenthesized sequences into s-expressions; it is equally straightforward (as we see above) to turn s-expressions into proper syntax trees. We like to call such two-level languages *bicameral*, in loose analogy to government legislative houses: the lower-level does rudimentary well-formedness checking, while the upper-level does deeper validity checking.

The term is introduced in PLAI.

The virtues of this syntax are thus manifold. The amount of code it requires is small, and can easily be embedded in many contexts. By integrating the syntax into the language, it becomes easy for programs to manipulate representations of programs (as we will see more of in [section 24.4]. It's therefore no surprise that even though many Lisp-based languages—from Lisp 1.5 to Common Lisp to Scheme to Racket to Clojure and more—have had wildly different semantics, they all share this syntactic legacy.

Of course, we could just use XML instead. That might be much nicer. Or JSON. Because that wouldn't be anything like an s-expression *at all*.

Chapter 24

Processing Programs: A First Look at Interpretation

Now we're ready to write an *evaluator*—a program that turns programs into answers—in the form of an *interpreter*, for our arithmetic language. We choose arithmetic first for three reasons: (a) you already know how it works, so we can focus on the mechanics of writing evaluators; (b) it's contained in every language we will encounter later, so we can build upwards and outwards from it; and (c) it's (surprisingly) sophisticated enough to convey some important points.

The term "evaluate" means "to reduce to a value".

24.1 Representing Arithmetic

Let's first agree on how we will represent arithmetic expressions. Let's say we want to support only two operations—addition and multiplication—in addition to primitive numbers. We need to represent arithmetic *expressions*. What are the rules that govern the nesting of arithmetic expressions? We're actually free to nest any expression inside another.

Do Now!

Why did we not include division? What impact does it have on the remarks above?

We've ignored division because it forces us into a discussion of what expressions we might consider legal: clearly the representation of `1/2` ought to be legal; the representation of `1/0` is much more debatable; and that of `1/(1-1)` seems even more controversial. We'd like to sidestep this controversy for now and return to it later [chapter 28].

Thus, we want a representation for numbers and arbitrarily nestable addition and multiplication. Here's one we can use:

```
data ArithC:
  | numC(n :: Number)
  | plusC(l :: ArithC, r :: ArithC)
  | multC(l :: ArithC, r :: ArithC)
end
```

24.2 Writing an Interpreter

Now let's write an interpreter for this arithmetic language. First, we should think about what its type is. It clearly consumes a `ArithC` value. What does it produce? Well, an interpreter evaluates—and what kind of value might arithmetic expressions reduce to? Numbers, of course. So the interpreter is going to be a function from arithmetic expressions to numbers.

Exercise

Write your examples for the interpreter.

Because we have a recursive datatype, it is natural to structure our interpreter as a recursive function over it. Here's a first template:

Templates are explained in detail in *How to Design Programs*.

```
fun interp(e :: ArithC) -> Number:
  cases (ArithC) e:
    | numC(n) => ...
    | plusC(l, r) => ...
    | multC(l, r) => ...
  end
end
```

You're probably tempted to jump straight to code, which you can:

```
fun interp(e :: ArithC) -> Number:
  cases (ArithC) e:
    | numC(n) => n
    | plusC(l, r) => l + r
    | multC(l, r) => l * r
  end
where:
  interp(numC(3)) is 3
end
```

which works just fine, passing its test.

Do Now!

Do you spot the errors?

Instead, let's expand the template out a step:

```
fun interp(e :: ArithC) -> Number:
  cases (ArithC) e:
    | numC(n) => ...
    | plusC(l, r) => ... interp(l) ... interp(r) ...
    | multC(l, r) => ... interp(l) ... interp(r) ...
  end
end
```

and now we can fill in the blanks:

```
fun interp(e :: ArithC) -> Number:
  cases (ArithC) e:
    | numC(n) => n
    | plusC(l, r) => interp(l) + interp(r)
    | multC(l, r) => interp(l) * interp(r)
  end
end
```

Later on [section 26.3], we're going to wish we had returned a more complex datatype than just numbers. But for now, this will do.

Congratulations: you've written your first interpreter! We know, it's very nearly an anticlimax. But they'll get harder—much harder—pretty soon, we promise.

24.3 A First Taste of "Semantics"

We just slipped something by you:

Do Now!

What is the "meaning" of addition and multiplication in this new language?

That's a pretty abstract question, isn't it. Let's make it concrete. I'll pose the problem as follows.

Which of these is the same?

- `1 + 2`

- `1 + 2`
- `'1' + '2'`
- `'1' + '2'`

What we're driving at is that there are many kinds of addition in computer science:

- First of all, there are many different kinds of *numbers*: fixed-width (e.g., 32-bit) integers, signed fixed-width (e.g., 31-bits plus a sign-bit) integers, arbitrary precision integers; in some languages, rationals; various formats of fixed- and floating-point numbers; in some languages, complex numbers; and so on. After the numbers have been chosen, addition may support only some combinations of them.

- In addition, some languages permit the addition of datatypes such as matrices.

- Furthermore, many languages support "addition" of strings (we use scare-quotes because we don't really mean the mathematical concept of addition, but rather the operation performed by an operator with the syntax `+`). In some languages this always means concatenation; in some others, it can result in numeric results (or numbers stored in strings).

These are all different "meanings for addition". *Semantics* is the mapping of syntax (e.g., `+`) to meaning (e.g., some or all of the above).

Returning to our interpreter, what semantics do we have? We've adopted whatever semantics Pyret provides, because we map `+` to Pyret's `+`. In fact that's not even quite true: Pyret may, for all we know, also enable `+` to apply to strings (which in fact it does), so we've chosen the restriction of Pyret's semantics to numbers.

Exercise

In what way have we restricted `+` to apply only to numbers? Where exactly is this restriction?

If we wanted a different semantics, we'd have to implement it explicitly.

Exercise

What all would you have to change so that the number had signed 32-bit arithmetic?

In general, we have to be careful about too readily borrowing from the host language. However, because we have lots of interesting things to study already, we will adopt Pyret's numbers as our numbers for now.

24.4 Desugaring: Growing the Language Without Enlarging It

We've picked a very restricted first language, so there are many ways we can grow it. Some, such as representing data structures and functions, will clearly force us to add new features to the interpreter itself. Others, such as adding more of arithmetic itself, can possibly be done without disturbing the core language and hence its interpreter: this is known as adding *syntactic sugar*, or "sugar" for short. Let's investigate.

24.4.1 Extension: Binary Subtraction

First, we'll add subtraction. Because our language already has numbers, addition, and multiplication, it's easy to define subtraction: $a - b = a + -1 \times b$.

Okay, that was easy! But now we should turn this into concrete code. To do so, we face a decision: where does this new subtraction operator reside? It is tempting, and perhaps seems natural, to just add one more case to our existing `ArithC` datatype.

Do Now!

What are the negative consequences of modifying `ArithC`?

This creates a few problems:

1. The first, obvious, one is that we now have to modify all programs that process `ArithC`. So far that's only our interpreter, which is pretty simple, but in a more complex implementation, there could be many programs built around the datatype—a type-checker, compiler, etc.—which must all be changed, creating a heavy burden.

2. Second, we were trying to add new constructs that we can define in terms of existing ones; it feels slightly self-defeating to do this in a way that isn't modular.

3. Third, and most subtly, there's something *conceptually* unnecessary about modifying `ArithC`. That's because `ArithC` represents a perfectly good *core* language. Atop this, we might want to include any number of additional operations that make the user's life more convenient, but there's no need to put these in the core. Rather, it's wise to record conceptually different ideas in distinct datatypes, rather than shoehorn them into one. The separation can look a little unwieldy sometimes, but it makes the program much easier for

future developers to read and maintain. Besides, for different purposes you might want to layer on different extensions, and separating the core from the surface enables that.

Therefore, we'll define a new datatype to reflect our intended surface syntax terms:
<arith-dt> ::=

```
  data ArithExt:
    | numExt (n :: Number)
    | plusExt (l :: ArithExt, r :: ArithExt)
    | multExt (l :: ArithExt, r :: ArithExt)
    | bminusExt (l :: ArithExt, r :: ArithExt)
    <uminus-dt>
  end
```

This looks almost exactly like `ArithC`, other than the added case, which follows the familiar recursive pattern. Note that the children of each node refer to `ArithExt`, not `ArithC`.

Do Now!

What happens if the children are declared to be `ArithC` rather than `ArithExt`?

If we did this, then we would be able to use sugar only at the top-level, not in any sub-expressions.

Given this datatype, we should do two things. First, we should modify our parser to also parse `-` expressions, and always construct `ArithExt` terms (rather than any `ArithC` ones). Second, we should implement a `desugar` function that translates `ArithExt` values into `ArithC` ones.

Desugaring is the act of removing syntactic sugar.

Let's write the obvious part of `desugar`:
<main> ::=

```
  fun desugar(s :: ArithExt) -> ArithC:
    cases (ArithExt) s:
      | numExt(n) => numC(n)
      | plusExt(l, r) => plusC(desugar(l), desugar(r))
      | multExt(l, r) => multC(desugar(l), desugar(r))
      <bminus>
      <uminus>
    end
  end
```

Now let's convert the mathematical description of subtraction above into code:
<bminus> ::=

```
  | bminusExt(l, r) =>
    plusC(desugar(l), multC(numC(-1), desugar(r)))
```

Do Now!

It's a common mistake to forget the recursive calls to `desugar` on `l` and `r`. What happens when you forget them? Try for yourself and see.

24.4.2 Extension: Unary Negation

Now let's consider another extension, which is a little more interesting: unary negation. This forces you to do a little more work in the parser because, depending on your surface syntax, you may need to look ahead to determine whether you're in the unary or binary case. But that's not even the interesting part!

Exercise

Modify `parse` to handle unary subtraction.

There are many ways we can desugar unary negation. We can define it naturally as $-b = 0 - b$, or we could abstract over the desugaring of binary subtraction with this expansion: $-b = 0 + -1 \times b$.

Do Now!

Which one do you prefer? Why?

It's tempting to pick the first expansion, because it's much simpler. Imagine we've extended the `ArithExt` datatype with a representation of unary negation:
<uminus-dt> ::=

```
  | uminusExt (e :: ArithExtU)
```

Now the implementation in `desugar` is straightforward:
<uminus> ::=

```
  | uminusExt(e) => desugar(bminusExt(numExt(0), e))
```

Let's make sure the types match up. Observe that `e` is a `ArithExt` term, so it is valid to use as an argument to `bminusExt`, and the entire term can legally be passed to `desugar`. It is therefore important to *not* desugar `e` but rather embed it directly in the generated term. This embedding of an input term in another one and recursively calling desugar is a common pattern in desugaring tools; it is called a *macro* (specifically, the "macro" here is this definition of `uminusExt`).

However, there are two problems with the definition above:

If you haven't heard of generative recursion before, read the section on it in *How to Design Programs*. Essentially, in generative recursion the sub-problem is a computed function of the input, rather than a structural piece of it. This is an especially simple case of generative recursion, because the "function" is simple: it's just the `bminusExt` constructor.

1. The first is that the recursion is *generative*, which forces us to take extra care. We might be tempted to fix this by using a different rewrite:

   <*uminus/alt*> ::=

   ```
   | uminusExt(e) => bminusExt(numExt(0), desugar(e))
   ```

   which does indeed eliminate the generativity.

   Do Now!

   Unfortunately, this desugaring transformation won't work at all! Do you see why? If you don't, try to run it.

2. The second is that we are implicitly depending on exactly what `bminusExt` means; if its meaning changes, so will that of `uminusExt`, even if we don't want it to. In contrast, defining a functional abstraction that consumes two terms and generates one representing the addition of the first to -1 times the second, and using this to define the desugaring of both `uminusExt` and `bminusExt`, is a little more fault-tolerant.

   You might say that the meaning of subtraction is never going to change, so why bother? Yes and no. Yes, it's *meaning* is unlikely to change; but no, its *implementation* might. For instance, the developer may decide to log all uses of binary subtraction. In the first expansion all uses of unary negation would also get logged, but they would not in the second expansion.

Fortunately, in this particular case we have a much simpler option, which is to define $-b = -1 \times b$. This expansion works with the primitives we have, and follows structural recursion. The reason we took the above detour, however, is to alert you to these problems, and warn that you might not always be so fortunate.

24.5 A Three-Stage Pipeline

This concludes our first look at the standard pipeline we're going to use. We will first parse programs to convert them to abstract syntax; we will then desugar them to eliminate unnecessary constructs. From now on, we will usually focus just on the resulting core language, which will be subject to not only interpretation but also type-checking and other actions.

Chapter 25

Interpreting Conditionals

Now that we have the first stirrings of a programming language, let's grow it out a little. The heart of a programming language consists of *control*—the ability to order executed instructions—and *data*—the representations of information consumed and produced by programs. We will therefore add both control and data to our language, through a simple and important kind: conditionals. Though this seems (and is) a very simple concept, it will force us to tackle several design issues in both the language and the interpreter's pipeline, and is thus surprisingly illuminating.

25.1 The Design Space of Conditionals

Even the simplest conditional exposes us to many variations in language design. Consider one of the form:

```
(if test-exp then-part else-part)
```

The intent is that `test-exp` is evaluated first; if it results in a true value then (only) `then-part` is evaluated, else (only) `else-part` is evaluated. (We usually refer to these two parts as *branches*, since the program's control must take one or the other.) However, even this simple construct results in at least three different, mostly independent design decisions:

1. What kind of values can the `test-exp` be? In some languages they must be Boolean values (two values, one representing truth and the other falsehood). In other languages this expression can evaluate to just about any value, with some set—colloquially called *truthy*—representing truth (i.e., they result in execution of the `then-part`) while the remaining ones are *falsy*, meaning they cause `else-part` to run.

Initially, it may seem attractive to design a language with several truthy and falsy values: after all, this appears to give the programmer more convenience, permitting non-Boolean-valued functions and expressions to be used in conditionals. However, this can lead to bewildering inconsistencies across languages:

Value	**JS**	**Perl**	**PHP**	**Python**	**Ruby**
`0`	falsy	falsy	falsy	falsy	truthy
`""`	falsy	falsy	falsy	falsy	truthy
`NaN`	falsy	truthy	truthy	truthy	truthy
nil/null/None/undef	falsy	falsy	falsy	falsy	falsy
`"0"`	truthy	falsy	falsy	truthy	truthy
`-1`	truthy	truthy	truthy	truthy	truthy
`[]`	truthy	truthy	falsy	falsy	truthy
empty map/object	truthy	falsy	falsy	falsy	truthy

Of course, it need not be so complex. Scheme, for instance, has only *one* value that is falsy: false itself (written as `#f`). *Every* other value is truthy. For those who value allowing non-Boolean values in conditionals, this represents an elegant trade-off: it means a function need not worry that a type-consistent value resulting from a computation might cause a conditional to reverse itself. (For instance, if a function returns strings, it need not worry that the empty string might be treated differently from every other string.) Note that Ruby and Lua have relatively few falsy values; it may not be coincidental that their creators were deeply influenced by Scheme.

While writing this chapter, I stumbled on a strange bug in Pyret: all numeric s-expressions parsed as `s-num` values except `0`, which parsed as a `s-sym`. Eventually Justin Pombrio reported: "It's a silly bug with a `if` in JavaScript that's getting `0` and thinking it's false."

2. What kind of terms are the branches? Some languages make a distinction between *statements* and *expressions*; in such languages, designers need to decide which of these are permitted. In some languages, there are even two syntactic forms of conditional to reflect these two choices: e.g., in C, `if` uses statements (and does not return any value) while the "ternary operator" (`(...?...:...)`) permits expressions and returns a value.

3. If the branches are expressions and hence allowed to evaluate to values, how do the values relate? Many (but not all) languages with static type systems expect the two branches to have the same type [section 27.3]. Languages without static type systems usually place no restrictions.

For now, we will assume that the conditional expression can only be a Boolean value; the branches are expressions (because that is all we have in our language anyway); and the two branches can return values of different types.

25.2 The Game Plan for Conditionals

To add conditionals to the language, we have to cover a surprising amount of ground:

- First, we need to define syntax. We'll use

  ```
  true
  false
  (if test-exp then-exp else-exp)
  ```

  to represent the two Boolean constants and the conditional expression.

- We need to modify the representation of programs to handle these new constructs. Here's our new expression language (with the name adjusted to signal its growth beyond pure arithmetic):

  ```
  data ExprC:
    | trueC
    | falseC
    | numC(n :: Number)
    | plusC(l :: ExprC, r :: ExprC)
    | multC(l :: ExprC, r :: ExprC)
    | ifC(c :: ExprC, t :: ExprC, e :: ExprC)
  end
  ```

  We need to adjust the pre-desugaring language (`ExprExt`) as well to account for the new constructs.

- We need to modify the parser and desugarer.

Exercise

Modify `parse` and `desugar` to work with the extended language. Adjust the datatypes as needed. Be sure to write tests.

Do Now!

There's one more big change needed. Do you see what it is?

25.2.1 The Interpreter's Type

If our terms are no longer purely arithmetic in nature, then we can no longer expect our interpreter to produce only numeric answers! For instance, what should be the result of evaluating the following program:

```
trueC
```

(that is, the program corresponding to the source text `true`)?

"A lesser man might have wavered that day in the hospital corridor, a weaker man might have compromised on such excellent substitutes as Drum Major, Minor Major, Sergeant Major, or C. Sharp Major, but Major Major's father had waited fourteen years for just such an opportunity, and he was not a person to waste it."—Joseph Heller

Beware: what I've presented you with is actually a test of your character! You might be sorely tempted to decide that `true` should evaluate to `1` (and, for good measure, that `false` should evaluate to `0`). What are the consequences of this?

- It precludes being able to have a language with pure Booleans. This will have consequences when we get to types [chapter 27].
- It means you can perform arithmetic on truth values. This might not sound so surprising: after all, conjunction (and) and disjunction (or) can, after all, be thought in terms of arithmetic. But once you you say truth values *are* numbers, you can no longer detect if a programmer accidentally subtracts one truth value from another, divides them, and so on.
- It isn't even clear which numbers should represent which truth values. Historically, some languages have made zero represent truth; others have even chosen to use non-negative numbers for truth and negative numbers for falsity. None of these choices is more clearly "correct" than other ones, which suggests we're really just guessing our way around here.
- Most of all, we can't keep hacking our way out of this situation. How are we going to represent strings or lists? With Gödel numbering? What happens when we get to functions, coroutines, continuations?

In short, *avoid encoding*! There's no good reason to make numbers do double-duty: let Booleans be their own type. In any respectable implementation this will impose little to no additional cost to program execution, while greatly reducing programmer confusion.

Of course, you're welcome to experiment with different decisions. The beauty of writing little interpreters is you can change what you want and explore the consequences of those changes.

The consequence of this decision is that we will need a way to represent all the possible outcomes from the interpreter.

Do Now!

Try to sketch a representation for yourself.

Here's a reasonable one:

```
data Value:
  | numV(n :: Number)
  | boolV(b :: Boolean)
end
```

For now, this is naturally a quite shallow representation: it simply helps us tell numbers and Booleans apart. Later, we will add values that have much more interesting structure.

25.2.2 Updating Arithmetic

Finally, we're ready to augment our interpreter. We can ignore the arithmetic lines, which should be unchanged (because we haven't changed anything about how we will perform arithmetic), and focus on the new parts of the language.

Do Now!

Right?

Wrong. Because we've changed the type of value the interpreter produces, we have to update the rules for arithmetic, too, to reflect that. We can do this quickly, but we'll do it in a few steps to illustrate a point.

First, we'll handle the easy case:

<ext-arith-cond-interp> ::=

```
  fun interp(e :: ExprC) -> Value:
    cases (ExprC) e:
      | numC(n) => numV(n)
      <ext-arith-cond-arith-cases>
      <ext-arith-cond-bool-cases>
    end
  end
```

Now let's consider addition and multiplication. We could do it directly:

```
| plusC(l, r) =>
  numV(interp(l).n + interp(r).n)
```

but this will get repetitive: calling `interp` on each branch, dereferencing the numeric field, and wrapping the answer in `numV`. It would be better to abstract over this so we don't have to repeat code.

25.2.3 Defensive Programming

Actually, there is a more serious problem lurking in the above code. It's in this expression:

```
interp(l).n
```

(and in the other, similar one). First, it blindly refers to the `n` field irrespective of whether the resulting value actually represents a number; if some other variant also had a field of this name, this would silently succeed! Therefore, we really ought to make sure that the recursive call to `interp` really does return a number; and now the logic is clearly getting more complicated than we would like to do in-line. Instead, we'll define a helper function that takes the operation to perform and does everything else:

```
fun arith-binop(op :: (Number, Number -> Number),
    l :: ExprC,
    r :: ExprC) -> Value:
  l-v = interp(l)
  r-v = interp(r)
  if is-numV(l-v) and is-numV(r-v):
    numV(op(l-v.n, r-v.n))
  else:
    raise('argument not a number')
  end
end
```

With this, we can now show the revised definitions of the arithmetic operations:
<ext-arith-cond-arith-cases> ::=

```
  | plusC(l, r) => arith-binop(lam(x, y): x + y end, l, r)
  | multC(l, r) => arith-binop(lam(x, y): x * y end, l, r)
```

Do Now!

Before we move on, let's ponder one more question. Suppose we could be certain that no other variant would have a field named `n`. Then, is there any difference between the version that checks `is-numV` for each of the values and the version that does not?

Seemingly, the answer is no: if there is no `n` field, the version that accesses `n` will halt with an error signaled by Pyret, just as `arith-binop` would have stopped it. However, there is an important philosophical difference between the two versions:

- The version that performs a check in `arith-binop` is providing the error *at the level of the language being implemented*. It does not depend on Pyret to perform any checks; furthermore, it can give an error in terms of the interpreted language, using terminology that makes sense to the programmer in that language.

- In contrast, the version that delegates the check to Pyret is allowing a *meta-error* to percolate through. This requires being very certain of how Pyret works, whether it will perform the check at the right time and in the right place, and then halt program execution in the appropriate way. Furthermore, the error message it produces might make no sense to the programmer: Pyret might say "Field `n` not found", but to a person using a language with only arithmetic and conditionals, the very term "field" might mean nothing.

Thus, in a production system, we should be sure to catch errors in our implementation, not hope that the implement*ing* language will do the right thing. In this study, however, we will sometimes be loose about this, to keep the code simpler and more readable.

25.2.4 Interpreting Conditionals

Finally, we are ready to handle the actual point of this exercise: conditionals. The two constants are easy:

<ext-arith-cond-bool-cases> ::=

```
  | trueC => boolV(true)
  | falseC => boolV(false)
```

  <ext-arith-cond-bool-if>

The conditional expression is not actually hard; it just forces us to think. This is where we encode truthy/falsy distinctions; indeed, one could arguably have *three possibilities*: true values, false values, and other values! (A third possibility here is tantamount to having an equality operator returning anything other than true and false values. Though highly unusual, it is neither impossible nor nonexistent: in fact, Pyret does this [section 21.6.3]!) However, because we've decided that we will only handle Boolean values, and there are only two kinds of these (not least because we've chosen to represent them using Pyret's Booleans), this becomes quite simple:

Thinking, after all, rather being the point of this entire study.

<ext-arith-cond-bool-if> ::=

```
  | ifC(cnd, thn, els) =>
    ic = interp(cnd)
    if is-boolV(ic):
      if ic.b:
        interp(thn)
      else:
        interp(els)
      end
    else:
      raise('not a boolean')
    end
```

And that's it. Now we have a complete, working interpreter for conditionals.

25.3 Growing the Conditional Language

To create a truly useful language of conditionals, however, we need at least two more things:

1. A way to *compute* Boolean values, not just write them as constants. For instance, we should add operations on numbers (such as numeric comparison). This is relatively easy, especially given that we already have `arith-binop` parameterized over the operation to perform and returning a `Value` (rather than a number). The bigger nuisance is pushing this through parsing and desugaring. It would instead be better to create generic unary and binary operations and look them up in a table.

2. It would also be useful to have a way to combine conditionals (negation, disjunction, conjunction).

Exercise

Generalize the parser and desugarer to look up a table of unary and binary operations and represent them uniformly, instead of having a different variant for each one.

Negation is straightforward: it's just a unary function. However, in a programming language, disjunction (or) and conjunction (and) should not be thought of as functions. For instance, in Scheme, it is common to write:

```
(and (not (= x 0)) (/ 1 x))
```

If both `(not (= x 0))` and `(/ 1 x)` were treated as arguments and evaluated right away, then the very situation we're trying to protect against—division by zero—would occur right away. Therefore, it is better to think of these are desugaring into cascading conditionals: for instance, one possible desugaring for the above expression might be

Of course, this is not a problem in a lazy language [REF].

```
(if (not (= x 0))
    false
    (/ 1 0))
```

Exercise

Implement negation, conjunction, and disjunction.

Exercise

Define a multi-armed conditional expression that desugars into nested `if`s.

Chapter 26

Interpreting Functions

26.1 Adding Functions to the Language

Now that we have basic expressions and conditionals, let's grow to have a complete programming languageby adding functions.

26.1.1 Defining Data Representations

Imagine we're modeling a simple programming environment. The developer defines functions in a definitions window, and uses them in an interactions window (which provides a prompt at which they can run expressions). For now, let's assume all definitions go in the definitions window only (we'll relax this soon: section 26.3), and all stand-alone expressions in the interactions window only. Thus, running a program simply loads definitions. Our interpreter will correspond to the interactions window prompt and assume it has been supplied with a set of definitions.

For historic reasons, the interactions window is also called a REPL or "*r*ead-*e*val-*p*rint *l*oop".

A *set* of definitions suggests no ordering, which means, presumably, any definition can refer to any other. That's what we intend here, but when you are designing your own language, be sure to think about this.

To keep things simple, let's just consider functions of one argument. Here are some Pyret examples:

```
fun double(x): x + x end

fun quad(x): double(double(x)) end

fun const5(_): 5 end
```

Exercise

When a function has multiple arguments, what simple but important criterion governs the names of those arguments?

What are the parts of a function definition? It has a name (above, `double`, `quad`, and `const5`), which we'll represent as a string (`"double"`, etc.); its *formal parameter* or *argument* has a name (e.g., `x`), which too we can model as a string (`"x"`); and it has a body. We'll determine the body's representation in stages, but let's start to lay out a datatype for function definitions:

```
data FunDefC:
  | fdC(name :: String, arg :: String, body :: ExprC)
end
```

What is the body? Clearly, it has the form of an arithmetic expression, and sometimes it can even be represented using the existing `ArithC` language: for instance, the body of `const5` can be represented as `numC(5)`. But representing the body of `double` requires something more: not just addition (which we have), but also "`x`". You are probably used to calling this a *variable*, but we will *not* use that term for now. Instead, we will call it an *identifier*.

We've discussed this terminological difference in section 21.4.

Do Now!

Anything else?

Finally, let's look at the body of `quad`. It has yet another new construct: a function *application*. Be very careful to distinguish between a function *definition*, which describes what the function *is*, and an *application*, which *uses* it. The *argument* (or *actual parameter*) in the inner application of `double` is `x`; the argument in the outer application is `double(x)`. Thus, the argument can be any complex expression.

Let's commit all this to a crisp datatype. Clearly we're extending what we had before (because we still want all of arithmetic). We'll give a new name to our datatype to signify that it's growing up:

<datatype> ::=

```
  data ExprC:
    | numC(n :: Number)
    | plusC(l :: ExprC, r :: ExprC)
    | multC(l :: ExprC, r :: ExprC)
    | trueC
    | falseC
    | ifC(c :: ExprC, t :: ExprC, e :: ExprC)
    | <appC-dt>
    | <idC-dt>
  end
```

Identifiers are closely related to formal parameters. When we apply a function by giving it a value for its parameter, we are in effect asking it to replace all instances of that formal parameter in the body—i.e., the identifiers with the same name as the formal parameter—with that value. To simplify this process of search-and-replace, we might as well use the same datatype to represent both. We've already chosen strings to represent formal parameters, so:

Observe that we are being coy about a few issues: what kind of "value" and when to replace [REF].

<idC-dt> ::=
```
  | idC(s :: String)
```
Finally, applications. They have two parts: the function's name, and its argument. We've already agreed that the argument can be any full-fledged expression (including identifiers and other applications). As for the function name, it again makes sense to use the same datatype as we did when giving the function its name in a function definition. Thus:

<appC-dt> ::=
```
  | appC(f :: String, a :: ExprC)
```
identifying which function to apply, and providing its argument.

Using these definitions, it's instructive to write out the representations of the examples we defined above:

- `fdC("double", "x", plusC(idC("x"), idC("x")))`
- `fdC("quad", "x", appC("double", appC("double", idC("x"))))`
- `fdC("const5", "_", numC(5))`

We also need to choose a representation for a set of function definitions. It's convenient to represent these by a list.

Look out! Did you notice that we spoke of a *set* of function definitions, but chose a *list* representation? That means we're using an ordered collection of data to represent an unordered entity. At the very least, then, when testing, we should use any and all permutations of definitions to ensure we haven't subtly built in a dependence on the order.

Exercise

Extend `desugar` with support for identifiers and applications.

26.1.2 Growing the Interpreter

Now we're ready to tackle the interpreter proper. First, let's remind ourselves of what it needs to consume. Previously, it consumed only an expression to evaluate. Now it also needs to take a list of function definitions:

<fof-interp> ::=
```
  fun interp(e :: ExprC, fds :: List<FunDefC>) -> Value:
    cases (ExprC) e:
      <fof-interp-body>
    end
  end
```

Let's revisit our old interpreter. In the case of numbers, clearly we still return the number as the answer. In the addition and multiplication case, we still need to recur (because the sub-expressions might be complex), but which set of function definitions do we use? Because the act of evaluating an expression neither adds nor removes function *definitions*, the set of definitions remains the same, and should just be passed along unchanged in the recursive calls. Similarly for conditionals.

<fof-interp-body> ::=

```
  | numC(n) => numV(n)
  | plusC(l, r) => arith-binop(lam(x, y): x + y end, l, r, fds)
  | multC(l, r) => arith-binop(lam(x, y): x * y end, l, r, fds)
  | trueC => boolV(true)
  | falseC => boolV(false)
  | ifC(cnd, thn, els) =>
    ic = interp(cnd, fds)
    if is-boolV(ic):
      if ic.b:
        interp(thn, fds)
      else:
        interp(els, fds)
      end
    else:
      raise('not a boolean')
    end
  <fof-interp-idC>
  <fof-interp-appC>
```

Exercise

Modify `arith-binop` to pass along `fds` unchanged in recursive calls.

Now let's tackle application. First we have to look up the function definition, for which we'll assume we have a helper function of this type available:

<get-fundef> ::=

```
  fun get-fundef(name :: String, fds :: List<FunDefC>)
      -> FunDefC:
    <get-fundef-body>
  end
```

Assuming we find a function of the given name, we need to evaluate its body. However, remember what we said about identifiers and parameters? We must "search-and-replace", a process you have seen before in school algebra called *substitution*.

This is sufficiently important that we should talk first about substitution before returning to the interpreter [section 26.1.4].

26.1.3 Substitution

Substitution is the act of replacing a name (in this case, that of the formal parameter) in an expression (in this case, the body of the function) with another expression (in this case, the actual parameter). Its header is:

<subst> ::=
```
  fun subst(w :: ExprC, at :: String, in :: ExprC) -> ExprC:
    <subst-body>
  end
```
The first argument is what we want to replace the name *with*; the second is *at* what name we want to perform substitution; and the third is *in* which expression we want to do it.

Do Now!

Suppose we want to substitute `3` for the identifier `x` in the bodies of the three example functions above. What should it produce?

In `double`, this should produce `3 + 3`; in `quad`, it should produce `double(double(3))`; and in `const5`, it should produce `5` (i.e., no substitution happens because there are no instances of `x` in the body).

A common mistake is to assume that the result of substituting, e.g., `3` for `x` in `double` is `fun double(x): 3 + 3 end`. This is incorrect. We only substitute *at the point when we apply the function*, at which point the function's invocation is replaced by its body. The header enables us to find the function and ascertain the name of its parameter; but only its body participates in evaluation. Examine the use of substitution in the interpreter to see how returning a function *definition* would result in a type error.

These examples already tell us what to do in almost all the cases. Given a number, there's nothing to substitute. If it's an identifier, we have to to replace the identifier if it's the one we're trying to substitute, otherwise leave it alone. In the other cases, descend into the sub-expressions, performing substitution.

Before we turn this into code, there's an important case to consider. Suppose the name we are substituting happens to be the name of a function. Then what should happen?

Do Now!

What, indeed, should happen?

There are many ways to approach this question. One is from a design perspective: function names live in their own "world", distinct from ordinary program identifiers. Some languages (such as C and Common Lisp, in slightly different ways) take this perspective, and partition identifiers into different *namespaces* depending on how they are used. In other languages, there is no such distinction; indeed, we will examine such languages soon [section 26.3].

For now, we will take a pragmatic viewpoint. If we evaluate a function name, it would result in a number or Boolean. However, these cannot name functions. Therefore, it makes no sense to substitute in that position, and we should leave the function name unmolested irrespective of its relationship to the variable being substituted. (Thus, a function could have a parameter named `x` as well as refer to another *function* called `x`, and these would be kept distinct.)

Now we've made all our decisions, and we can provide the body:

<subst-body> ::=

```
cases (ExprC) in:
  | numC(n) => in
  | plusC(l, r) => plusC(subst(w, at, l), subst(w, at, r))
  | multC(l, r) => multC(subst(w, at, l), subst(w, at, r))
  | trueC => trueC
  | falseC => falseC
  | ifC(cnd, thn, els) =>
    ifC(subst(w, at, cnd), subst(w, at, thn), subst(w, at, els))
  | appC(f, a) => appC(f, subst(w, at, a))
  | idC(s) =>
    if s == at:
      w
    else:
      in
    end
end
```

Exercise

Observe that, whereas in the `numC` case the interpreter returned `numV(n)`, substitution returns `in` (i.e., the original expression, equivalent at that point to writing `numC(n)`). Why?

26.1.4 The Interpreter, Resumed

Phew! Now that we've completed the definition of substitution (or so we think), let's complete the interpreter. Substitution was a heavyweight step, but it also does much of the work involved in applying a function. It is tempting to write

<fof-interp-appC/alt> ::=

```
| appC(f, a) =>
  fd = get-fundef(f, fds)
  subst(a, fd.arg, fd.body)
```

Tempting, but wrong.

Do Now!

Do you see why?

Reason from the types. What does the interpreter return? Values. What does substitution return? Oh, that's right, expressions! For instance, when we substituted in the body of `double`, we got back the representation of `5 + 5`. This is not a valid answer for the interpreter. Instead, it must be reduced to an answer. That, of course, is precisely what the interpreter does:

<fof-interp-appC> ::=

```
  | appC(f, a) =>
    fd = get-fundef(f, fds)
    interp(subst(a, fd.arg, fd.body), fds)
```

Okay, that leaves only one case: identifiers. What could possibly be complicated about them? They should be just about as simple as numbers! And yet we've put them off to the very end, suggesting something subtle or complex is afoot.

Do Now!

Work through some examples to understand what the interpreter should do in the identifier case.

Let's suppose we had defined `double` as follows:

```
fun double(x): x + y end
```

When we substitute `5` for `x`, this produces the expression `5 + y`. So far so good, but what is left to substitute `y`? As a matter of fact, it should be clear from the very outset that this definition of `double` is *erroneous*. The identifier `y` is said to be *free*, an adjective that in this setting has negative connotations.

In other words, the interpreter should never confront an identifier. All identifiers ought to be parameters that have already been substituted (known as *bound* identifiers—here, a positive connotation) before the interpreter ever sees them. As a result, there is only one possible response given an identifier:

<fof-interp-idC> ::=

```
  | idC(s) => raise("unbound identifier")
```

And that's it!

Finally, to complete our interpreter, we should define `get-fundef`:

<get-fundef-body> ::=

```
  cases (List<FunDefC>) fds:
    | empty => raise("couldn't find function")
    | link(f, r) =>
```

```
      if f.name == name:
        f
      else:
        get-fundef(name, r)
      end
  end
```

26.1.5 Oh Wait, There's More!

Earlier, we declared `subst` as:

```
fun subst(w :: ExprC, at :: String, in :: ExprC) -> ExprC:
  ...
end
```

Sticking to surface syntax for brevity, suppose we apply `double` to `1 + 2`. This would substitute `1 + 2` for each `x`, resulting in the following expression—`(1 + 2) + (1 + 2)`—for interpretation. Is this necessarily what we want?

When you learned algebra in school, you may have been taught to do this differently: first reduce the argument to an answer (in this case, `3`), then substitute the answer for the parameter. This notion of substitution might have the following type instead:

```
fun subst(w :: Value, at :: String, in :: ExprC) -> ExprC:
  ...
end
```

In fact, we don't even have substitution quite right! The version of substitution we have doesn't scale past this language due to a subtle problem known as "name capture". Fixing substitution is complex, subtle, and an exciting intellectual endeavor, but it's not the direction we want to go in here. We'll instead sidestep this problem in this book. If you're interested, however, read about the *lambda calculus* [CITE], which provides the tools for defining substitution correctly.

Exercise

Modify your interpreter to substitute names with answers, not expressions.

We've actually stumbled on a profound distinction in programming languages. The act of evaluating arguments before substituting them in functions is called *eager* application, while that of deferring evaluation is called *lazy*—and has some variations. For now, we will actually prefer the eager semantics, because this is what most mainstream languages adopt. Later [REF], we will return to talking about the lazy application semantics and its implications.

26.2 From Substitution to Environments

Though we have a working definition of functions, you may feel a slight unease about it. When the interpreter sees an identifier, you might have had a sense that it needs to “look it up”. Not only did it not look up anything, we defined its behavior to be an error! While absolutely correct, this is also a little surprising. More importantly, we write interpreters to *understand* and *explain* languages, and this implementation might strike you as not doing that, because it doesn’t match our intuition.

There’s another difficulty with using substitution, which is the number of times we traverse the source program. It would be nice to have to traverse only those parts of the program that are actually evaluated, and then, only when necessary. But substitution traverses everything—unvisited branches of conditionals, for instance—and forces the program to be traversed once for substitution and once again for interpretation.

Exercise

Does substitution have implications for the time complexity of evaluation?

There’s yet another problem with substitution, which is that it is defined in terms of representations of the program source. Obviously, our interpreter has and needs access to the source, to interpret it. However, other implementations—such as compilers—have no need to store it for that purpose. It would be nice to employ a mechanism that is more portable across implementation strategies.

Compilers might store versions of or information about the source for other reasons, such as reporting runtime errors, and JITs may need it to re-compile on demand.

26.2.1 Introducing the Environment

The intuition that addresses the first concern is to have the interpreter “look up” an identifier in some sort of directory. The intuition that addresses the second concern is to *defer* the substitution. Fortunately, these converge nicely in a way that also addresses the third. The directory records the *intent to substitute*, without actually rewriting the program source; by recording the intent, rather than substituting immediately, we can defer substitution; and the resulting data structure, which is called an *environment*, avoids the need for source-to-source rewriting and maps nicely to low-level machine representations. Each name association in the environment is called a *binding*.

This does not mean our study of substitution was useless; to the contrary, many tools that work over programs—such as compilers and analyzers—use substitution. Just not for the purpose of evaluating it at run-time.

Observe carefully that what we are changing is the *implementation strategy* for the programming language, *not the language itself*. Therefore, none of our datatypes for representing programs should change, neither—and this is the critical part—should the answers that the interpreter provides. As a result, we should think of the previous interpreter as a “reference implementation” that the one we’re about to write should match. Indeed, we should create a generator that creates lots of tests, runs them through both interpreters, and makes sure their answers are the same: i.e., the previous implementation is an oracle [section 14.4]. Ideally, we should *prove* that the two interpreters behave the same, which is a good topic for advanced study.

One subtlety is in defining precisely what "the same" means, especially with regards to failure.

Let's first define our environment data structure. An environment is a collection of names associated with...what?

Do Now!

A natural question to ask here might be what the environment maps names to. But a better, more fundamental, question is: How to determine the answer to the "natural" question?

Remember that our environment was created to defer substitutions. Therefore, the answer lies in substitution. We discussed earlier [section 26.1.5] that we want substitution to map names to answers, corresponding to an eager function application strategy. Therefore, the environment should map names to answers.

```
data Binding:
  | bind(name :: String, value :: Value)
end

type Environment = List<Binding>
mt-env = empty
xtnd-env = link
```

26.2.2 Interpreting with Environments

Now we can tackle the interpreter. One case is easy, but we should revisit all the others:

<fof-env-interp> ::=

```
fun interp(e :: ExprC, nv :: Environment, fds :: List<FunDefC>) -> Va
  cases (ExprC) e:
    <fof-env-interp-arith>
    <fof-env-interp-cond>
    <fof-env-interp-idC>
    <fof-env-interp-appC>
  end
end
```

The arithmetic operations are easiest. Recall that before, the interpreter recurred without performing any new substitutions. As a result, there are no new deferred substitutions to perform either, which means the environment does not change:

<fof-env-interp-arith> ::=

```
  | numC(n) => numV(n)
  | plusC(l, r) => arith-binop(lam(x, y): x + y end, l, r, nv, fds)
  | multC(l, r) => arith-binop(lam(x, y): x * y end, l, r, nv, fds)
```

Conditionals are similarly straightforward:

<fof-env-interp-cond> ::=

```
  | trueC => boolV(true)
  | falseC => boolV(false)
  | ifC(cnd, thn, els) =>
    ic = interp(cnd, nv, fds)
    if is-boolV(ic):
      if ic.b:
        interp(thn, nv, fds)
      else:
        interp(els, nv, fds)
      end
    else:
      raise('not a boolean')
    end
```

Now let's handle identifiers. Clearly, encountering an identifier is no longer an error: this was the very motivation for this change. Instead, we must look up its value in the directory:

<fof-env-interp-idC> ::=

```
  | idC(s) => lookup(s, nv)
```

Do Now!

Implement `lookup`.

Finally, application. Observe that in the substitution interpreter, the only case that caused new substitutions to occur was application. Therefore, this should be the case that constructs bindings. Let's first extract the function definition, just as before:

<fof-env-interp-appC> ::=

```
  | appC(f, a) =>
    fd = get-fundef(f, fds)
    <fof-env-interp-appC-rest>
```

Previously, we substituted, then interpreted. Because we have no substitution step, we can proceed with interpretation, so long as we record the deferral of substitution. Let's also evaluate the argument:

<fof-env-interp-appC-rest> ::=

```
  arg-val = interp(a, nv, fds)
  interp(fd.body, <fof-env-interp-appC-rest-xtnd>, fds)
```

That is, the set of function definitions remains unchanged; we're interpreting the body of the function, as before; but we have to do it in an environment that binds the formal parameter. Let's now define that binding process:

<fof-env-interp-appC-rest-xtnd> ::=

```
  xtnd-env(bind(fd.arg, arg-val), nv)
```

But we'll return to this.

The name being bound is the formal parameter (the same name that was substituted for, before). It is bound to the result of interpreting the argument (because we've decided on an eager application semantics). And finally, this extends the environment we already have. Type-checking this helps to make sure we got all the little pieces right.

Once we have a definition for `lookup`, we'd have a full interpreter. So here's one:

<fof-env-interp-lookup> ::=

```
  fun lookup(s :: String, nv :: Environment) -> Value:
    cases (List) nv:
      | empty => raise("unbound identifier: " + s)
      | link(f, r) =>
        if s == f.name:
          f.value
        else:
          lookup(s, r)
        end
    end
  end
```

Observe that looking up a free identifier still produces an error, but it has moved from the interpreter—which is by itself unable to determine whether or not an identifier is free—to `lookup`, which determines this based on the content of the environment.

Now we have a full interpreter. You should of course test it make sure it works as you'd expect. Let's first set up some support code for testing:

<fof-env-interp-tests-setup> ::=

```
  check:
    f1 = fdC("double", "x", plusC(idC("x"), idC("x")))
    f2 = fdC("quad", "x", appC("double", appC("double", idC("x"))))
    f3 = fdC("const5", "_", numC(5))
    f4 = fdC("f4", "x", s2p2d("(if x 1 0)"))
    funs = [list: f1, f2, f3, f4]
    fun i(e): interp(e, mt-env, funs) end
```

<fof-env-interp-tests>
For instance, these tests pass:
<fof-env-interp-tests> ::=
```
  i(plusC(numC(5), appC("quad", numC(3)))) is numV(17)
  i(multC(appC("const5", numC(3)), numC(4))) is numV(20)
  i(plusC(numC(10), appC("const5", numC(10)))) is numV(10 + 5)
  i(plusC(numC(10), appC("double", plusC(numC(1), numC(2)))))
  is numV(10 + 3 + 3)
  i(plusC(numC(10), appC("quad", plusC(numC(1), numC(2)))))
  is numV(10 + 3 + 3 + 3 + 3)
```
<fof-env-interp-another-test>
So we're done, right?

Do Now!

Spot the bug.

26.2.3 Deferring Correctly

Here's another test:
<fof-env-interp-another-test> ::=
```
  interp(appC("f1", numC(3)), mt-env,
    [list: fdC("f1", "x", appC("f2", numC(4))),
      fdC("f2", "y", plusC(idC("x"), idC("y")))])
  raises "unbound identifier: x"
```

`raise` is explained earlier: section 14.5.

In our interpreter, this evaluates to `numV(7)`. Should it?

Translated into Pyret, this test corresponds to the following two definitions and expression:

```
fun f1(x): f2(4) end
fun f2(y): x + y end

f1(3)
```

What should this produce? `f1(3)` substitutes `x` with `3` in the body of `f1`, which then invokes `f2(4)`. But notably, in `f2`, the identifier `x` is *not bound*! Sure enough, Pyret will produce an error.

In fact, so will our substitution-based interpreter!

Why does the substitution process result in an error? It's because, when we replace the representation of `x` with the representation of `3` in the representation of `f1`, we do so in `f1` *only*. (Obviously: `x` is `f1`'s parameter; even if another function

This "the representation of" is getting a little annoying, isn't it? Therefore, we'll stop saying that, but do make sure you understand why we had to say it. It's an important bit of pedantry.

had a parameter named `x`, that's a *different* `x`.) Thus, when we get to evaluating the body of `f2`, its `x` hasn't been substituted, resulting in the error.

What went wrong when we switched to environments? Watch carefully: this is subtle. We can focus on applications, because only they affect the environment. When we substituted the formal for the value of the actual, we did so by *extending the current environment*. In terms of our example, we asked the interpreter to substitute not only `f2`'s substitution in `f2`'s body, but also the current ones (those for the caller, `f1`), and indeed all past ones as well. That is, the environment only grows; it never shrinks.

Because we agreed that environments are only an alternate implementation strategy for substitution—and in particular, that the language's meaning should not change—we have to alter the interpreter. Concretely, we should not ask it to carry around all past deferred substitution requests, but instead make it start afresh for every new function, just as the substitution-based interpreter does. This is an easy change:

<fof-env-interp-appC-rest-xtnd-2> ::=

```
  xtnd-env(bind(fd.arg, arg-val), mt-env)
```

Now we have truly reproduced the behavior of the substitution interpreter.

26.2.4 Scope

The broken environment interpreter above implements what is known as *dynamic scope*. This means the environment accumulates bindings as the program executes. As a result, whether an identifier is even bound depends on the *history of program execution*. We should regard this unambiguously as a flaw of programming language design. It adversely affects all tools that read and process programs: compilers, IDEs, and humans.

In contrast, substitution—and environments, done correctly—give us *lexical scope* or *static scope*. "Lexical" in this context means "as determined from the source program", while "static" in computer science means "without running the program", so these are appealing to the same intuition. When we examine an identifier, we want to know two things: (1) Is it bound? (2) If so, where? By "where" we mean: if there are multiple bindings for the same name, which one governs this identifier? Put differently, which one's substitution will give a value to this identifier? In general, these questions cannot be answered statically in a dynamically-scoped language: so your IDE, for instance, cannot overlay arrows to show you this information (the way an IDE like DrRacket does). Thus, even though the rules of scope become more complex as the space of names becomes richer (e.g., objects, threads, etc.), we should always strive to preserve the spirit of static scoping.

A different way to think about it is that in a dynamically-scoped language, the answer to these questions is the same for *all* identifiers, and it simply refers to the dynamic environment. In other words, it provides no useful information.

26.2.5 How Bad Is It?

You might look at our running example and wonder whether we're creating a tempest in a teapot. In return, you should consider two situations:

1. To understand the binding structure of your program, you may need to look at *the whole program*. No matter how much you've decomposed your program into small, understandable fragments, it doesn't matter if you have a free identifier anywhere.

2. Understanding the binding structure is not only a function of the *size* of the program but also of the complexity of its control flow. Imagine an interactive program with numerous callbacks; you'd have to track through every one of them, too, to know which binding governs an identifier.

Need a little more of a nudge? Let's replace the expression of our example program with this one:

```
if moon-visible():
  f1(10)
else:
  f2(10)
end
```

Suppose `moon-visible` is a function that evaluates to false on new-moon nights, and true at other times. Then, this program will evaluate to an answer except on new-moon nights, when it will fail with an unbound identifier error.

Exercise

What happens on cloudy nights?

26.2.6 The Top-Level Scope

Matters become more complex when we contemplate top-level definitions in many languages. For instance, some versions of Scheme (which is a paragon of lexical scoping) allow you to write this:

```
(define y 1)
(define (f x) (+ x y))
```

which seems to pretty clearly suggest where the `y` in the body of `f` will come from, except:

```
(define y 1)
(define (f x) (+ x y))
(define y 2)
```

is legal and `(f 10)` produces `12`. Wait, you might think, always take the last one! But consider:

```
(define y 1)
(define f (let ((z y)) (lambda (x) (+ x y z))))
(define y 2)
```

Here, `z` is bound to the first value of `y` whereas the inner `y` is bound to the second value. There is actually a valid explanation of this behavior in terms of lexical scope, but it can become convoluted, and perhaps a more sensible option is to prevent such redefinition. Pyret does precisely this, thereby offering the convenience of a top-level without its pain.

Most "scripting" languages exhibit similar problems. As a result, on the Web you will find enormous confusion about whether a certain language is statically- or dynamically-scoped, when in fact readers are comparing behavior inside functions (often static) against the top-level (usually dynamic). Beware!

26.2.7 Exposing the Environment

If we were building the implementation for others to use, it would be wise and a courtesy for the exported interpreter to take only an expression and list of function definitions, and invoke our defined `interp` with the empty environment. This both spares users an implementation detail, and avoids the use of an interpreter with an incorrect environment. In some contexts, however, it can be useful to expose the environment parameter. For instance, the environment can represent a set of pre-defined bindings: e.g., if the language wishes to provide `pi` automatically bound to `3.2` (in Indiana).

26.3 Functions Anywhere

The introduction to the Scheme programming language definition establishes this design principle:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

As design principles go, this one is hard to argue with. (Some restrictions, of course, have good reason to exist [section 26.5], but this principle forces us to argue for them, not admit them by default.) Let's now apply this to functions.

One of the things we stayed coy about when introducing functions [section 26.1] is exactly where functions go. We suggested we're following the model of an idealized programming environment, with definitions and their uses kept separate. But, inspired by the Scheme design principle, let's examine how *necessary* that is.

Why can't functions definitions be expressions? In our current arithmetic-centric language we face the uncomfortable question "What value does a function *definition* represent?", to which we don't really have a good answer. But a real programming language obviously computes more than numbers and Booleans, so we no longer need to confront the question in this form; indeed, the answer to the above can just as well be, "A function value". Let's see how that might work out.

What can we do with functions as values? Clearly, functions are a distinct kind of value than a number, so we cannot, for instance, add them. But there is one evident thing we can do: apply them to arguments! Thus, we can allow function values to appear in the function position of an application. The behavior would, naturally, be to apply the function. We are therefore proposing a language where the following would be a valid program (where I've used brackets so we can easily identify the function, and made up a syntax for it):

```
(+ 2 ([deffun f x (* x 3)] 4))
```

This would evaluate to `(+ 2 (* 4 3))`, or `14`. (Did you see that we just used substitution?)

26.3.1 Functions as Expressions and Values

Let's first define the core language to include function definitions:

<hof-named-dd> ::=

```
  data ExprC:
    | numC(n :: Number)
    | plusC(l :: ExprC, r :: ExprC)
    | multC(l :: ExprC, r :: ExprC)
    | trueC
    | falseC
    | ifC(c :: ExprC, t :: ExprC, e :: ExprC)
    | idC(s :: String)
    <hof-named-dd-fdC/1>
    <hof-named-dd-appC>
  end
```

For now, we'll simply copy function definitions into the expression language. We're free to change this if necessary as we go along, but for now it at least allows us to reuse our existing test cases.

<hof-named-dd-fdC/1> ::=

```
  | fdC(name :: String, arg :: String, body :: ExprC)
```

This enables us to now get rid of `FunDef`.

We also need to determine what an application looks like. What goes in the function position of an application? We want to allow an entire function definition, not just its name. Because we've lumped function definitions in with all other expressions, we need the annotation to be `ExprC`, but we can add a refinement ([REF]) to make clear it has to be a function definition:

<hof-named-dd-appC> ::=

```
  | appC(f :: ExprC%(is-fdC), a :: ExprC)
```

With this definition of application, we no longer have to look up functions by name, so the interpreter can get rid of the list of function definitions. If we need it we can restore it later, but for now let's just explore what happens with function definitions are written at the point of application: so-called *immediate* functions. Thus our interpreter looks like this:

<hof-named-interp/1> ::=

```
  fun interp(e :: ExprC, nv :: Environment):
    # removed return annotation of Value because fdC is not a Value!
    cases (ExprC) e:
      | numC(n) => numV(n)
      | plusC(l, r) => arith-binop(lam(x, y): x + y end, l, r, nv)
      | multC(l, r) => arith-binop(lam(x, y): x * y end, l, r, nv)
      | trueC => boolV(true)
      | falseC => boolV(false)
      | ifC(cnd, thn, els) =>
        ic = interp(cnd, nv)
        if is-boolV(ic):
          if ic.b:
            interp(thn, nv)
          else:
            interp(els, nv)
          end
        else:
          raise('not a boolean')
        end
      | idC(s) => lookup(s, nv)
      <hof-named-interp-fun/1>
      <hof-named-interp-app/1>
```

Do Now!

Observe that we've left out the return annotation on `interp`. Why do you think this is? Run some examples to figure it out.

We need to add a case to the interpreter for function definitions, and this is a good candidate:

<hof-named-interp-fun/1> ::=
```
  | fdC(_, _, _) => e
```
The interpreter now no longer returns just `Values`; now it also returns function definitions. We could update our definition of `Value` (and thus restore the annotation), but we'll soon find that we need to think this through a little more than we have.

When we need to evaluate an application, we can simply evaluate the function position to obtain a function definition, and the rest of the evaluation process can remain unchanged:

<hof-named-interp-app/1> ::=
```
  | appC(f, a) =>
    fun-val = interp(f, nv)
    arg-val = interp(a, nv)
    interp(fun-val.body, xtnd-env(bind(fun-val.arg, arg-val), mt-env))
```
With that, our former examples works just fine:

```
check:
  f1 = fdC("double", "x", plusC(idC("x"), idC("x")))
  f2 = fdC("quad", "x", appC(f1, appC(f1, idC("x"))))
  f3 = fdC("const5", "_", numC(5))
  f4 = fdC("f4", "x", s2p2d("(if x 1 0)"))
  fun i(e): interp(e, mt-env) end

  i(plusC(numC(5), appC(f2, numC(3)))) is numV(17)
  i(multC(appC(f3, numC(3)), numC(4))) is numV(20)
  i(plusC(numC(10), appC(f3, numC(10)))) is numV(10 + 5)
  i(plusC(numC(10), appC(f1, plusC(numC(1), numC(2)))))
    is numV(10 + 3 + 3)
  i(plusC(numC(10), appC(f2, plusC(numC(1), numC(2)))))
    is numV(10 + 3 + 3 + 3 + 3)
end
```

26.3.2 A Small Improvement

Do Now!

Is there any part of our interpreter definition that we never use?

Yes there is: the `name` field of a function definition is never used. This is because we no longer look up functions by name: we obtain their definition through evaluation. Therefore, a simpler definition suffices:

<hof-fun/2> ::=
```
  | fdC(arg :: String, body :: ExprC)
```

Do Now!

Do you see what else you need to change?

In addition to the test cases, you also need to alter the interpreter fragment that handles definitions:

<hof-interp-fun/2> ::=
```
  | fdC(_, _) => e
```
In other words, our functions are now *anonymous*.

26.3.3 Nesting Functions

The body of a function definition is an arbitrary expression. A function definition is itself an expression. That means a function definition can contain a...function definition. For instance:

```
inner-fun = fdC("x", plusC(idC("x"), idC("x")))
outer-fun = fdC("x", inner-fun)
```

which evaluates to

```
fdC("x", fdC("x", plusC(idC("x"), idC("x"))))
```

Applying this to `numC(4)` results in

```
fdC("x", plusC(idC("x"), idC("x")))
```

We might try to apply this to a number—which it should double—but we run afoul of the refinement annotation on the function position of an application, which envisioned *only* immediate functions, not expressions that can *evaluate* to functions. Therefore, we should remove this restriction:

...

Suppose, however, we use a slightly different function definition:

```
appC(fdC("x", fdC("y", plusC(idC("x"), idC("y")))), numC(4))
```

which evaluates to

```
fdC("y", plusC(idC("x"), idC("y")))
```

Now we have a clear problem, because `x` is no longer bound, even though it clearly was in an outer scope. Indeed, if we apply it to any value, we get an error because of the unbound identifier.

26.3.4 Nested Functions and Substitution

Consider the last two examples with a substitution-based interpreter instead. If we evaluate the application

```
appC(fdC("x", fdC("x", plusC(idC("x"), idC("x")))), numC(4))
```

using substitution, the inner binding *masks* the outer one, so no substitutions should take place, giving the same result:

```
fdC("x", plusC(idC("x"), idC("x")))
```

In the other example—

```
appC(fdC("x", fdC("y", plusC(idC("x"), idC("y")))), numC(4))
```

—however, substitution would replace the outer identifier, resulting in

```
fdC("y", plusC(numC(4), idC("y")))
```

So once again, if we take substitution as our definition of correctness, we see that our interpreter produces the wrong answer.

In other words, we're again failing to faithfully capture what substitution would have done. A function value needs to *remember the substitutions* that have already been applied to it. Because we're representing substitutions using an environment, a function value therefore needs to be bundled with an environment. This resulting data structure is called a *closure*.

"Save the environment! Create a closure today!"—Cormac Flanagan

26.3.5 Updating Values

In other words, a function can't just evaluate to its body: it must evaluate to a closure:

<hof-value> ::=

```
  data Value:
    | numV(n :: Number)
    | boolV(b :: Boolean)
    | closV(f :: ExprC%(is-fdC), e :: Environment)
  end
```

The refinement annotation reflects that we are expecting a very specific kind of expression—that representing a function definition—in a closure.

Look, we got our return value annotation back!

The interpreter now uses it. Most cases are unchanged from before:

<hof-interp> ::=
```
  fun interp(e :: ExprC, nv :: Environment) -> Value:
    cases (ExprC) e:
      <hof-named-interp/1>
      <hof-interp-fdC>
      <hof-interp-appC>
    end
  end
```
There are just two interesting cases: closure construction and closure use.

Do Now!

Write out these two cases.

“[Closures] are relegated to relative obscurity until Java makes them popular by not having them.”—James Iry

When evaluating a function, we have to create a closure that records the environment at the time of function creation:

<hof-interp-fdC> ::=
```
  | fdC(_, _) => closV(e, nv)
```
This leaves function applications. Now the function position could be any expression, so we have to evaluate it first. That produces a value that we expect is an instance of `closV`. From it we can therefore extract the function's body (`.f.body`) and argument name (`.f.arg`), and we evaluate the body in the environment *taken from the closure* (`clos.e`):

<hof-interp-appC> ::=
```
  | appC(f, a) =>
    clos = interp(f, nv)
    arg-val = interp(a, nv)
    interp(clos.f.body, xtnd-env(bind(clos.f.arg, arg-val), clos.e))
```

Exercise

Observe that the argument to `interp` is `clos.e` rather than `mt-env`. Write a program that illustrates the difference.

This now computes the same answer we would have gotten through substitution.

Do Now!

If we now switch back to using substitution, will we encounter any problems?

Yes, we will. We've defined substitution to replace program text in other program text. Strictly speaking we can no longer do this, because `Value` terms cannot be contained inside `ExprC` ones. That is, substitution is predicated on the assumption that the type of answers is a form of syntax. It is actually possible to carry through a study of programming under this assumption, but we won't take that path here.

26.3.6 Sugaring Over Anonymity

Now let's get back to the idea of naming functions, which has evident value for program understanding. Observe that we *do* have a way of naming things: by passing them to functions, where they acquire a local name (that of the formal parameter). Anywhere within that function's body, we can refer to that entity using the formal parameter name.

Therefore, we can name a function definion using another...function definition. For instance, the Pyret code

```
fun double(x): x + x end
double(10)
```

could first be rewritten as the equivalent

```
double = lam(x): x + x end
double(10)
```

which by substitution evaluates to `(lam(x): x + x end)(10)` or `20`.

Indeed, this pattern is a local naming mechanism, and virtually every language has it in some form or another. In languages like Lisp and ML variants, it is usually called `let`. For instance, in Racket:

Note that in different languages, `let` has different scope rules: in some cases it permits recursive definitions, and in others it doesn't.

```
(let ([double (lambda (x) (+ x x))])
  (double 10))
```

In Pyret, as in several other languages like Java, there is no explicitly named construct of this sort, but any definition block permits local definitions such as this:

```
fun something():
  double = lam(x): x + x end
  double(10)
end
```

Here's a more complex example, written in Racket to illustrate a point about scope:

```
(define (double x) (+ x x))
(define (quad x) (double (double x)))
(quad 10)
```

This could be rewritten as

```
(let ([double (lambda (x) (+ x x))])
  (let ([quad (lambda (x) (double (double x)))])
    (quad 10)))
```

which works just as we'd expect; but if we change the order, it no longer works—

```
(let ([quad (lambda (x) (double (double x)))])
  (let ([double (lambda (x) (+ x x))])
    (quad 10)))
```

—because `quad` can't "see" `double`. So we see that top-level binding is different from local binding: essentially, the top-level has "infinite scope". This is the source of both its power and problems.

26.4 Recursion and Non-Termination

Hopefully you can convince yourself that our pure expression languages—with only arithmetic and conditionals—could not create non-terminating programs. Why? Because its interpreter is *purely structural over a non-cyclic datatype*. In contrast, even our very first function interpreter is *generative*, which therefore opens up the possibility that it can have non-terminating computation.

Do Now!

Construct a non-terminating program for that interpreter.

And, indeed, it can. Here's a function definition:

```
il = fdC("inf-loop", "x", appC("inf-loop", numC(0)))
```

and we just need to get it started:

```
interp(appC("inf-loop", numC(0)), [list: il])
```

Exercise

Precisely identify the generative recursion that enables this.

Do Now!

Why does this work? Why is this an infinite loop?

What's happening here is actually somewhat subtle. The initial call to `interp` results in the interpreter finding a function and interpreting its body, which results in another call to `interp`: which finds the function and interprets its body, which results...and so on. If for some reason Pyret did not support recursion (which, historically, some languages did not!), then this would not work. Indeed, there is still something we are leaving to Pyret:

Do Now!

Does this program truly run for "ever" (meaning, as long as the computer is functioning properly), or does it run out of stack space?

Okay, that was easy. Now let's consider our most recent interpreter. What can it do?

Consider this simple infinite loop in Pyret:

```
fun loop-forever(): loop-forever() end
loop-forever()
```

Let's convert it to use an anonymous function:

```
loop-forever = lam(): loop-forever() end
loop-forever()
```

Seems fine, right? Use the `let` desugaring above:

```
(lam(loop-forever): loop-forever() end)(lam(): loop-forever() end)
```

But `loop-forever` isn't bound!

Therefore, Pyret's `fun` is clearly doing something more than just textual substitution: it is also "tying the loop" for recursive definitions through a hidden `rec` [section 21.3].

Do Now!

Can we try anything else that might succeed?

Actually, we can. Here it is. To make it more readable we'll first give the important intermediate term a name (and then see that the name isn't necessary):

```
little-omega = lam(x): x(x) end
```

Given this, we can then define:

```
omega = little-omega(little-omega)
```

Exercise

Why does this run forever? Consider using substitution to explain why.

Note that we could have written the whole thing without any names at all:

```
(lam(x): x(x) end)(lam(x): x(x) end)
```

As the names above suggest, the function is conventionally called ω (little omega in Greek), and the bigger term Ω (capital omega). To understand how we could have arrived at this magical term, see [REF].

26.5 Functions and Predictability

We began [section 26.1] with a language where at all application points, we knew exactly which function was going to be invoked (because we knew its name, and the name referred to one of a fixed global set). These are known as *first-order* functions. In contrast, we later moved to a language [section 26.3] with *first-class* functions: those that had the same status as any other value in the language.

This transition gave us a great deal of new flexiblity. For instance, we saw [section 26.3.6] that some seemingly necessary language features could instead be implemented just as syntactic sugar; indeed, with true first-class functions, we can define all of computation ([REF]). So what's not to like?

The subtle problem is that whenever we increase our *expressive* power, we correspondingly weaken our *predictive* power. In particular, when confronted with a particular function application in a program, the question is, can we tell precisely which function is going to be invoked at this point? With first-order functions, yes; with higher-order functions, this is undecidable. Having this predictive power has many important consequences: a compiler can choose to inline (almost) every function application; a programming environment can give substantial help about which function is being called at that point; a security analyzer can definitively rule out known bad functions, thereby reducing the number of useless alerts it generates. Of course, with higher-order functions, all these operations are still *sometimes* possible; but they are not *always* possible, and how possible they are depends on the structure of the program and the cleverness of tools.

Exercise

With higher-order functions, why is determining the precise function at an application undecidable?

Exercise

Why does the above reference to inlining say “almost”?

Chapter 27

Reasoning about Programs: A First Look at Types

One of the themes of this book is predictability [section 1.2]. One of our key tools in reasoning about program behavior before we run it is the static checking of *types*. For example, when we write `x :: Number`, we mean that `x` will always hold a `Number`, and that all parts of the program that depend on `x` can rely on this statement being enforced. As we will see, types are just one point in a spectrum of invariants we might wish to state, and static type checking—itself a diverse family of techniques—is also a point in a spectrum of methods we can use to enforce the invariants.

27.1 Types as a Static Discipline

In this chapter, we will focus especially on *static type checking*: that is, checking (declared) types before the program even executes. We will explore some of the design space of types and their trade-offs. Finally, though static typing is an especially powerful and important form of invariant enforcement, we will also examine some other techniques that we have available [REF].

This is an extremely rich and active subject. For further study, we strongly recommend reading Pierce's *Types and Programming Languages*.

Consider this Pyret program:

```
fun f(n :: Number) -> Number:
  n + 3
end

f("x")
```

We would like to receive a type error before the program begins execution. The

Pyret does not currently perform static type checking, but this will soon change.

same program (without the type annotations) can fail only at run-time:

```
fun f(n):
  n + 3
end

f("x")
```

Exercise

How would you test the assertions that one fails before the program executes while the other fails during execution?

Now consider the following Pyret program:

```
fun f n:
  n + 3
end
```

This too fails before program execution begins, with a parse error. Though we think of parsing as being somehow distinct from type-checking—usually because a type-checker assumes it has a parsed program to begin with—it can be useful to think of parsing as being simply the very simplest kind of type-checking: determining (typically) whether the program obeys a context-*free* syntax. Type-checking then asks whether it obeys a context-*sensitive* (or richer) syntax. In short, type-checking is a generalization of parsing, in that both are concerned with *syntactic* methods for enforcing disciplines on programs.

This particular, and very influential, phrasing is due to John Reynolds.

We will begin by introducing a traditional core language of types. Later, we will explore both extensions [REF] and variations [REF].

27.2 The Principle of Substitutability

The essence of *any* typing mechanism is usually the principle of *substitutability*: two types `A` and `B` "match" when values of one can be used in place of values of the other. Therefore, the design of a type system implicitly forces us to consider when such substitutions are safe (in the sense given by section 28.3).

Of course, the simplest notion of substitutability is simply identity: a type can only be substituted with itself, and nothing else. For instance, if the declared type of a function's parameter is `String`, then you can only call it with `String`-typed values, nothing else. This is known as *invariance*: the set of values that can be passed into a type cannot "vary" from the set expected by that type. This is so obvious that it might seem to hardly warrant a name! However, it is useful to name because it sets up a contrast with later type systems when we will have richer, non-trivial notions of substitutability (see section 32.6.1).

27.3 A Type(d) Language and Type Errors

Before we can define a type checker, we have to fix two things: the syntax of our *typed* core language and, hand-in-hand with that, the syntax of types themselves.

We'll begin with our language with functions-as-values [section 26.3]. To this language we have to add type annotations. Conventionally, we don't impose type annotations on constants or on primitive operations such as addition, because this would be unbearably tedious; instead, we impose them on the boundaries of functions or methods. Over the course of this study we will explore why this is a good locus for annotations.

Given this decision, our typed core language becomes:

```
data TyExprC:
  | numC(n :: Number)
  | plusC(l :: TyExprC, r :: TyExprC)
  | multC(l :: TyExprC, r :: TyExprC)
  | trueC
  | falseC
  | ifC(c :: TyExprC, t :: TyExprC, e :: TyExprC)
  | idC(s :: String)
  | appC(f :: TyExprC, a :: TyExprC)
  | fdC(arg :: String, at :: Type, rt :: Type, body :: TyExprC)
end
```

That is, every procedure is annotated with the type of argument it expects and type of argument it purports to produce.

Now we have to decide on a language of types. To do so, we follow the tradition that the types *abstract over the set of values*. In our language, we have three kinds of values. It follows that we should have three kinds of types: one each for numbers, Booleans, and functions.

What information does a number type need to record? In most languages, there are actually *many* numeric types, and indeed there may not even be a single one that represents "numbers". However, we have ignored these gradations between numbers [section 24.3], so it's sufficient for us to have just one. Having decided that, do we record additional information about *which* number? We could in principle, but that would mean for types to check, we would have to be able to decide whether two expressions compute the same number—a problem that reduces to the Halting Problem [REF].

In some specialized type systems, however, we do record some information about the number. These systems either have some means of approximation that lets them avoid the Halting Problem, or embrace it by not guaranteeing termination!

We treat Booleans just like numbers: we ignore which Boolean it is. Here, we perhaps have more value in being precise, because there are only two values we need to track, not an infinite number. That means in some cases, we even know which branch of a conditional we will take, and can examine only that branch (though that may miss a type-error lurking in the other branch: what should we do about that?). However, even the problem of knowing precisely which Boolean we have reduces to the Halting Problem [REF].

Exercise

Construct an argument for why determining which number or Boolean an arbitrary expression evaluates to is equivalent to solving the Halting Problem.

As for functions, we have more information: the type of expected argument, and the type of claimed result. We might as well record this information we have been given until and unless it has proven to not be useful. Combining these, we obtain the following abstract language of types:

```
data Type:
  | numT
  | boolT
  | funT(a :: Type, r :: Type)
end
```

Now that we've fixed both the term and type structure of the language, let's make sure we agree on what constitute type errors in our language (and, by fiat, everything not a type error must pass the type checker). There are three obvious forms of type errors:

- One or both arguments of `+` is not a number, i.e., does not have type `numT`.
- One or both arguments of `*` is not a number.
- The expression in the function position of an application is not a function, i.e., does not have type `funT`.

Do Now!

Any more?

We're actually missing one:

- The expression in the function position of an application is a function but the type of the actual argument does not match the type of the formal argument expected by the function.

Do Now!

Any more?

What about:

- The expression in the function position of an application is a function but its return type does not match the type expected by the expression that invokes the function?

And we're still not done!

Instead of this kind of ad hoc enumeration, what we really ought to do is systematically go over each of the syntactic forms of our language and ask how each of them can produce a type error. That indicates:

```
| numC(n :: Number)
| plusC(l :: TyExprC, r :: TyExprC)
| multC(l :: TyExprC, r :: TyExprC)
```

A number on its own can never be a type error. For addition and multiplication, both branches must have numeric type.

```
| trueC
| falseC
| ifC(c :: TyExprC, t :: TyExprC, e :: TyExprC)
```

Just as with numbers, Boolean constants on their own cannot be a type error. In a conditional, however, we require:

- The conditional expression must have type Boolean.
- Both branches must have the same type (whatever it may be).

Implicit is the idea that we can easily determine when two types are the "same". We'll return to this in section 32.6.1.

And finally:

```
| idC(s :: String)
| appC(f :: TyExprC, a :: TyExprC)
| fdC(arg :: String, at :: Type, rt :: Type, body :: TyExprC)
```

An identifier on its own is never type-erroneous. Applications expect:

- The function position (`f`) must have a function type (`funT`).

- The type of the actual argument expression (`a`) must match the argument type (`.arg`) of the function position.

And finally, a function definition expects:

- The type of the body—assuming the formal argument (`arg`) has been given a value of the declared type (`at`)—matches the type declared (`rt`) as the return type.

27.3.1 Assume-Guarantee Reasoning

The last few cases we just saw had a very interesting structure. Did you spot it?

The rules for function *definition* and *declaration* complement each other perfectly. Let's illustrate this with a program written in Pyret syntax:

```
fun f(x :: String) -> Number:
  if x == "pi":
    3.14
  else:
    2.78
  end
end

2 + f("pi")
```

When type-checking the definition of `f`, we *assume* that if and when `f` is eventually applied, it will be applied to a value of `String` type. We *do* assume this because the annotation on `x` is `String`. We *can* assume this because when checking the application, we will first look up the type of `f`, observe that it expects a `String`-typed value, and confirm that the actual argument indeed matches this type. That is, the type-checker's treatment of application *guarantees* that this assumption is safe.

Similarly, when type-checking the application, having looked up the type of `f`, we *assume* that it will indeed return a value of type `Number`. We *can* assume this because that is the return type annotation of `f`. We *do* assume it because the type-checker will ensure that the body of `f`—assuming the type of `x`—will indeed return a `Number`. That is, once again, the type-checker's treatment of function definitions *guarantees* that the assumption at function applications is safe.

In short, the treatment of function definition and application are complementary. They are joined together by a method called *assume-guarantee reasoning*, whereby each side's assumptions are guaranteed by the other side, and the two stitch together perfectly to give us the desired safe execution (which we elaborate on later: section 28.3).

27.4 A Type Checker for Expressions and Functions

27.4.1 A Pure Checker

Since the job of a type-checker is to pass judgment on programs—in particular, to indicate whether a program passes or fails type-checking—a natural type for a type-checker would be:

```
tc :: TyExprC -> Boolean
```

However, because we know expressions contain identifiers, it soon becomes clear that we will want a *type environment*, which maps names to types, analogous to the value environment we have seen so far.

Exercise

Define the types and functions associated with type environments.

Thus, we might begin our program as follows:

<hof-tc-bool> ::=

```
  fun tc(e :: TyExprC, tnv :: TyEnv) -> Boolean:
    cases (TyExprC) e:
      <hof-tc-bool-numC>
      <hof-tc-bool-idC>
      <hof-tc-bool-appC>
    end
  end
```

As the abbreviated set of cases above suggests, this approach will not work out. We'll soon see why.

Let's begin with the easy case: numbers. Does a number type-check? Well, on its own, of course it does; it may be that the surrounding context is not expecting a number, but that error would be signaled elsewhere. Thus:

<hof-tc-bool-numC> ::=

```
  | numC(_) => true
```

(Notice that we're expressly ignoring *which* number it is.)

Now let's handle identifiers. Is an identifier well-typed? Again, on its own it would appear to be, provided it is actually a bound identifier; it may not be what the context desires, but hopefully that too would be handled elsewhere. Thus we might write

<hof-tc-bool-idC> ::=

```
  | idC(s) => ty-lookup(s, tnv)
```

where `ty-lookup` returns `true` if the identifier is bound, and `false` otherwise.

This should make you a little uncomfortable: we seem to be throwing away valuable information about the type of the identifier. Of course, types do throw away information (e.g., which specific number an expression computes). However, the kind of information we're throwing away here is much more significant: it's not about a specific value within a type, but the type itself. Nevertheless, let's push on.

It might also bother you that, by only returning a Boolean, we have no means to express *what* type error occurred. But you might assuage yourself by saying that's only because we have too weak a return type.

Now we tackle applications. We should type-check both the function part, to make sure it's a function, then ensure that the actual argument's type is consistent with what the function declares to be the type of its formal argument. How does the code look?

<hof-tc-bool-appC> ::=

```
  | appC(f, a) =>
    f-t = tc(f, tnv)
    a-t = tc(a, tnv)
    ...
```

The two recursive calls to `tc` can only tell us whether the function and argument expressions type-check or not. Critically, they cannot tell us whether the argument expression's type (what is it?) matches that of the function's expected argument type (what is *it*?). Though we might be able to fudge this in the case of simple expressions, for complex ones we cannot just examine the expression; furthermore, this violates our principle of wanting to avoid probing deep into expressions. Put differently, we'd like to have written

```
| appC(f, a) =>
  f-t = tc(f, tnv)
  a-t = tc(a, tnv)
  if is-funT(f-t):
    if a-t == f-t.arg:
```

but `f-t` is a Boolean and hence can never pass `is-funT`; similarly, comparing `a-t` with `f-t.arg` is meaningless because both are Booleans (representing whether or not the corresponding sub-expressions type-checked), not the actual types of those expressions.

In other words, what we need is something that will *calculate* the type of an expression, no matter how complex it is. Of course, such a procedure could only succeed if the expression is well-typed; otherwise it would not be able to provide a coherent answer. In other words, *a type "calculator" has type "checking" as a special case*!

Do Now!

That was subtle. Read it again.

We should therefore *strengthen the inductive invariant* on `tc`: that it not only tells us whether an expression is typed, but also what its type is. Indeed, by giving any type at all it confirms that the expression types, and otherwise it signals an error.

27.4.2 A Calculator and Checker

Let's now define this richer notion of a type "checker".
<hof-tc> ::=

```
  fun tc(e :: TyExprC, tnv :: TyEnv) -> Type:
    cases (TyExprC) e:
      <hof-tc-numC>
      <hof-tc-plusC>
      <hof-tc-multC>
      <hof-tc-bools>
      <hof-tc-idC>
      <hof-tc-fdC>
      <hof-tc-appC>
    end
  end
```

Now let's fill in the pieces. Numbers are easy: they have the numeric type.
<hof-tc-numC> ::=

```
  | numC(_) => numT
```

Similarly, identifiers have whatever type the environment says they do (and if they aren't bound, looking them up signals an error).
<hof-tc-idC> ::=

```
  | idC(s) => ty-lookup(s, tnv)
```

Observe, so far, the similarity to and difference from interpreting: in the identifier case we did essentially the same thing (except we returned a type rather than an actual value), whereas in the numeric case we returned the abstract "number" (`numT`) rather than indicate which specific number it was.

Let's now examine addition. We must make sure both sub-expressions have numeric type; only if they do will the overall expression evaluate to a number itself. It will be useful to employ a helper function:
<hof-tc-plusC> ::=

```
  | plusC(l, r) => tc-arith-binop(l, r, tnv)
```

where:

```
fun tc-arith-binop(l :: TyExprC, r :: TyExprC, tnv :: TyEnv) -> Type:
  if (tc(l, tnv) == numT) and (tc(r, tnv) == numT):
    numT
  else:
    raise('type error in arithmetic')
  end
end
```

It's worth not glossing over multiplication:
<hof-tc-multC> ::=

```
  | multC(l, r) => tc-arith-binop(l, r, tnv)
```

Do Now!

Did you see what's different?

That's right: *nothing*! That's because, from the perspective of type-checking (in this type language), there is no difference between addition and multiplication, or indeed between *any* two operations that consume two numbers and return one. Because we are ignoring the actual numbers, we don't even need to bother passing `tc-arith-binop` a function that reflects what to do with the pair of numbers.

Observe another difference between interpreting and type-checking. Both care that the arguments be numbers. The interpreter then returns a precise sum or product, but the type-checker is indifferent to the differences between them: therefore the expression that computes what it returns (`numT`) is a constant, and the same constant in both cases.

Next, let's handle Boolean values and conditionals. We're simply going to transcribe into code what we earlier agreed to do:
<hof-tc-bools> ::=

```
  | trueC => boolT
  | falseC => boolT
  | ifC(cnd, thn, els) =>
    cnd-t = tc(cnd, tnv)
    if cnd-t == boolT:
      thn-t = tc(thn, tnv)
      els-t = tc(els, tnv)
      if thn-t == els-t:
        thn-t
      else:
        raise("conditional branches don't match")
      end
    else:
      raise("conditional isn't Boolean")
    end
```

However, recall our discussion of section 25.1, all of which have consequences for type-checking. Here we are applying the decisions we made there.

Exercise

Consider each of the three earlier decisions. Change each one, and explain the consequences it has for the type-checker.

Finally, the two hard cases: application and functions. We've already discussed what application must do: compute the value of the function and argument expressions; ensure the function expression has function type; and check that the argument expression is of compatible type. If all this holds up, then the type of the overall application is whatever type the function body would return (because the value that eventually returns at run-time is the result of evaluating the function's body).

Note that this subtly depends on evaluation and type-checking being in harmony. We discuss this under section 28.3.

<hof-tc-appC> ::=

```
  | appC(f, a) =>
    f-t = tc(f, tnv)
    a-t = tc(a, tnv)
    if is-funT(f-t):
      if a-t == f-t.arg:
        f-t.ret
      else:
        raise("argument type doesn't match declared type")
      end
    else:
      raise("not a function in application position")
    end
```

That leaves function definitions. The function has a formal parameter; unless this is bound in the type environment, any use of that parameter in body would result in a type error. Thus we have to extend the type environment with the formal name bound to its type, and in that extended environment type-check the body. Whatever value this computes must be the same as the declared type of the body. If that is so, then the function itself has a function type from the type of the argument to the type of the body.

<hof-tc-fdC> ::=

```
| fdC(a, at, rt, b) =>
  bt = tc(b, xtend-t-env(tbind(a, at), tnv))
  if bt == rt:
    funT(at, rt)
  else:
    raise("body type doesn't match declared type")
  end
```

27.4.3 Type-Checking Versus Interpretation

Do Now!

When confronted with a first-class function, our interpreter created a closure. However, we don't seem to have any notion of a "closure" in our type-checker, even though we're using an (type) environment. Why not? In particular, recall that the absence of closures resulted in violation of static scope. Is that happening here? Write some tests to investigate.

Observe a curious difference between the interpreter and type-checker. In the interpreter, application was responsible for evaluating the argument expression, extending the environment, and evaluating the body. Here, the application case does check the argument expression, but leaves the environment alone, and simply returns the type of the body *without traversing it*. Instead, the body is actually traversed by the checker when checking a function *definition*, so this is the point at which the environment actually extends.

Exercise

Why is the time of traversal different between interpretation and type-checking?

The consequences of this are worth understanding.

- Consider the Pyret function

  ```
  p =
    lam(x :: Number) -> (Number -> Number):
      lam(y :: Number) -> Number:
        x + y
      end
    end
  ```

  When we simply define `p`, the interpreter does not traverse the interior of these expressions, in particular the `x + y`. Instead, these are suspended waiting for later use (a feature we actually exploit [REF laziness]). Furthermore, when we apply `p` to some argument, this evaluates the outer function, resulting in a closure (that closes over the binding of `x`).

Now instead consider the type-checker. As soon as we are given this definition, it traverses the entire expression, including the innermost sub-expression. Because it knows everything it needs to know about `x` and `y`—their types—it can immediately type-check the entire expression. This is why it doesn't not require to create a closure: there is nothing to be put off until application time (indeed, we don't *want* to put type-checking off until execution).

Another way to think about it is that it behaves like substitution does—and substitution did not need closures to provide static scoping, either—but even more eagerly: it can perform substitution with just the program text without any values at all, because it is substituting types, which are already given. The fact that we use a type environment makes this harder to see, because we may have come to associate environments with closures. However, what matters is when the necessary value is available. Put differently, we used an environment primarily out of convention: here, we could have used (type) substitution just as well.

Exercise

Write examples to study this. Consider converting the above example as a starting point. Also convert your examples from earlier.

- Consider the following expression:

```
lam(f :: (Number -> String), n :: Number) -> String:
  f(n)
end
```

When evaluating the inner `f(n)`, the interpreter has access to actual values for `f` and `n`. In contrast, when type-checking it, it does not know which function will be passed in as `f`. How, then, can it type-check the use?

The answer is that the *annotation tells the type-checker everything it needs to know*. The annotation says that `f` must accept numbers; since `n` is annotated to be a number, the application works. It also says that `f` will return strings; because that is what the overall function returns, this also passes.

In other words, the annotation `(Number -> String)` represents not one but an infinite family of *all* functions of that type, without committing to any one of them. The type checker then checks that *any* such function will work in this setting. Once it has done its job, it doesn't matter which function we actually pass in, provided it has this type. Checking that is, of course, the heart of section 27.3.1.

27.5 Type-Checking, Testing, and Coverage

A type-checker can be thought of as a very particular kind of testing framework:

- Instead of using concrete values, it uses only types. Therefore, it cannot check fine gradations inside values.

- In return, it works statically: that is, it's like running a lightweight testing procedure before ever running the program. (We should not underestimate the value of this: programs that depend on interactive or other external input, on specialized hardware, on timing, and so on, can be quite difficult to test. For such programs, especially, obtaining a lightweight form of testing that does not require being able to run it at all is invaluable.)

- Testing only covers the parts of a program that are exercised by test cases. In contrast, the type-checker exercises the whole program. Therefore, it can catch lurking errors. Of course, it also means that the entire program has to be type-conformant: you can't have some parts (e.g., conditional branches) that are not yet conformant, the way they can fail to work correctly but can be ignored by tests that don't exercise them.

- Finally, types provide another very important property: *quantification*. Recall our earlier example: the type checker has established something about an infinite number of functions!

This last point gets to the heart of the tradeoff between types and testing: types are "broad" while tests are "deep". That is, because tests deal with very specific values and their actual evaluation, they can ask arbitrarily deep questions but about that one situation only. Types, in contast, lacking the specificity provided by both values and evaluation, cannot ask deep questions; they compensate by being able to talk about all possible values of some shape, providing their breadth. As this discussion illustrates, neither attribute dominates the other: a good software practice will use a judicious combination of both.

27.6 Recursion in Code

Now that we've obtained a basic programming language, let's add recursion to it. We saw earlier [section 26.4] that this could be done quite easily. It'll prove to be a more complex story here.

27.6.1 A First Attempt at Typing Recursion

Let's now try to express a simple recursive function. We've already seen how to write infinite loops for first-order functions. Annotating them introduces no complications.

Exercise

Confirm that adding types to recursive and non-terminating first-order functions causes no additional problems.

Now let's move on to higher-order functions. We've already seen that this results in an infinite loop:

```
(fun(x): x(x) end)(fun(x):  x(x) end)
```

Now that we have a typed language, we have to annotate it. (Conventionally, we call this term Ω.)

Recall that this program is formed by applying ω to itself. Of course, it is not a given that identical terms must have precisely the same type, because it depends on the context of use. However, the specific structure of ω means that it is the *same* term that ends up in both contexts—as function and argument—so the types of these had better be the same. In other words, typing one instance of ω suffices to type them both.

Therefore, let's try to type ω; let's call this type `T`. It's clearly a function type, and the function takes one argument, so it must be of the form `A -> B`. Now what is that argument? It's ω itself. That is, the type of the value going into `A` is itself `T`. Thus, the type of ω is `T`, which is `A -> B`, which is the same as `T -> B`. This expands into `(A -> B) -> B`, which is the same as `(T -> B) -> B`. Therefore, this further expands to `((A -> B) -> B) -> B`, and so on. In other words, this type cannot be written as any finite string!

Do Now!

Did you notice the subtle but important leap we just made?

Do Now!

We have just argued that we can't type ω. But why does it follow that we can't type Ω?

27.6.2 Program Termination

Because type-checking follows by recurring on sub-terms, to type Ω, we have to be able to type ω and then combine its type to obtain one for Ω. But, as we've seen, typing ω seems to run into serious problems. From that, however, we jumped to the conclusion that ω's type cannot be written as any finite string, for which we've given only an intuition, not a proof. In fact, something even stranger is true: in the type system we've defined so far, *we cannot type* Ω *at all*!

This is a strong statement, but it follows from something even stronger. The *typed* language we have so far has a property called *strong normalization*: *every* expression that has a type will terminate computation after a finite number of steps. In other words, this special (and peculiar) infinite loop program isn't the only one we can't type; we can't type *any* infinite loop (or even potential infinite loop). A rough intuition that might help is that any type—which must be a finite string—can have only a finite number of `->`'s in it, and each application discharges one, so we can perform only a finite number of applications.

Exercise

Why is this not true when we have named first-order functions?

If our language permitted only straight-line programs, this would be unsurprising. However, we have conditionals and even functions being passed around as values, and with those we can encode almost every program we're written so far. Yet, we still get this guarantee! That makes this a somewhat astonishing result.

Exercise

Try to encode lists using functions in the untyped and then in the typed language (see [REF] if you aren't sure how). What do you see? And what does that tell you about the impact of this type system on the encoding?

This result also says something deeper. It shows that, contrary to what you may believe—that a type system only prevents a few buggy programs from running—a type system can *change the semantics* of a language. Whereas previously we could write an infinite loop in just one to two lines, now we cannot write one at all. It also shows that the type system can establish invariants not just about a particular program, but *about the language itself*. If we want to absolutely ensure that a program will terminate, we simply need to write it in this language and pass the type checker, and the guarantee is ours!

What possible use is a language in which all programs terminate? For general-purpose programming, none, of course. But in many specialized domains, it's a tremendously useful guarantee to have. Here are several examples of domains that could benefit from it:

- A complex scheduling algorithm (the guarantee would ensure that the scheduler completes and that the tasks being scheduled will actually run).
- A packet-filter in a router. (Network elements that go into infinite loops put a crimp on utility.)
- A compiler. (The program it generates may or may not terminate, but it ought to at least finish generating the program.)
- A device initializer. (Modern electronics—such as a smartphones and photocopiers—have complex initialization routines. These have to finish so that the device can actually be put to use.)
- The callbacks in JavaScript. (Because the language is single-threaded, not relinquishing control means the event loop starves. When this happens in a Web page, the browser usually intervenes after a while and asks whether to kill the page—because otherwise the rest of the page (or even browser) becomes unresponsive.)
- A configuration system, such as a build system or a linker.

In Standard ML, the linker uses essentially this language for module linking specifications.

Notice also an important difference between types and tests [section 27.5]: you can't *test* for termination!

27.6.3 Typing Recursion

What this says is, if we want potentially unbounded recursion, we must make it an explicit part of the typed language. To illustrate this, we will add a simple `rec` construct that recursively binds an identifier to a function. Thus, in the surface syntax, one might write

For convenience, we have also added an `if0` construct that compares the test expression's value with `0`.

```
(rec (S num (n num)
        (if0 n
             0
             (n + (S (n + -1)))))
  (S 10))
```

for a summation function, where `S` is the name of the function, `n` its argument, the first `num` the type of `n` and the second `num` the type returned by the function. The expression `(S 10)` represents the use of this function to sum the numbers from `10` until `0`.

How do we type such an expression? Clearly, we must have `n` bound in the body of the function as we type it (but not, of course, in the use of the function, due to static scope); this much we know from typing functions. But what about `S`? Obviously it must be bound in the type environment when checking the use (`S 10)`), and its type must be `num -> num`. But it must *also* be bound, to the same type, when checking the body of the function. (Observe, too, that the type returned by the body must match its declared return type.)

Now we can see how to break the shackles of the finiteness of the type. It is certainly true that we can write only a finite number of `->`'s in types in the program source. However, this rule for typing recursion *duplicates* the `->` in the body that refers to itself, thereby ensuring that there is an inexhaustible supply of applications.

It's our infinite quiver of arrows.

The code to implement this rule would be as follows. Assuming `f` is bound to the function's name, `v` its parameter's name, `at` is the function's argument type and `rt` is its return type, `b` is the function's body, and `c` is the function's use:

<tc-recC> ::=
```
  | recC(f, v, at, rt, b, c) =>
    extended-env = xtend-t-env(tbind(f, funT(at, rt)), tnv)
    if not(rt == tc(b, xtend-t-env(tbind(v, at), extended-env))):
      raise("rec: function return type not correct")
    else:
      tc(c, extended-env);
```

27.7 Recursion in Data

We have seen how to type recursive programs, but this doesn't yet enable us to create recursive data. We already have one kind of recursive datum—the function type—but this is built-in. We haven't yet seen how developers can create their own recursive datatypes.

27.7.1 Recursive Datatype Definitions

When we speak of allowing programmers to create recursive data, we are actually talking about three different facilities at once:

- Creating a new type.

- Letting instances of the new type have one or more fields.
- Letting some of these fields refer to instances of the same type.

In fact, once we allow the third, we must allow one more:

- Allowing non-recursive base-cases for the type.

This confluence of design criteria leads to what is commonly called an *algebraic datatype*. For instance, consider the following definition of a binary tree of numbers:

Later [chapter 29], we will discuss how types can be parameterized.

```
data BinTree:
  | leaf
  | node (value :: Number,
          left :: BinTree,
          right :: BinTree)
end
```

Observe that without a name for the new datatype, BinTree, we would not have been able to refer back ot it in node. Similarly, without the ability to have more than one kind of BinTree, we would not have been able to define leaf, and thus wouldn't have been able to terminate the recursion. Finally, of course, we need multiple fields (as in node) to construct useful and interesting data. In other words, all three mechanisms are packaged together because they are most useful in conjunction. (However, some langauges do permit the definition of stand-alone structures. We will return to the impact of this design decision on the type system later [REF].)

This style of data definition is sometimes also known as a *sum of products*. At the outer level, the datatype offers a set of choices (a value can be a leaf *or* a node). This corresponds to disjunction ("or"), which is sometimes written as a sum (the truth table is suggestive). Inside each sum is a set of fields, *all* of which must be present. These correspond to a conjunction ("and"), which is sometimes written as a product (ditto).

That covers the notation, but we have not explained where this new type, BinTree, comes from. It is obviously impractical to pretend that it is baked into our type-checker, because we can't keep changing it for each new recursive type definition—it would be like modifying our interpreter each time the program contains a recursive function! Instead, we need to find a way to make such definitions intrinsic to the type language.

27.7.2 Introduced Types

Now, what impact does a datatype definition have? First, it introduces a new type; then it uses this to define several constructors, predicates, and selectors. For instance, in the above example, it first introduces `BinTree`, then uses it to ascribe the following types:

```
leaf :: BinTree  # a constant, so no arrow
node :: Number, BinTree, BinTree -> BinTree
is-leaf :: BinTree -> Bool
is-node :: BinTree -> Bool
.value :: BinTree%(is-node) -> Number
.left :: BinTree%(is-node) -> BTnum
.right :: BinTree%(is-node) -> BTnum
```

Do Now!

In what two ways are the last three entries above fictitious?

Observe a few salient facts:

- Both the constructors create instances of `BinTree`, not something more refined. We will discuss this design tradeoff later [REF].
- Both predicates consume values of type `BinTree`, not "any" value. This is because the type system can already tell us what type a value is. Thus, we only need to distinguish between the variants of that one type.
- The selectors really only work on instances of the relevant variant—e.g., `.value` can work only on instances of `node`, not on instances of `leaf`—but we don't have a way to express this in the static type system for lack of a suitable static type. Thus, applying these can only result in a dynamic error, not a static one caught by the type system.

There is more to say about recursive types, which we will return to shortly [REF].

27.7.3 Selectors

`.value`, `.left`, and `.right` are *selectors*: they select parts of the record by name. But here are the two ways in which they are fictitious. First, syntactically: in most languages with "dotted field access", there is no such stand-alone operator as `.value`: e.g., you cannot write `.value(...)`. But even setting aside this syntactic matter (which could be addressed by arguing that writing `v.value` is just an obscure syntax for applying this operator) the more interesting subtlety is the semantic one.

Above, we have given a very particular type to `.value`. Suppose, however, that this datatype was also defined in the same program:

```
data Payment:
  | cash(value :: Number)
  | card(number :: Number, value :: Number)
end
```

This too appears to define a `.value` operator with the type:

```
.value :: Payment(is-cash) -> Number
.value :: Payment(is-card) -> Number
```

or equivalently,

```
.value :: Payment -> Number
```

Will the real `.value` please stand up? How many `.value` operations are there? Indeed, it would appear that this "operator" freely cross-cuts every datatype definition, and even every module boundary!

These issues are not really specific to types: the cross-cutting nature of field access is independent of it. However, ascribing types forces us to confront these issues, because we cannot ignore the difficulty of typing the operation.

To put this in perspective, consider two other very different styles of handling selectors:

- A characteristic of scripting languages is that objects are merely hash tables, and all field access is turned into a hash-table reference on the string representing the field-name. Hence, `o.f` is just syntactic sugar for looking up the value indexed by `"f"` in the dictionary associated with `o`.

- In Racket, the structure definitions such as

  ```
  (struct cash (value))
  (struct card (number value))
  ```

  generate distinct selectors: in this case, `cash-value` and `card-value`, respectively. Now there is no longer any potential for confusion, because they have different names that can each have distinct types.

Compiling between these languages then highlights these distinctions. Compiling from Pyret or Java to JavaScript is easy, because *all* field dereferences turn into dictionary lookups. Compiling from (untyped) Pyret to Racket is especially easy because the languages are so similar—until we get to dotted access. Then, assuming we wish to compile Pyret data definitions to Racket's corresponding structure definitions, the compiler would have to traverse the Pyret program to gather up all fields with a common name, and turn them into a discriminating selector: for instance, `v.value` might compile to Racket's `(->value v)`, where `->value` is defined as (given the above two data definitions):

```
(define (->value v)
  (cond
    [(node? v) (node-value v)]
    [(cash? v) (cash-value v)]
    [(card? v) (card-value v)]))
```

In contrast, going in the other direction is easy: `(node-value v)` would check that `v` is indeed a `node`, and then access `v.value`.

27.7.4 Pattern-Matching and Desugaring

Once we have understood the names introduced by datatype definitions, and the nature of selectors, the only thing left is to provide an account of pattern-matching. For instance, we can write the expression

```
cases (BinTree) t:
  | leaf => e1
  | node(v, l, r) => e2
end
```

This simply expands into uses of the above predicates, and binding the pieces:

```
if is-leaf(t):
  e1
else if is-node(t):
  v = t.value
  l = t.left
  r = t.right
  e2
end
```

In short, this can be done by desugaring, so pattern-matching does not need to be in the core language. This, in turn, means that one language can have many different pattern-matching mechanisms.

Except, that's not quite so easy. Somehow, the desugaring that generates the code above in terms of `if` needs to know that the three positional selectors for a `node` are `value`, `left`, and `right`, respectively. This information is explicit in the type definition but only implicitly present in the use of the pattern-matcher (that, indeed, being the point). Somehow this information must be communicated from definition to use. Thus, the desugarer needs something akin to the type environment to accomplish its task.

Observe, furthermore, that expressions such as `e1` and `e2` cannot be type-checked—indeed, cannot even be reliable identified as *expressions*—until desugaring expands the use of `cases`. Thus, desugaring depends on the type environment, while type-checking depends on the result of desugaring. In other words, the two are symbiotic and need to happen, not quite in "parallel", but rather in lock-step. What this implies is that building desugaring for a typed language *when the syntactic sugar makes assumptions about types* is more intricate than doing so for an untyped language.

Chapter 28

Safety and Soundness

Now that we've had a first look at a type system, we're ready to talk about the way in which types offer some notion of predictability, a notion called *type soundness*. Intertwined with this are terms you have probably heard like *safety* (or *type safety*) as well as others such as *strong typing* (and conversely *weak typing*) and *memory safety*. We should understand all of them.

A type *system* usually has three components: a language of types, a set of type rules, and an algorithm that enforces these rules. By presenting types via a checking function we have blurred the distinction between the second and third of these, but they should still be thought of as intellectually distinct: the former provides a *declarative* description while the latter an *executable* one. The distinction becomes relevant when implementing subtyping [REF].

28.1 Safety

Many operations in a language are *partial*: given some domain over which they are defined, they accept some but not all elements of the domain. For instance, while addition—defined over numbers—is usually total, division is usually partial, because division by zero is considered an error. In just about every language, the function application operator is limited to applying only function values, i.e., an application like `3(5)`—the number `3` applied to one argument, `5`—is illegal.

Of course, exactly whether an operator is partial or total is a matter of how we define the domain. For instance, if we define the domain of division's second argument as non-zero numbers, it becomes total; whereas if we consider the domain of addition's arguments as all values, it becomes partial. Also, some operations are treated quite differently across different languages. For instance, the square-root function, when applied to `-1`, can variously

- halt with an error,
- return a special value called "not-a-number" (`NaN`), or
- return an imaginary number (e.g., `0+1i` in Scheme).

Furthermore, it might be surprising to consider that an operation can work over *all* values, but that is precisely what parametric polymorphism enables [chapter 29].

What matters, then, is whether an operation precludes *any* values at all or not. If it does, then we can ask whether the language prevents it from being used with any precluded values. If the language does prevent it, then we call the language *safe*. If it does not, we call it *unsafe*. Of course, a language may be safe for some operations and unsafe for others; usually we apply the term “safe” to a language as a whole if *all* of its operations are safe.

A safe language offers a very important guarantee to programmers: that no operation will be performed on meaningless data. Sticking with numeric addition, in an unsafe language we might be permitted to add a number to a string, which will produce some value dependent on the precise representation of strings and *might change* if the representation of strings changes. (For instance, the string might be zero-terminated or might record its length, which alters what the first word of the string will be.) We might even be able to add a string to a function, a result that is certainly nonsensical (what meaningful number does the first word of the machine representation of a function represent?). Therefore, though safety does not at all ensure that computations will be correct, at least it ensures they will be *meaningful*: no nonsensical operations will have been performed.

Observe that we have not imposed any requirement whatsoever on *how* a language achieves safety. Usually, safety can be achieved dynamically through *run-time checks*: for instance, addition would check that its arguments are numeric, division would also ensure its second argument is not zero, and so on. In addition, a static type system can also ensure safety. Because this is a common source of confusion, we should be clear: safety *only* means that operations are not applied to meaningless values. It does not fix any particular implementation strategy for ensuring the property.

We will return to the issue of implementations below [section 28.4]. But first, we have some important foundational material to address.

28.2 “Untyped” Languages

It is common in popular writing to use the phrase “untyped” language. This is a source of considerable confusion, so we should tease apart its meanings. There are two different things it might mean, and these meanings are non-overlapping:

1. A language with no types at all. Of course all data have some representation that gives them meaning, but it means there is only one type in the language, and all data belong to that type. Furthermore, this datatype has no variants, because that would introduce type-based discrimination. For instance, all

data might be a byte, or a number, or a string, or some other single, distinctive value. Typically, no operation can fail to take a particular kind of value, because that might imply some kind of type distinction, which by definition can't exist. Note that this is a *semantic* notion of untypedness.

2. A language with a partitioning of its run-time values—e.g., numbers are distinct from functions—but without *static annotations or checking*. Note that this is a *syntactic* notion of untypedness.

Virtually no contemporary language other than machine code—where the single type is usually a "word"—exists. In contrast, there are many languages of the latter kind (e.g., Python and Racket).

Because the two meanings are mutually contradictory, it would be useful to have two different names for these. Some people use the terms *latently typed* or *dynamically typed* for the latter category, to tell these apart.

Following modern convention, we will use the latter term, while recognizing that some others consider the term *typed* to only apply to languages that have static disciplines, so the phrase "dynamically typed" is regarded as an oxymoron. Note that our preceding discussion gives us a way out of this linguistic mess. A dynamically typed language that does *not* check types at run-time is not very interesting (the "types" may as well not exist). In contrast, one that does check at run-time already has a perfectly good name: *safe* [section 28.1]. Therefore, it makes more sense to use the name *dynamically safe* for a language where all safety-checks are performed at run-time, and (with a little loss of precision) *statically safe* for one where as many safety-checks as possible are performed statically, with only the undecidable ones relegated to run-time.

28.3 The Central Theorem: Type Soundness

We have seen earlier [section 27.6.2] that certain type languages can offer very strong theorems about their programs: for instance, that all programs in the language terminate. In general, of course, we cannot obtain such a guarantee (indeed, we added general recursion precisely to let ourselves write unbounded loops). However, a meaningful type system—indeed, anything to which we wish to bestow the noble title of a *type system*—ought to provide some kind of meaningful guarantee that all typed programs enjoy. This is the payoff for the programmer: by typing this program, she can be certain that certain bad things will certainly not happen. Short of this, we have just a bug-finder; while it may be useful, it is not a sufficient basis for building any higher-level tools (e.g., for obtaining security or privacy or robustness guarantees).

What theorem might we want of a type system? Remember that the type checker runs over the static program, before execution. In doing so, it is essentially making a *prediction* about the program's behavior: for instance, when it states that a particular complex term has type `Number`, it is predicting that when run, that term will produce a numeric value. How do we know this prediction is sound, i.e., that the type checker never lies? Every type system should be accompanied by a theorem that proves this.

There is a good reason to be suspicious of a type system, beyond general skepticism. There are many differences between the way a type checker and an interpreter work:

- The type checker sees only program text, whereas the interpreter runs over actual data.
- The type environment binds identifiers to types, whereas the interpreter's environment binds identifiers to values or locations.
- The type checker compresses (even infinite) sets of values into types, whereas the interpreter treats the elements of these sets distinctly.
- The type checker always terminates, whereas the interpreter might not.
- The type checker passes over the body of each expression only once, whereas the interpreter might pass over each body anywhere from zero to infinite times.

Therefore, it is unwise to assume that these two will correspond, and historically, they have often failed to do so.

The central result we wish to have for a given type-system is called *soundness*. It says this. Suppose we are given an expression (or program) `e`. We type-check it and conclude that its type is `t`. When we run `e`, let us say we obtain the value `v`. Then `v` will also have type `t`.

The standard way of proving this theorem is to divide it in two parts, known as *progress* and *preservation*. Progress says that if a term passes the type-checker, it will be able to make a step of evaluation (unless it is already a value); preservation says that the result of this step will have the same type as the original. If we interleave these steps (first progress, then preservation; rinse and repeat), we can conclude that the final answer will indeed have the same type as the original, so the type system is indeed sound.

For instance, consider this expression: `5 + (2 * 3)`. It has the type `Number`. In a sound type system, progress offers a proof that, because this term types, and is not already a value, it can take a step of execution—which it clearly can. After one step the program reduces to `5 + 6`. Sure enough, as preservation proves, this has the same type as the original: `Number`. Progress again says this can take a step, producing `11`. Preservation again shows that this has the same type as the previous expressions representing the program: `Number`. Now progress finds that we are at an answer, so there are no steps left to be taken, and our answer is of the same type as that given for the original expression.

However, this isn't the entire story. There are two caveats:

1. The program may not produce an answer at all; it might loop forever. In this case, the theorem strictly speaking does not apply. However, we can still observe that every intermediate representation of the program has the same type as the whole expression, so the program is computing meaningfully even if it isn't producing a value.

2. Any rich enough language has properties that cannot be decided statically (and others that perhaps could be, but the language designer chose to put off until run-time to reduce the burden on the programmer to make programs pass the type-checker). When one of these properties fails—e.g., the array index being within bounds—there is no meaningful type for the program. Thus, implicit in every type soundness theorem is some set of published, permitted exceptions or error conditions that may occur. The developer who uses a type system implicitly signs on to accepting this set.

As an example of the latter set, the user of a typical typed language acknowledges that vector dereference, list indexing, and so on may all yield exceptions.

A different type system design might make this set a parameter.

The latter caveat looks like a cop-out. However, it is actually a strongly positive statement, in that says any exception not in this set will provably *not* be raised. Of course, in languages designed with static types in the first place, it is not clear (except by loose analogy) what these exceptions might be, because there would be no need to define them. But when we retrofit a type system onto an existing programming language—especially languages with only dynamic enforcement, such as Racket or Python—then there is already a well-defined set of exceptions, and the type-checker is explicitly stating that some set of those exceptions (such as "non-function found in application position" or "method not found") will simply never occur. This is therefore the payoff that the programmer receives in return for accepting the type system's syntactic restrictions.

28.4 Types, Time, and Space

Even in a typed language, it is common to have several run-time checks. To explain this, we will begin with an dynamically-typed account. Consider the following data definition

```
data Tree:
  | base
  | node(v :: Number, l :: Tree, r :: Tree)
end
```

and a function that uses it:

```
fun size(t :: Tree) -> Number:
  cases (Tree) t:
    | base => 0
    | node(_, l, r) => 1 + size(l) + size(r)
  end
end
```

In an dynamically-typed language, every value `t` needs to hold a *type tag* that indicates its type. When a value is passed to `size`, the implementation will check that this is actually a `Tree`. Such a value will have additional *variant tags* that indicate whether it is a `base` or `node` kind of `Tree`. This secondary tag will be used to choose a branch of the `cases` expression.

Assume instead we are in a typed language. The type-checker will have ensured that there no non-`Tree` value could have been substituted for a `Tree`-typed identifier. Therefore, there is no need for the type tag at all. However, the variant tags are still needed, and will be used to dispatch between the branches. In the example, only one bit is needed to tell apart `base` and `node` values. This same bit position can be reused to tell apart variants in some other type without causing any confusion, because the type checker is responsible for keeping the types from mixing.

Type tags would, however, still be needed by the garbage collector, though other representations such as BIBOP can greatly reduce their space impact. The BIBOP scheme appears to be due to Guy Steele, who designed it for MACLISP on the PDP-10 and wrote about it in MIT AI Memo 420.

In other words, if there are two different datatypes that each have two variants, in the dynamically-typed world all these four variants require distinct representations. In contrast, in the typed world their representations can overlap across types, because the static type system will ensure one type's variants are never confused for that the another. Thus, types have a genuine space (saving representation) and time (eliminating run-time checks) performance benefit for programs.

Do Now!

It is conventional in computer science to have a ☞ *space-time tradeoff*. Instead, here we have a situation where we improve *both* space *and* time. This seems almost paradoxical! How is this possible?

This dual benefit comes at some cost to the developer, who must convince the static type system that their program does not induce type errors; due to the limitations of decidability, even programs that might have run without error might run afoul of the type system. Nevertheless, for programs for which this can be done, types provide a notable saving.

28.5 Types Versus Safety

To conclude, we have now identified two classifications for language:

1. Whether or not a language is *typed*, i.e., has static type checks.

2. Whether or not a language's run-time system is *safe*, i.e., performs residual checks not done by a static system—of which there might not even be one).

Given two phenomena with two options each, this suggests there are four different kinds of languages:

	Safe	**Unsafe**
Typed	ML, Java	C, C++
Not Typed	Python, Racket	machine code

The entry for machine code is a little questionable because the language isn't even typed, so there's no classification to check statically. Similarly, there is arguably nothing to check for in the run-time system, so it must best be described as "not even unsafe". However, in practice we do end up with genuine problems, such as security vulnerabilities that arise from being able to jump and execute from arbitrary locations that hold data.

That leaves the truly insidious corner, which languages like C and C++ inhabit. Here, the static type system *gives the impression* that values are actually segregated by type and checked for membership. And indeed they are, in the static world. However, once a programmer passes the type-checker there are no run-time checks. To compound the problem, the language offers primitives like arbitrary pointer arithmetic, making it possible to interpret data of one kind as data of another. As a result, we should have a special place of shame for languages that actively mislead programmers.

Exercise

Construct examples of C or C++ interpreting data of one kind as data of another kind.

Historically, people have sometimes used the phrase *strong typing* to reflect the kind of type-checking that ML and Java use, and *weak typing* for the other kinds. However, these phrases are at best poorly defined.

Do Now!

If you have ever used the phrases “strong typing” or “weak typing”, define them.

That’s what we thought. But thank you for playing.

Indeed, the phrases are not only poorly defined, they are also *wrong*, because the problem is not with the “strength” of the type checker but rather with the nature of the run-time system that backs them. The phrases are even more wrong because they fail to account for whether or not a theorem backs the type system.

It is therefore better to express our intent by sticking to these concepts: *safety*, *typedness*, and *soundness*. Indeed, we should think of this as a continuum. With rare exceptions, we want a language that is safe. Often, we want a language that is also typed. If it is typed, we would like it to be sound, so that we know that the types are not lying. In all these cases, “strong” and “weak” typing do not have any useful meaning.

Chapter 29

Parametric Polymorphism

Which of these is the same?

- `List<String>`
- `List<String>`
- `List<String>`

Actually, none of these is quite the same. But the first and third are very alike, because the first is in Java and the third in ML, whereas the second, in C++, is different. All clear? No? Good, read on!

29.1 Parameterized Types

Consider what would be the intended type of `map` in Pyret:

```
((A -> B), List<A> -> List<B>)
```

This says that for all types `A` and `B`, `map` consumes a function that generates `B` values from `A` values, and a list of `A` values, and generates the corresponding list of `B` values. Here, `A` and `B` are not concrete types; rather, each is a ☞ *type variable* (in our terminology, these should properly be called "type identifiers" because they don't change within the course of an instantiation; however, we will stick to the traditional terminology).

A different way to understand this is that there is actually an infinite family of `map` functions. For instance, there is a `map` that has this type:

```
((Number -> String), List<Number> -> List<String>)
```

and another one of this type (nothing says the types have to be base types):

```
((Number -> (Number -> Number)),
 List<Number> -> List<(Number -> Number)>)
```

and yet another one of this type (nothing says `A` and `B` can't be the same):

```
((String -> String), List<String> -> List<String>)
```

and so on. Because they have different types, they would need different names: `map-num-str`, `map-num-num-to-num`, `map-str-str`, and so on. But that would make them different functions, so we'd have to always refer to a specific `map` rather than each of the generic one.

Obviously, it is impossible to load all these functions into our standard library: there's an infinite number of these! We'd rather have a way to obtain each of these functions on demand. Our naming convention offers a hint: it is as if `map` takes two *type* parameters in addition to its two regular value ones. Given the pair of types as arguments, we can then obtain a `map` that is customized to that particular type. This kind of *parameterization over types* is called *parametric polymorphism*.

Not to be confused with the "polymorphism" of objects, which we will discuss separately [REF].

29.2 Making Parameters Explicit

In other words, we're effectively saying that `map` is actually a function of perhaps four arguments, two of them types and two of them actual values (a function and a list). In a language with explicit types, we might try to write

```
fun map(A :: ???, B :: ???, f :: (A -> B), l :: List<A>)
    -> List<B>:
  ...;
```

but this raises many questions:

- What goes in place of the `???`? These are the types that are going to take the place of `A` and `B` on an actual use. But if `A` and `B` are bound to *types*, then what is their type?

- Do we really want to call `map` with four arguments every time we invoke it?

- Do we want to be passing types—which are static—at the same time as dynamic values?

- If these are types but they are only provided at run-time invocation, how can we type-check clients, who need to know what kind of list they are getting?

The answers to these questions actually lead to a very rich space of polymorphic type systems, most of which we will *not* explore here.

Observe that once we start parameterizing, more code than we expect ends up being parameterized. For instance, consider the type of the humble `link`. Its type really is parametric over the type of values in the list (even though it doesn't actually depend on those values!—more on that in a bit [section 29.6]) so every use of `link` must be instantiated at the appropriate type. For that matter, even `empty` must be instantiated to create an empty list of the correct type! Of course, Java and C++ programmers are familiar with this pain.

29.3 Rank-1 Polymorphism

Instead, we will limit ourselves to one particularly useful and tractable point in this space, which is the type system of Standard ML, of earlier versions of Haskell, roughly that of Java and C# with generics, and roughly that obtained using templates in C++. This language defines what is called *predicative*, *rank-1*, or *prenex* polymorphism.

We first divide the world of types into two groups. The first group consists of the type language we've used until now, but extended to include type variables; these are called *monotypes*. The second group, known as *polytypes*, consists of parameterized types; these are conventionally written with a ∀ prefix, a list of type variables, and then a monotype expression that might use these variables. Thus, the type of `map` would be:

```
∀ A, B : ((A -> B), List<A> -> List<B>)
```

Since "∀" is the logic symbol for "for all", you would read this as: "for all types `A` and `B`, the type of `map` is...".

In rank-1 polymorphism, the type variables can only be substituted with monotypes. (Furthermore, these can only be concrete types, because there would be nothing left to substitute any remaining type variables.) As a result, we obtain a clear separation between the type variable-parameters and regular parameters. We don't need to provide a "type annotation" for the type variables because we know precisely what kind of thing they can be. This produces a relatively clean language that still offers considerable expressive power.

Impredicative languages erase the distinction between monotypes and polytypes, so a type variable can be instantiated with another polymorphic type.

Observe that because type variables can only be replaced with monotypes, they are all independent of each other. As a result, all type parameters can be brought to the front of the parameter list. In Pyret, for instance, the following defines a polymorphic identity function:

<pyret-poly-id> ::=

```
  fun<T> id(x :: T) -> T: x;
```

where `T` is the type parameter. At every use, we separate the provision of *type* parameters from *value* parameters by using `<...>` for the type parameters and

`(...)` for the values. In general, then, we can write types in the form `∀ tv, ... : t` where the `tv` are type variables and `t` is a monotype (that might refer to those variables). This justifies not only the syntax but also the name "prenex". It will prove to also be useful in the implementation.

29.4 Interpreting Rank-1 Polymorphism as Desugaring

The simplest implementation of this feature is to view it as a form of desugaring: this is essentially the interpretation taken by C++. (Put differently, because C++ has a macro system in the form of templates, by a happy accident it obtains a form of rank-1 polymorphism through the use of templates.) Consider the abve polymorphic identity function (*<pyret-poly-id>*). Suppose the implementation is that, on every provision of a type to the name, it replaces the type variable with the given type in the body: given a concrete type for `T`, it yields a procedure of one argument of type `(T -> T)` (where `T` is appropriately substituted). Thus we can instantiate `id` at many different types—

```
id-num = id<Number>
id-str = id<String>
```

—thereby obtaining identity functions at each of those types:

```
check:
  id-num(5) is 5
  id-str("x") is "x"
end
```

In contrast, expressions like

```
id-num("x")
id-str(5)
```

will, as we would expect, *fail to type-check* (rather than fail at run-time).

However, this approach has two important limitations.

1. Let's try to define a recursive polymorphic function, such as `filter`. Earlier we have said that we ought to instantiate every single polymorphic value (such as even `cons` and `empty`) with types, but to keep our code concise we'll focus just on type parameters for `filter`. Here's the code:

```
fun<T> filter(pred :: (T -> Bool), l :: List<T>) -> List<T>:
  cases (List) l:
    | empty => empty
```

```
      | link(f, r) =>
        if pred(f):
          link(f, filter<T>(pred, r))
        else:
          filter<T>(pred, r);
    end
  end
```

Observe that at the recursive uses of `filter`, we must instantiate it with the appropriate type.

This is a perfectly good definition. There's just one problem. If we try to use it—e.g.,

```
filter-num = filter<Number>
```

the implementation will not terminate. This is because the desugarer is repeatedly trying to make new *copies of the code of* `filter` at each recursive call.

Exercise

If, in contrast, we define a local helper function that performs the recursion, this problem can be made to disappear. Can you figure out that version?

2. Consider two instantiations of the identity function. They would necessarily be different because they are two different pieces of code residing at different locations in memory. However, all this duplication is unnecessary! There's absolutely nothing in the body of `id`, for instance, that actually depends on the type of the argument. Indeed, the entire infinite family of `id` functions can share just one implementation. The simple desugaring strategy fails to provide this.

Indeed, the use of parametric polymorphism in C++ is notorious for creating code bloat.

In other words, the desugaring based strategy, which is essentially an implementation by substitution, has largely the same problems we saw earlier with regards to substitution as an implementation of parameter instantiation [section 26.2]. However, in other cases substitution also gives us a ground truth for what we expect as the program's behavior. The same will be true with polymorphism, as we will soon see.

Observe that one virtue to the desugaring strategy is that it does not require our type checker to "know" about polymorphism. Rather, the core type language can continue to be monomorphic, and all the (rank-1) polymorphism is handled entirely through expansion. This offers a cheap strategy for adding polymorphism to a language, though—as C++ shows—it also introduces significant overheads.

Finally, though we have only focused on functions, the preceding discussion applies equally well to data structures.

29.5 Alternate Implementations

There are other implementation strategies that don't suffer from these problems. We won't go into them here, but the essence is to memoize [section 22.3] expansion. Because we can be certain that, for a given set of type parameters, we will always get the same typed body, we never need to instantiate a polymorphic function at the same type twice. This avoids the infinite loop. If we type-check the instantiated body once, we can avoid checking at other instantiations of the same type (because the body will not have changed). Furthermore, we do not need to retain the instantiated sources: once we have checked the expanded program, we can dispose of the expanded terms and retain just one copy at run-time. This avoids all the problems discussed in the pure desugaring strategy shown above, while retaining the benefits.

Actually, we are being a little too glib. One of the benefits of static types is that they enable us to pick more precise run-time representations. For instance, in most languages a static type can tell us whether we have a 32-bit or 64-bit number, or for that matter a 32-bit value or a 1-bit value (effectively, a boolean). A compiler can then generate specialized code for each representation, taking advantage of how the bits are laid out (for example, 32 booleans can use a ☞ *packed representation* to fit into a single 32-bit word). Thus, after type-checking at each used type, the polymorphic instantiator may keep track of all the special types at which a function or data structure was used, and provide this information to the compiler for code-generation. This will then result in several copies of the function, but only as many as those for which the compiler can generate distinct, efficient representations—which is usually fixed, and far fewer than the total number of types a program can use. Furthermore, the decision to make these copies reflects a ☞ *space-time tradeoff*.

29.6 Relational Parametricity

This is a good time to reiterate our recommendation to read Pierce's *Types and Programming Languages*, which covers this topic in the depth it deserves.

There's one last detail we must address regarding polymorphism.

We earlier said that a function like `cons` doesn't depend on the specific values of its arguments. This is also true of `map`, `filter`, and so on. When `map` and `filter` want to operate on individual elements, they take as a parameter another function which in turn is responsible for making decisions about how to treat the elements; `map` and `filter` themselves simply obey their parameter functions.

One way to "test" whether this is true is to substitute some different values in the argument list, and a correspondingly different parameter function. That is, imagine we have a relation between two sets of values; we convert the list elements according to the relation, and the parameter function as well. The question is, will the output from `map` and `filter` also be predictable by the relation? If, for some input, this was not true of the output of `map`, then it must be that `map` somehow affected the value itself, not just letting the function do it. But in fact this won't happen for `map`, or indeed most of the standard polymorphic functions.

Functions that obey this relational rule are called *relationally parametric*. This is another very powerful property that types give us, because they tell us there is a strong limit on the kinds of operations such polymorphic functions can perform: essentially, that they can drop, duplicate, and rearrange elements, but not directly inspect and make decisions on them.

Read Wadler's *Theorems for Free!* and Reynolds's *Types, Abstraction and Parametric Polymorphism*.

At first this sounds very impressive (and it is!), but on inspection you might realize this doesn't square with your experience. In Java, for instance, a polymorphic method can still use `instanceof` to check which particular kind of value it obtained at run-time, and change its behavior accordingly. Such a method would not be relationally parametric! In fact, relational parametricity can equally be viewed as a statement of the weakness of the language: that it permits only a very limited set of operations. (You could still inspect the type—but not act upon what you learned, which makes the inspection pointless. Therefore, a run-time system that wants to simulate relational parametricity would have to remove operations like `instanceof` as well as various proxies to it: for instance, adding `1` to a value and catching exceptions would reveal whether the value is a number.) Nevertheless, it is a very elegant and surprising result, and shows the power of program reasoning possible with rich type systems.

On the Web, you will often find this property described as the inability of a function to inspect the argument—which is not quite right.

Chapter 30

Type Inference

Until now, we have been studying programming languages that require a user to explicitly annotate their programs with types. In languages like Haskell and variants of ML, however, a user can leave out annotations and the language has the ability to automatically infer these annotations for them. For instance, a programmer can write the equivalent of

```
fun f(x, y):
  if x:
    y + 1
  else:
    y - 1
  end
end
```

and the system will automatically infer that the header of `f` ought to have been

```
fun f(x :: Boolean, y :: Number): ...
```

Newer languages like Scala and Typed Racket have this in more limited measure: a feature called *local type inference*. Here, however, we will study the more traditional and powerful form.

30.1 Type Inference as Type Annotation Insertion

First, let's understand what type inference is doing. Some people mistakenly think of languages with inference as having no type declarations, with inference taking their place. This is confused at multiple levels. For one thing, even in languages with inference, programmers are free (and for documentation purposes, are often encouraged) to annotate types. Furthermore, in the absence of such declarations, it is not quite clear what inference actually *means*.

Sometimes, inference is also undecidable and programmers have no choice but to declare some of the types. Finally, writing explicit annotations can greatly reduce indecipherable error messages.

Instead, it is better to think of the underlying language as being fully, explicitly typed, like the languages we have studied [chapter 27]. It is as if the programmer had witten

```
fun f(x :: ___, y :: ___): ...
```

and some programming environment tool had filled in concrete annotations in place of the `___`'s: an especially sophisticated kind of desugaring, as it were. This last remark helps us put inference in perspective: there are really two languages, one with optional type annotations and the other with required ones. Once these annotations are filled in, we are left with a traditional program that can be checked using the methods we have already studied, though in practice this is not necessary [section 30.3]. Thus, inference becomes simply a user convenience for alleviating the burden of writing type annotations, but the language underneath is explicitly typed.

30.2 Understanding Inference

For worked examples and more details, see Chapter 30 of *Programming Languages: Application and Interpretation*.

Suppose we have an expression (or program) *e* written in an explicitly typed language: i.e., *e* has type annotations everywhere they are required. Now suppose we erase all annotations in *e*, and use a procedure `infer` to deduce them back.

Do Now!

What property do we expect of `infer`?

We could demand many things of it. One might be that it produces precisely those annotations that *e* originally had. This is problematic for many reasons, not least that *e* might not even type-check, in which case how could `infer` possibly know what they were (and hence should be)? This might strike you as a pedantic trifle: after all, if *e* didn't type-check, how can erasing its annotations and filling them back in make it do so? Since neither program type-checks, who cares?

Do Now!

Is this reasoning correct?

Suppose *e* is

```
lam(x :: Number) -> String: x end
```

This procedure obviously fails to type-check. But if we erase the type annotations—obtaining

```
lam(x): x end
```

—we equally obviously obtain a typeable function! Therefore, a more reasonable demand might be that if the original *e* type-checks, then so must the version with annotations replaced by inferred types. This one-directional implication is useful in two ways:

1. It does not say what must happen if *e* fails to type-check, i.e., it does not preclude a type inference algorithm that makes the faultily-typed identity function above typeable.

2. More importantly, it assures us that we lose nothing by employing type inference: no program that was previously typeable will now cease to be so. That means we can focus on using explicit annotations where we want to, but will not be forced to do so.

Of course, this only holds if inference is decidable.

We might also expect that both versions type to the same type, but that is not a given: the function

```
lam(x :: Number) -> Number: x end
```

types to `Number -> Number`, whereas applying inference to it after erasing types yields a much more general type, as we will see. Therefore, relating these types and giving a precise definition of type equality is not trivial [section 29.6].

With these preliminaries out of the way, we are now ready to delve into the mechanics of type inference. The most important thing to note is that our simple, recursive-descent type-checking algorithm [section 27.4] will no longer work. That was possible because we already had annotations on all function boundaries, so we could descend into function bodies carrying information about those annotations in the type environment. Sans these annotations, it is not clear how to descend. In fact, it is not clear that there is any particular direction that makes more sense than another.

All this information is in the function. But how do we extract it systematically and in an algorithm that terminates and enjoys the property we have stated above? We do this in two steps. First we *generate constraints*, based on program terms, on what the types must be. Then we *solve constraints* to identify inconsistencies and join together constraints spread across the function body. Each step is relatively simple, but the combination creates magic.

30.2.1 Constraint Generation

Our goal, ultimately, is to find a type to fill into every type annotation position. It will prove to be just as well to find a type for every *expression*. A moment's thought will show that this is likely necessary anyway: for instance, how can we determine the type to put on a function without knowing the type of its body? It is also sufficient, in that if every expression has had its type calculated, this will include the ones that need annotations.

First, we must generate constraints to (later) solve. Constraint generation walks the program source, emitting appropriate constraints on each expression, and returns this set of constraints. It works by recursive descent mainly for simplicity; it really computes a *set* of constraints, so the order of traversal and generation really does not matter in principle—so we may as well pick recursive descent, which is easy—though for simplicity we will use a list to represent this set.

What are constraints? They are simply statements about the types of expressions. In addition, though the binding instances of variables are not expressions, we must calculate their types too (because a function requires both argument and return types). In general, what can we say about the type of an expression?

1. That it is related to the type of some identifier.
2. That it is related to the type of some other expression.
3. That it is a base type, such as numbers and Booleans.
4. That it is a constructed type such as a function, whose domain and range types are presumably further constrained.

The name TyCHS is short for "type (Ty) constraint (C) left-or-right hand (H) side (S)".

Thus, we define the following two datatypes:

```
data TyCon: tyeq(l :: TyCHS, r :: TyCHS) end

data TyCHS:
  | t-expr(e :: TyExprC)
  | t-con(name :: String, fields :: List<TyCHS>)
end
```

Note that we have collapsed both base and constructed types into one representation, `t-con`: a base type will have an empty list of fields, while a constructed type will have a non-empty one corresponding to its arity. Concretely:

```
numeric-t-con = t-con("num", empty)
boolean-t-con = t-con("bool", empty)
fun mk-fun-t-con(a, r):
  t-con("fun", [list: a, r])
end
```

Now we can define the process of generating constraints:

<tyinf-generate> ::=

```
  fun generate(e :: TyExprC) -> List<TyCon>:
    cases (TyExprC) e:
    <tyinf-generate-numC>
    <tyinf-generate-plusC/multC>
    <tyinf-generate-trueC/falseC>
    <tyinf-generate-ifC>
    <tyinf-generate-idC>
    <tyinf-generate-fdC>
    <tyinf-generate-appC>
    end
  end
```

When the expression is a number, all we can say is that we expect the type of the expression to be numeric:

<tyinf-generate-numC> ::=

```
  | numC(_) =>
    [list: tyeq(t-expr(e), numeric-t-con)]
```

This might sound trivial, but what we don't know is what other expectations are being made of this expression by those containing it. Thus, there is the possibility that some outer expression will contradict the assertion that this expression's type must be numeric, leading to a type error.

Identifiers do not constrain the program in any new way. The identifier will (if bound) have its type constrained at the point of binding. Therefore, there are no constraints:

We are assuming that all bound identifiers in the program have distinct names, so there is no danger of confusion between two different identifiers.

<tyinf-generate-idC> ::=

```
  | idC(s) =>
    empty
```

If the context limits its type, then this expression's type will automatically be limited, and must then be consistent with what its context expects of it.

Addition gives us our first look at a contextual constraint. For an addition expression, we must first make sure we generate (and return) constraints in the two sub-expressions, which might be complex. That done, what do we expect? That each of the sub-expressions be of numeric type. (If the form of one of the sub-expressions demands a type that is not numeric, this will lead to a type error.) Finally, we assert that the entire expression's type is itself numeric. Because the treatment of multiplication is identical, we abstract over both:

<tyinf-generate-plusC/multC> ::=

```
  | plusC(l, r) => generate-arith-binop(e, l, r)
  | multC(l, r) => generate-arith-binop(e, l, r)
```

where the interesting work is done by the abstraction:

```
fun generate-arith-binop(e :: TyExprC, l :: TyExprC, r :: TyExprC) ->
  [list: tyeq(t-expr(e), numeric-t-con),
    tyeq(t-expr(l), numeric-t-con),
    tyeq(t-expr(r), numeric-t-con)] +
  generate(l) +
  generate(r)
end
```

Like numbers, Boolean values constrain the current expression to be of Boolean type:

<tyinf-generate-trueC/falseC> ::=

```
  | trueC =>
    [list: tyeq(t-expr(e), boolean-t-con)]
  | falseC =>
    [list: tyeq(t-expr(e), boolean-t-con)]
```

The case for the conditional is again interesting. We must make sure the conditional expression is of Boolean type, and that the two brances have the same type:

<tyinf-generate-ifC> ::=

```
  | ifC(cnd, thn, els) =>
    [list: tyeq(t-expr(cnd), boolean-t-con),
      tyeq(t-expr(thn), t-expr(els)),
      tyeq(t-expr(thn), t-expr(e))] +
    generate(cnd) + generate(thn) + generate(els)
```

Exercise

What happens if you leave out the

```
tyeq(t-expr(thn), t-expr(e))
```

?

Exercise

What if we instead wrote

```
tyeq(t-expr(els), t-expr(e))
```

? Would it make a difference?

Now we get to the other two interesting cases, function declaration and application. In both cases, we must remember to generate and return constraints of the sub-expressions.

In a function definition, the type of the function is a function type, whose argument type is that of the formal parameter, and whose return type is that of the body:

<tyinf-generate-fdC> ::=

```
  | fdC(a, b) =>
    [list:
      tyeq(t-expr(e),
        mk-fun-t-con(t-expr(idC(a)), t-expr(b)))] +
    generate(b)
```

Do Now!

Do you see why we have `idC(a)` instead of just `a`?

This is necessary to make the types work out: `a` on its own is just bound to a string, which is not a `TyExprC`. Of course, the *binding* positions of functions are not truly identifiers, so we're playing fast-and-loose here. In this particular context, we can get away with it, and it saves us having to come up with a whole new representation of programs.

Finally, we have applications. We cannot directly state a constraint on the type of the application. Rather, we can say that the function in the application position must consume arguments of the actual parameter expression's type, and return types of the application expression's type:

<tyinf-generate-appC> ::=

```
  | appC(f, a) =>
    [list:
      tyeq(t-expr(f),
        mk-fun-t-con(t-expr(a), t-expr(e)))] +
    generate(f) +
    generate(a)
```

And that's it! We have finished generating constraints; now we just have to solve them.

30.2.2 Constraint Solving Using Unification

The process used to solve constraints is known as *unification*. A unifier is given a set of equations. Each equation maps a variable to a term, whose datatype is above.

For our purposes, the goal of unification is to generate a *substitution*, or mapping from variables to terms that do not contain any variables. This should sound familiar: we have a set of simultaneous equations in which each variable is used linearly; such equations are solved using *Gaussian elimination*. In that context, we know that we can end up with systems that are both under- and over-constrained. The same thing can happen here, as we will soon see.

The unification algorithm works iteratively over the set of constraints. Because each constraint equation has two terms and each term can be one of two kinds, there are four cases to cover.

The algorithm begins with the set of all constraints, and the empty substitution. Each constraint is considered once and removed from the set, so in principle the termination argument should be utterly simple, but it will prove to be slightly more tricky. As constraints are disposed, the substitution set tends to grow. When all constraints have been disposed, unification returns the final substitution set.

For a given constraint, the unifier examines the left-hand-side of the equation. If it is a variable, it is now ripe for elimination. The unifier adds the variable's right-hand-side to the substitution and, to truly eliminate it, replaces all occurrences of the variable in the substitution with the this right-hand-side.

It is worth noting that because the constraints are equalities, eliminating a variable is tantamount to associating it with the same set as whatever replaces it. In other words, we can use union-find [section 22.1] to implement this process efficiently, though if we need to backtrack during unification (as we do for logic programming [REF]), this becomes much more tricky.

Do Now!

Did you notice the subtle error above?

The subtle error is this. We said that the unifier *eliminates* the variable by replacing all instances of it in the substitution. However, that assumes that the right-hand-side does not contain any instances of the same variable. Otherwise we have a circular definition, and it becomes impossible to perform this particular substitution. For this reason, unifiers include a *occurs check*: a check for whether the same variable occurs on both sides and, if it does, decline to unify. For simplicity we will ignore this here.

Do Now!

Construct a term whose constraints would trigger the occurs check.

Do you remember ω [section 26.4]?

Let us now implement unification. For simplicity, we will use a list of type constraints as the representation of the subtitution.

As you read this, keep in mind that unification is a generic procedure, completely independent of type-inference: indeed, the unification algorithm was invented before and spurred the creation of the type-inference process.

Exercise

If we use type constraints to represent the substitution, what invariant would we expect the computed set of constraints to have?

It will be convenient to have a helper function that takes the current substitution as an accumulated parameter. Let's therefore include it, and get the easy case out of the way:

<tyinf-unify> ::=

```
fun unify(cs :: List<TyCon>) -> List<TyCon>:
  fun help(shadow cs :: List<TyCon>, sub :: List<TyCon>) -> List<TyCon>:
    cases (List) cs:
      | empty => sub
      | link(f, r) =>
        <tyinf-unify-link>
    end
  end
  help(cs, empty)
end
```

There are four cases we need to consider, because either side can be a `t-expr` or `t-con`:

- If both sides are `t-expr`'s, then we simply replace one with the other (this is the "variable elimination" case of the Gaussian procedure). We must perform this replacement everywhere: in the remaining terms but also in the substitution already performed.

Exercise

What happens if we miss doing this replacement in one or the other?

- If one side is a `t-expr` and the other a `t-con`, then we have resolved that expression's type to a concrete type. Record this and substitute.
- There are two cases of a `t-expr` and `t-con`: for simplicity, we handle one case and in the other case, rewrite the problem to the former case and recur.

This swapping of sides is legal because these are *equational* constraints.

- If we have to unify two constructors, then they had better be the same constructor! If they are not, we have a type error. If they are, then we recur on their parameters.

Here it is in code:

<tyinf-unify-link> ::=

```
lhs = f.l
rhs = f.r
```

```
  ask:
    | is-t-expr(lhs) and is-t-expr(rhs) then:
      help(subst(lhs, rhs, r), link(f, subst(lhs, rhs, sub)))
    | is-t-expr(lhs) and is-t-con(rhs) then:
      help(subst(lhs, rhs, r), link(f, subst(lhs, rhs, sub)))
    | is-t-con(lhs) and is-t-expr(rhs) then:
      help(link(tyeq(rhs, lhs), r), sub)
    | is-t-con(lhs) and is-t-con(rhs) then:
      if lhs.name == rhs.name:
        help(map2(tyeq, lhs.fields, rhs.fields) + r, sub)
      else:
        raise('type error: ' + lhs.name + ' vs. ' + rhs.name)
      end
  end
```

In terms of proving termination, note that the last two cases do not shrink the input: the third keeps it the same, while the fourth in some cases *grows* it.

The unifier depends on:

```
fun subst(to-rep :: TyCHS%(is-t-expr), rep-with :: TyCHS, in :: List<Ty
    -> List<TyCon>:
  cases (List) in:
    | empty => empty
    | link(f, r) =>
      lhs = f.l
      rhs = f.r
      link(
        tyeq(
          if lhs == to-rep: rep-with else: lhs end,
          if rhs == to-rep: rep-with else: rhs end),
        subst(to-rep, rep-with, r))
  end
end
```

Exercise

There is a subtle bug in the above implementation of unification. It assumes that two textually identical expressions must have the same type. Construct a counter-example to show that this is not true. Then fix the implementation (consider using reference rather than structural equality [section 19.1]).

Exercise

The algorithm above is rather naive. Given a choice, we would rather see the types of identifiers rather than those of expressions. Modify the algorithm to bias in this direction.

Exercise

The output of the above algorithm is unsatisfying: a set of (solved) constraints rather than an “answer”. Extract the type of the top-level expression, and “pretty-print” it in terms of only type constants, referring to expressions only when necessary (section 30.4).

Exercise

Prove the termination of this algorithm. Make an argument based on the size of the constraint set and on the size of the substitution.

Exercise

Augment this implementation with the occurs check.

Exercise

Use union-find to optimize this implementation. Measure the performance gain.

With this, we are done. Unification produces a substitution. We can now traverse the substitution and find the types of all the expressions in the program, then insert the type annotations accordingly.

30.3 Type Checking and Type Errors

A theorem, which we will not prove here, dictates that the success of the above process implies that the program would have typed-checked, so we need not explicitly run the type-checker over this program.

Observe, however, that the nature of a type error has now changed dramatically. Previously, we had a recursive-descent algorithm that walked a expressions using a type environment. The bindings in the type environment were programmer-declared types, and could hence be taken as (intended) authoritative *specifications* of types. As a result, any mismatch was blamed on the expressions, and reporting type errors was simple (and easy to understand). Here, however, a type error is a *failure to notify*. The unification failure is based on events that occur at the confluence of two smart algorithms—constraint generation and unification—and hence are not necessarily comprehensible to the programmer. In particular, the equational nature of these constraints means that the location reported for the error, and the location of the "true" error, could be quite far apart. As a result, producing better error messages remains an active research area.

In practice the algorithm will maintain metadata on which program source terms were involved and probably on the history of unification, to be able to trace errors back to the source program.

30.4 Over- and Under-Constrained Solutions

Remember that the constraints may not precisely dictate the type of all variables. If the system of equations is *over*-constrained, then we get clashes, resulting in type errors. If instead the system is *under*-constrained, that means we don't have enough information to make definitive statements about all expressions. For instance, in the expression `(fun (x) x)` we do not have enough constraints to indicate what the type of `x`, and hence of the entire expression, must be. This is not an error; it simply means that `x` is free to be *any* type at all. In other words, its type is "the type of `x` `->` the type of `x`" with no other constraints. The types of these underconstrained identifiers are presented as type variables, so the above expression's type might be reported as `(A -> A)`.

The unification algorithm actually has a wonderful property: it automatically computes the *most general types* for an expression, also known as *principal types*. That is, any actual type the expression can have can be obtained by instantiating the inferred type variables with actual types. This is a remarkable result: in another example of computers beating humans, it says that no human can generate a more general type than the above algorithm can!

30.5 Let-Polymorphism

Unfortunately, though these type variables are superficially similar to the polymorphism we had earlier [chapter 29], they are not. Consider the following program:

```
(let ([id (fun (x) x)])
  (if (id true)
      (id 5)
      (id 6)))
```

If we write it with explicit type annotations, it type-checks:

```
(if (id<Boolean> true)
    (id<Number> 5)
    (id<Number> 6))
```

However, if we use type inference, it does not! That is because the `A`'s in the type of `id` unify either with `Boolean` or with `Number`, depending on the order in which the constraints are processed. At that point `id` effectively becomes either a `(Boolean -> Boolean)` or `(Number -> Number)` function. At the use of `id` of the other type, then, we get a type error!

The reason for this is because the types we have inferred through unification are not actually *polymorphic*. This is important to remember: just because you type variables, you don't necessarily have polymorphism! The type variables could be unified at the next use, at which point you end up with a mere monomorphic function. Rather, true polymorphism only obtains when you can *instantiate* type variables.

In languages with true polymorphism, then, constraint generation and unification are not enough. Instead, languages like ML and Haskell implement something colloquially called *let-polymorphism*. In this strategy, when a term with type variables is bound in a lexical context, the type is automatically promoted to be a quantified one. At each use, the term is effectively automatically instantiated.

There are many implementation strategies that will accomplish this. The most naive (and unsatisfying) is to merely *copy the code* of the bound identifier; thus, each use of `id` above gets its own copy of `(fun (x) x)`, so each gets its own type variables. The first might get the type `(A -> A)`, the second `(B -> B)`, the third `(C -> C)`, and so on. None of these type variables clash, so we get the effect of polymorphism. Obviously, this not only increases program size, it also does not work in the presence of recursion. However, it gives us insight into a better solution: instead of copying the code, why not just copy the *type*? Thus at each use, we create a renamed copy of the inferred type: `id`'s `(A -> A)` becomes `(B -> B)` at the first use, and so on, thus achieving the same effect as copying code but without its burdens. Because all these strategies effectively mimic copying code, however, they only work within a lexical context.

Chapter 31

Mutation: Structures and Variables

31.1 Separating Meaning from Notation

Which of these is the same?

- `f = 3`
- `o.f = 3`
- `f = 3`

Assuming all three are in Java, the first and third could behave exactly like each other or exactly like the second: it all depends on whether `f` is a local identifier (such as a parameter) or a field of the object (i.e., the code is really `this.f = 3`).

In either case, we are asking the evaluator to permanently change the value bound to `f`. This has important implications for other observers. Until now, for a given set of inputs, a computation always returned the same value. Now, the answer depends on *when* it was invoked: above, it depends on whether it was invoked before or after the value of `f` was changed. The introduction of time has profound effects on predicting the behavior of programs.

However, there are really two quite different notions of change buried in the uniform syntax above. Changing the value of a field (`o.f = 3` or `this.f = 3`) is extremely different from changing that of an identifier (`f = 3` where `f` is bound as a parameter or a local inside the method, not by the object). We will explore these in turn. We'll tackle fields below, and return to identifiers in section 31.4.

To study both these features, we will as usual write interpreters. However, to make sure we expose their essence, we will write these interpreters *without the use of state*. That is, we will do something quite remarkable: write mutation-free interpreters that faithfully mimic languages with mutation. The key to this will be a special pattern of passing information around in a computation.

31.2 Mutation and Closures

Before we proceed, make sure you've understood boxes (section 21.1), and especially the interaction between mutation and closures (section 21.5).

The interaction with closures (and their first-cousin, objects [chapter 32]) is particularly subtle. Consider the following situation: you are writing a GUI program to implement a calculator. The GUI element corresponding to each calculator button needs a callback—essentially a closure—that, when invoked, will report that that particular button was pressed. Therefore, it is tempting to initialize this as follows:

We first heard this example described by Corky Cartwright.

```
for(var i = 0; i < 10; i++) {
  button[i] = function() { return i; }
}
```

This pseudo-code translates with minimal change to any number of languages, with and without static types, ranging from Python to Swift. (Assume `button` has been initialized appropriately.)

Notice that the closures created above all have `i` in their environment. Now let us try to inspect the behavior of these closures:

```
for(var i = 0; i < 10; i++) {
  println(button[i]())
}
```

That is, we extract the `i`th closure from `button` and apply it to no arguments. This evaluates the `return i` statement.

We might have liked this to produce the sequence of values `0`, `1`, `2`, and so on through to `9`. In fact, however, it produces ten outputs, all the same: `10`.

Do Now!

Do you see why? How would you fix this?

The problem is that `i` in the `for` loop is allocated only once. Therefore, all the closures share *the same* `i`. Because that value had to become `10` for the `for` loop to terminate, all the closures report the value `10`.

With traditional `for` loops, there is no obvious way out of this problem. This seemingly confusing behavior often confounds programmers new to languages that make extensive use of closures. Because they cannot change the behavior of `for` loops, many languages have introduced new versions of `for` (or new keywords inside `for`) to address this problem. The solution is always to allocate a *fresh* `i` on each iteration, so that each closure is over a different variable; the looping construct copies the previous value of `i` as the initial value of the new one before applying the updater (in this case, `i++`) and then performing the comparison and loop body.

Observe that when programming functionally, the desired behavior comes for free. For instance:

```
funs =
  for map(i from range(0, 10)):
    lam(): i end
  end

check:
  map(lam(c): c() end, funs)
    is range(0, 10)
end
```

It is easier to see why the definition of `funs` works by writing the use of `map` more explicitly:

```
funs =
  map(
    lam(i):
      lam():
        i
      end
    end,
    range(0, 10))
```

Thus, we can see that on each iteration the outer `lam(i): ...` is applied, allocating a new `i`, which is the one closed over by the inner `lam(): ...`.

31.3 Mutable Structures

Equipped with these examples, let us now return to adding mutation to the language in the form of mutable structures (which are also a good basis for objects with mutable members). Besides mutable structures themselves, note that we must sometimes perform mutation in groups (e.g., removing money from one bank account and depositing it in another). Therefore, it is useful to be able to sequence a group of mutable operations. We will call this `begin`: it evaluates its sub-terms terms in order and returns the value of the last one.

Exercise

Why does it matter whether `begin` evaluates its sub-terms in some particular, specified order?

Does it matter what this order is?

Exercise

Define `begin` by desugaring into `let` (and hence into anonymous functions).

This is an excellent illustration of the non-canonical nature of desugaring. We've chosen to add to the core a construct that is certainly not necessary. If our goal was to shrink the size of the interpreter—perhaps at some cost to the size of the input program—we would not make this choice. But our goal here is to understand key ideas in languages, so we choose the combination of features that will be most instructive.

Even though it is possible to eliminate `begin` as syntactic sugar, it will prove extremely useful for understanding how mutation works. Therefore, we will add a simple, two-term version of sequencing (`seqC`) to the core. In turn, because our core language is becoming unwieldy, we will drop Boolean values and conditional values except where their presence makes things more interesting: and it does not here.

31.3.1 Extending the Language Representation

First, let's extend our core language datatype:

```
data ExprC:
  | numC(n :: Number)
  | plusC(l :: ExprC, r :: ExprC)
  | multC(l :: ExprC, r :: ExprC)
  | idC(s :: String)
  | appC(f :: ExprC, a :: ExprC)
  | fdC(arg :: String, body :: ExprC)
  | boxC(v :: ExprC)
  | unboxC(b :: ExprC)
  | setboxC(b :: ExprC, v :: ExprC)
  | seqC(b1 :: ExprC, b2 :: ExprC)
end
```

Observe that in a `setboxC` expression, both the box position and its new value are expressions. The latter is unsurprising, but the former might be. It means we can write programs such as this in corresponding Pyret:

```
fun make-box-list():
  b0 = box(0)
  b1 = box(1)
  l = [list: b0, b1]
  index(l, 0)!{v : 1}
  index(l, 1)!{v : 2}
  l
where:
  l = make-box-list()
  index(l, 0)!v is 1
  index(l, 1)!v is 2
end
```

This evaluates to a list of boxes, the first containing `1` and the second `2`. Observe that in each of the mutation statements, we are using a complex expression—e.g., `index(l, 0)`—rather than a literal or an identifier to obtain the box before mutating it (`!{v : 1}`). the first argument to the This is precisely analogous to languages like Java, where one can (taking some type liberties) write

```
public static void main (String[] args) {
    Box<Integer> b0 = new Box<Integer>(0);
    Box<Integer> b1 = new Box<Integer>(1);

    ArrayList<Box<Integer>> l = new ArrayList<Box<Integer>>();
    l.add(b0);
    l.add(b1);

    l.get(0).set(1);
    l.get(1).set(2);
}
```

Notice that `l.get(0)` is a compound expression being used to find the appropriate box, and evaluates to the box object on which `set` is invoked.

For convenience, we will assume that we have implemented desugaring to provide us with (a) `let` and (b) if necessary, more than two terms in a sequence (which can be desugared into nested sequences). We will also sometimes write expressions in the original Pyret syntax, both for brevity (because the core language terms can grow quite large and unwieldy) and so that you can run these same terms in Pyret and observe what answers they produce. As this implies, we are taking the behavior in Pyret—which mutable structures are similar in behavior to those of just about every mainstream language with mutable objects and structures—as the reference behavior.

31.3.2 The Interpretation of Boxes

First, because we've introduced a new kind of value, the box, we have to update the set of values:

<mut-str-value/1> ::=

```
data Value:
  | numV(n :: Number)
  | closV(f :: ExprC%(is-fdC), e :: Env)
  | boxV(v :: Value)
end
```

Now let's begin by reproducing our current interpreter:

<mut-str-interp/1> ::=

```
fun interp(e :: ExprC, nv :: Env) -> Value:
  cases (ExprC) e:
    | numC(n) => numV(n)
    | plusC(l, r) => plus-v(interp(l, nv), interp(r, nv))
    | multC(l, r) => mult-v(interp(l, nv), interp(r, nv))
    | idC(s) => lookup(s, nv)
    | fdC(_, _) => closV(e, nv)
    | appC(f, a) =>
      clos = interp(f, nv)
      arg-val = interp(a, nv)
      interp(clos.f.body,
        xtnd-env(bind(clos.f.arg, arg-val), clos.e))
    <mut-str-interp/1-boxC>
    <mut-str-interp/1-unboxC>
    <mut-str-interp/1-seqC>
  end
end
```

(You'll soon see why the `setboxC` case is missing.)

Two of these cases should be easy. When we're given a `box` expression, we simply evaluate the content and return it wrapped in a `boxV`:

<mut-str-interp/1-boxC> ::=

```
| boxC(v) => boxV(interp(v, nv))
```

Similarly, extracting a value from a box is easy:

<mut-str-interp/1-unboxC> ::=

```
| unboxC(b) => interp(b, nv).v
```

By now, you should be constructing a healthy set of test cases to make sure these behave as you'd expect.

Of course, we haven't done any hard work yet. All the interesting behavior is, presumably, hidden in the treatment of `setboxC`. It may therefore surprise you that we're going to look at `seqC` first instead (and you'll see why we included it in the core).

Let's take the most natural implementation of a sequence of two instructions:

<mut-str-interp/1-seqC> ::=

```
  | seqC(b1, b2) =>
    b1-value = interp(b1, nv)
    b2-value = interp(b2, nv)
    b2-value
```

That is, we evaluate the first term, then the second, and return the result of the second.

You should immediately spot something troubling. We bound the result of evaluating the first term, but didn't subsequently do anything with it. That's okay: presumably the first term contained a mutation expression of some sort, and its value is uninteresting. Thus, an equivalent implementation might be this:

<mut-str-interp/1-seqC/2> ::=

```
  | seqC(b1, b2) =>
    interp(b1, nv)
    interp(b2, nv)
```

Not only is this slightly dissatisfying in that it just uses Pyret's sequential behavior, it can't possibly be right! This can only work *only if the result of the mutation is being stored somewhere*. But because our interpreter only computes values and does not perform any mutation itself—because that would be cheating—any mutations in `interp(b1, nv)` are completely lost. This is obviously not what we want. (And therefore, we're not going to even attempt to define what to do in the `setbox` case.)

31.3.3 Can the Environment Help?

Here is an input example that can help:

```
(let ([b (box 0)])
  (begin (begin (set-box! b (+ 1 (unbox b)))
                (set-box! b (+ 1 (unbox b))))
         (unbox b)))
```

In Racket, this evaluates to `2`.

Exercise

Represent this expression in `ExprC`.

Let's consider the evaluation of the inner sequence. In both cases, the expression (the representation of `(set-box! ...)`) is exactly identical. Yet something is changing underneath, because this causes the value of the box to go from `0` to `2`! We can "see" this even more clearly if instead we evaluate

```
(let ([b (box 0)])
  (+ (begin (set-box! b (+ 1 (unbox b)))
            (unbox b))
     (begin (set-box! b (+ 1 (unbox b)))
            (unbox b))))
```

which evaluates to `3`. Here, the two calls to `interp` in the rule for addition are evaluating exactly the same textual expression in both cases. Yet somehow the effects from the left branch of the addition are being felt in the right branch.

Spukhafte Fernwirkung.

If the interpreter is being given precisely the same expression, how can it possibly avoid producing precisely the same answer? The most obvious way is if the interpreter's other parameter, the environment, were somehow different. As of now the exact same environment is sent to both both branches of the sequence and both arms of the addition, so our interpreter—which produces the same output every time on a given input—cannot possibly produce the answers we want.

Here is what we know so far:

1. We must somehow make sure the interpreter is fed different arguments on calls that are expected to potentially produce different results.

2. We must return from the interpreter some record of the mutations made when evaluating its argument expression.

Because the expression is what it is, the first point suggests that we might try to use the environment to reflect the differences between invocations. In turn, the second point suggests that each invocation of the interpreter should also *return* the environment, so it can be passed to the next invocation. So the interpreter should presumably be modified to return *both* the value *and* an updated environment. That is, the interpreter consumes an expression and environment; it evaluates in that environment, updating it as it proceeds; when the expression is done evaluating, the interpreter returns the answer (as it did before), *along with* an updated environment, which in turn is sent to the next invocation of the interpreter. And the treatment of `setboxC` would somehow impact the environment to reflect the mutation.

Before we dive into the implementation, however, we should consider the consequences of such a change. The environment already serves an important purpose: it holds deferred substitutions. In that respect, it already has a precise semantics—given by substitution—and we must be careful to not alter that. One consequence of its tie to substitution is that it is also the *repository of lexical scope information*.

If we were to allow the extended environment escape from one branch of addition and be used in the other, for instance, consider the impact on the equivalent of the following program:

```
(+ (let ([b (box 0)])
     1)
   b)
```

It should be evident that this program has an error: `b` in the right branch of the addition is unbound (the scope of the `b` in the left branch ends with the closing of the `let`—if this is not evident, desugar the above expression to use functions). But the extended environment at the end of interpreting the `let` clearly has `b` bound in it.

Exercise

Work out the above problem in detail and make sure you understand it.

You could try various other related proposals, but they are likely to all have similar failings. For instance, you may decide that, because the problem has to do with additional bindings in the environment, you will instead remove all added bindings in the returned environment. Sounds attractive? Did you remember we have closures?

Exercise

Consider the representation of the following program:

```
(let ([a (box 1)])
  (let ([f (fun x (+ x (unbox a)))])
    (begin
      (set-box! a 2)
      (f 10))))
```

What problems does this example cause?

Rather, we should note that while the *constraints* described above are all valid, the *solution* we proposed is not the only one. Observe that neither condition actually requires the environment to be the responsible agent. Indeed, it is quite evident that the environment *cannot* be the principal agent. We need something else.

31.3.4 Welcome to the Store

The preceding discussion tells us that we need *two* repositories to accompany the expression, not one. One of them, the environment, continues to be responsible for maintaining lexical scope. But the environment cannot directly map identifiers to their value, because the value might change. Instead, something else needs to be responsible for maintaining the dynamic state of mutated boxes. This latter data structure is called the *store*.

Like the environment, the store is a partial map. Its domain could be any abstract set of names, but it is natural to think of these as numbers, meant to stand for memory locations. This is because the store in the semantics maps directly onto (abstracted) physical memory in the machine, which is traditionally addressed by numbers. Thus the environment maps names to locations, and the store maps locations to values:

```
data Binding:
  | bind(name :: String, location :: Number)
end

type Env = List<Binding>
mt-env = empty
xtnd-env = link

data Storage:
  | cell(location :: Number, value :: Value)
end

type Store = List<Storage>
mt-sto = empty
xtnd-sto = link
```

We'll also equip ourselves with a function to look up values in the store, just as we already have one for the environment (which now returns locations instead):

```
fun lookup(s :: String, nv :: Env) -> Number: ...
fun fetch(n :: Number, st :: Store) -> Value: ...
```

Exercise

Fill in the bodies of `lookup` and `fetch`.

With this, we can refine our notion of values to the correct one:

<mut-str-value> ::=
```
  data Value:
    | numV(n :: Number)
    | closV(f :: ExprC, e :: Env)
    | boxV(l :: Number)
  end
```

31.3.5 Interpreting Boxes

Now we have something that the interpreter can return, updated, reflecting mutations during the evaluation of the expression, without having to change the environment in any way. Because a function can return only one value, we will use an ad hoc object with two fields: `v` for the value (which will (effectively) be the same as the one the interpreter originally returned), and `st` for the (potentially) updated store. We will use the following helper function to assemble the result:

```
fun ret(v :: Value, st :: Store): {v : v, st : st} end
```

Exercise

Why do we say "effectively" and "potentially" above?

Hint for "effectively": look at closures.

Thus our interpreter becomes:

<mut-str-interp> ::=

```
  fun interp(e :: ExprC, nv :: Env, st :: Store):
    cases (ExprC) e:
      <mut-str-interp/numC>
      <mut-str-interp/plusC>
      <mut-str-interp/idC>
      <mut-str-interp/fdC>
      <mut-str-interp/appC>
      <mut-str-interp/boxC>
      <mut-str-interp/unboxC>
      <mut-str-interp/setboxC>
      <mut-str-interp/seqC>
    end
  end
```

The easiest one to dispatch is numbers. Remember that we have to return the store reflecting all mutations that happened while evaluating the given expression. Because a number is a constant, no mutations could have happened, so the returned store is the same as the one passed in:

<mut-str-interp/numC> ::=

```
  | numC(n) => ret(numV(n), st)
```

A similar argument applies to closure creation; observe that we are speaking of the *creation*, not *use*, of closures:

<mut-str-interp/fdC> ::=

```
  | fdC(_, _) => ret(closV(e, nv), st)
```

Identifiers are almost as straightforward, though if you are simplistic, you'll get a type error that will alert you that to obtain a value, you must now look up both in the environment `and in the store`:

<mut-str-interp/idC> ::=

```
  | idC(s) => ret(fetch(lookup(s, nv), st), st)
```

Notice how `lookup` and `fetch` compose to produce the same result that `lookup` alone produced before.

Now things get interesting.

Let's take sequencing. Clearly, we need to interpret the two terms.

<mut-str-interp/seqC/alt> ::=

```
  | seqC(b1, b2) =>
    interp(b1, nv, st)
    interp(b2, nv, st)
```

Oh, but wait. The whole point was to evaluate the second term *in the store returned by the first one*—otherwise there would have been no point to all these changes. Therefore, instead we must evaluate the first term, capture the resulting store, and use it to evaluate the second. (Evaluating the first term also yields its value, but sequencing ignores this value and assumes the first term was run purely for its potential mutations.) Thus:

<mut-str-interp/seqC> ::=

```
  | seqC(b1, b2) =>
    b1-value = interp(b1, nv, st)
    interp(b2, nv, b1-value.st)
```

This says to interpret the first term: `interp(b1, nv, st)`; name the resulting value, which contains `v` and `st` fields, `b1-value`; and evaluate the second term in the store from the first: `interp(b2, nv, b1-value.st)`. The result will be the value and store returned by the second term, which is what we expect. The fact that the first term's effect is only on the store can be read from the code because we never use `b1-value.v`.

Do Now!

Spend a moment contemplating the code above. You'll soon need to adjust your eyes to read this pattern fluently.

Now let's move on to the binary arithmetic primitives. These are similar to sequencing in that they have two sub-terms, but in this case we really do care about the value from each branch. As usual, we'll look at only `plusC` since `multC` is virtually identical.

<mut-str-interp/plusC> ::=

```
  | plusC(l, r) =>
    lv = interp(l, nv, st)
    rv = interp(r, nv, lv.st)
    ret(plus-v(lv.v, rv.v), rv.st)
```

Observe that we've repeated the pattern because we have two sub-expressions to evaluate whose values we want to use. Thus the first store (`lv.st`) is used to interpret the second expression, and the overall result returns that of the second (`rv.st`).

Here's an important distinction. When we evaluate a term, we usually use the same environment for all its sub-terms in accordance with the scoping rules of the language. The environment thus flows in a recursive-descent pattern. In contrast, the store is *threaded*: rather than using the same store in all branches, we take the store from one branch and pass it on to the next, and take the result and send it back out. This pattern is called *store-passing style*.

Now the penny drops. We see that store-passing style is our secret ingredient: it enables the environment to preserve lexical scope while still giving a binding structure that can reflect changes. Our intution told us that the environment had to somehow participate in obtaining different results for the same syntactic expression, and we can now see how it does: not directly, by itself changing, but indirectly, by referring to the store, which updates. Now we only need to see how the store itself "changes".

Let's begin with boxing. To store a value in a box, we have to first allocate a new place in the store where its value will reside. The value corresponding to a box will then remember this location, for use in box mutation. To obtain a fresh value each time, we will use the stateful counter example we have seen earlier [section 21.5]:

```
new-loc = mk-counter()
```

Given this, we can now define the interpretation of box creation:

<mut-str-interp/boxC> ::=

```
  | boxC(v) =>
    val = interp(v, nv, st)
    loc = new-loc()
    ret(boxV(loc),
      xtnd-sto(cell(loc, val.v), st))
```

Do Now!

Observe that we have relied above on `new-loc`, which is itself implemented in terms of boxes! This is outright cheating. How would you modify the interpreter so that we no longer need mutation for this little bit of state?

To eliminate `new-loc`, the simplest option would be to add another parameter to and return value from the interpreter, representing the largest address used so far. Every operation that allocates in the store would return an incremented address, while all others would return it unchanged. In other words, this is precisely another application of the store-passing pattern. Writing the interpreter this way would make it extremely unwieldy and might obscure the more important use of store-passing for the store itself, which is why we have not done so. However, it is important to make sure that we can: that's what tells us that we are not reliant on state to add state to the language.

Now that boxes are recording the location in memory, getting the value corresponding to them is easy.

<mut-str-interp/unboxC> ::=

```
  | unboxC(b) =>
    val = interp(b, nv, st)
    ret(fetch(val.v.l, val.st), val.st)
```

It's the same pattern we saw before, where we have to use `fetch` to obtain the actual value residing at that location. Note that we are relying on Racket to halt with an error if the underlying value isn't actually a `boxV`; otherwise it would be dangerous to not check, since this would be tantamount to dereferencing arbitrary memory [REF memory safety].

Let's now see how to update the value held in a box. First we have to evaluate the box expression to obtain a box, and the value expression to obtain the new value to store in it. The box's value is going to be a `boxV` holding a location.

In principle we want to "change", or override, the value at that location in the store. We can do this in two ways.

1. One is to traverse the store, find the old binding for that location, and replace it with the new one, copying all the other store bindings unchanged.

2. The other, lazier, option is to simply extend the store with a new binding for that location, which works provided we always obtain the most recent binding for a location (which is how `lookup` works in the environment, so `fetch` can do the same in the store).

Observe that this latter option forces us to commit to lists rather than to sets.

The code below is written to be independent of these options:

<mut-str-interp/setboxC> ::=

```
  | setboxC(b, v) =>
    b-val = interp(b, nv, st)
    v-val = interp(v, nv, b-val.st)
    ret(v-val.v,
      xtnd-sto(cell(b-val.v.l, v-val.v), v-val.st))
```

If we've implemented `xtnd-sto` as `link` above, we've actually taken the lazier (and slightly riskier, because of its dependence on the implementation of `fetch`) option.

Exercise

Implement the other version of store alteration, whereby we update an existing binding and thereby avoid multiple bindings for a location in the store.

Exercise

When we look for a location to override the value stored at it, can the location fail to be present? If so, write a program that demonstrates this. If not, explain what invariant of the interpreter prevents this from happening.

Alright, we're now done with everything other than application! Most of application should already be familiar: evaluate the function position, evaluate the argument position, interpret the closure body in an extension of the closure's environment...but how do stores interact with this?

<mut-str-interp/appC> ::=

```
  | appC(f, a) =>
    clos = interp(f, nv, st)
    clos-v :: Value = clos.v
    clos-st :: Store = clos.st
    arg-val = interp(a, nv, clos-st)
    <mut-str-interp/appC/core>
```

Let's start by thinking about extending the closure environment. The name we're extending it with is obviously the name of the function's formal parameter. But what location do we bind it to? To avoid any confusion with already-used locations (a confusion we will explicitly introduce later!—section 31.4.3), let's just allocate a new location. This location is used in the environment, and the value of the argument resides at this location in the store:

<mut-str-interp/appC/core> ::=

```
  new-loc = new-loc()
  interp(clos-v.f.body,
    xtnd-env(bind(clos-v.f.arg, new-loc), clos-v.e),
    xtnd-sto(cell(new-loc, arg-val.v), arg-val.st))
```

Because we have not said the function parameter is mutable, there is no real need to have implemented procedure calls this way. We could instead have followed the same strategy as before. Indeed, observe that the mutability of this location will never be used: only `setboxC` changes what's in an existing store location (the `xtnd-sto` above is technically a store *initialization*), and then only when they are referred to by `boxV`s, but no box is being allocated above. However, we have chosen to implement application this way for uniformity, and to reduce the number of cases we'd have to handle.

You could call this the useless app store.

Exercise

It's a useful exercise to try to limit the use of store locations *only* to boxes. How many changes would you need to make?

31.3.6 Implementing Mutation: Subtleties and Variations

Even though we've finished the implementation, there are still many subtleties and insights to discuss.

1. Implicit in our implementation is a subtle and important decision: the *order of evaluation*. For instance, why did we not implement addition thus?

   <*mut-str-interp/plusC/alt*> ::=

   ```
   | plusC(l, r) =>
     rv = interp(r, nv, st)
     lv = interp(l, nv, rv.st)
     ret(plus-v(lv.v, rv.v), lv.st)
   ```

   It would have been perfectly consistent to do so. Similarly, embodied in the pattern of store-passing is the decision to evaluate the function position before the argument. Observe that:

   (a) Previously, we delegated such decisions to the underlying language implementation. Now, store-passing has forced us to *sequentialize* the computation, and hence make this decision ourselves (whether we realized it or not).

   (b) Even more importantly, *this decision is now a semantic one*. Before there were mutations, one branch of an addition, for instance, could not affect the value produced by the other branch. Because each branch can have mutations that impact the value of the other, we *must* choose some order so that programmers can predict what their program is going to do! Being forced to write a store-passing interpreter has made this clear.

The only effect they could have was halting with an error or failing to terminate—which, to be sure, are certainly observable effects, but at a much more gross level. A program would not terminate with two different answers depending on the order of evaluation.

2. Observe that in the application rule, we are passing along the *dynamic* store, i.e., the one resulting from evaluating both function and argument. This is precisely the opposite of what we said to do with the environment. This distinction is critical. The store is, in effect, "dynamically scoped", in that it reflects the history of the computation, not its lexical shape. Because we are already using the term "scope" to refer to the bindings of identifiers, however, it would be confusing to say "dynamically scoped" to refer to the store. Instead, we simply say that it is *persistent*.

   Languages sometimes dangerously conflate these two. In C, for instance, values bound to local identifiers are allocated (by default) on the stack. However, the stack matches the environment, and hence disappears upon completion of the call. If the call, however, returned references to any of these values, these references are now pointing to unused or even overridden memory: a genuine source of serious errors in C programs. The problem is that programmers want the values themselves to persist; but the storage for those values has been conflated with that for identifiers, who come and go with lexical scope.

3. We have already discussed how there are two strategies for overriding the store: to simply extend it (and rely on `fetch` to extract the newest one) or to "search-and-replace". The latter strategy has the virtue of not holding on to useless store bindings that will can never be obtained again.

   However, this does not cover all the wasted memory. Over time, we cease to be able to access some boxes entirely: e.g., if they are bound to only one identifier, and that identifier is no longer in scope. These locations are called *garbage*. Thinking more conceptually, garbage locations are those whose elimination does not have any impact on the value produced by a program. There are many strategies for *automatically* identifying and reclaiming garbage locations, usually called *garbage collection* [REF].

4. It's very important to evaluate every expression position and thread the store that results from it. Consider, for instance, this alternate implementation of `unboxC` (compare with *<mut-str-interp/unboxC>*):

   <mut-str-interp/unboxC/alt-1> ::=

   ```
     | unboxC(b) =>
       val = interp(b, nv, st)
       ret(fetch(val.v.l, st), val.st)
   ```

   Did you notice? We `fetch`ed the location from `st`, not `val.st`. But `st` reflects mutations up to but before the evaluation of the `unboxC` expression, not any *within* it. Could there possibly be any? Mais oui!

```
(let ([b (box 0)])
  (unbox (begin (set-box! b 1)
                b)))
```

With the incorrect code above, this would evaluate to `0` rather than `1`.

5. Here's another, similar, error (again compare with *<mut-str-interp/unboxC>*):

   <mut-str-interp/unboxC/alt-2> ::=

   ```
     | unboxC(b) =>
       val = interp(b, nv, st)
       ret(fetch(val.v.l, val.st), st)
   ```

   How do we break this? In the end we're returning the old store, the one before any mutations in the `unboxC` happened. Thus, we just need the outside context to depend on one of them.

   ```
   (let ([b (box 0)])
     (+ (unbox (begin (set-box! b 1)
                      b))
        (unbox b)))
   ```

   This should evaluate to `2`, but because the store being returned is one where `b`'s location is bound to the representation of `0`, the result is `1`.

   If we combined both bugs above—i.e., using `st` twice in the last line instead of `s-a` twice—this expression would evaluate to `0` rather than `2`.

   Exercise

   Go through the interpreter; replace every reference to an updated store with a reference to one before update; make sure your test cases catch all the introduced errors!

6. Observe that these uses of "old" stores enable us to perform a kind of *time travel*: because mutation introduces a notion of time, these enable us to go back in time to when the mutation had not yet occurred. This sounds both interesting and perverse; does it have any use?

   It does! Imagine that instead of directly mutating the store, we introduce the idea of a journal of *intended* updates to the store. The journal flows in a threaded manner just like the real store itself. Some instruction creates a new journal; after that, all lookups first check the journal, and only if the journal cannot find a binding for a location is it looked for in the actual store. There are two other new instructions: one to *discard* the journal (i.e., travel back in time), and the other to *commit* it (i.e., all of its edits get applied to the real store).

This is the essence of *software transactional memory*. Each thread maintains its own journal. Thus, one thread does not see the edits made by the other before committing (because each thread sees only its own journal and the global store, but not the journals of other threads). At the same time, each thread gets its own consistent view of the world (it sees edits it made, because these are recorded in the journal). If the transaction ends successfully, all threads atomically see the updated global store. If the transaction aborts, the discarded journal takes with it all changes and the state of the thread reverts (modulo global changes committed by other threads).

Software transactional memory offers one of the most sensible approaches to tackling the difficulties of multi-threaded programming, if we insist on programming with shared mutable state. Because most computers have only one global store, however, maintaining the journals can be expensive, and much effort goes into optimizing them. As an alternative, some hardware architectures have begun to provide direct support for transactional memory by making the creation, maintenance, and commitment of journals as efficient as using the global store, removing one important barrier to the adoption of this idea.

Exercise

Augment the language with the journal features of software transactional memory.

Exercise

An alternate implementation strategy is to have the environment map names to *boxed* `Values`. We don't do it here because it: (a) would be cheating, (b) wouldn't tell us how to implement the same feature in a language without boxes, (c) doesn't necessarily carry over to other mutation operations, and (d) most of all, doesn't really give us *insight* into what is happening here.

It is nevertheless useful to understand, not least because you may find it a useful strategy to adopt when implementing your own language. Therefore, alter the implementation to obey this strategy. Do you still need store-passing style? Why or why not?

31.4 Variables

Now that we've got structure mutation worked out, let's consider the other case: variable mutation. We have already discussed [section 21.4] our choice of terminology, and seen examplse of their use in Pyret. In particular, Whereas other languages overload the mutation syntax, as we have seen [section 31.1], in Pyret they are kept distinct: `!` mutates fields of objects while `:=` mutates variables. This forces Pyret programmers to confront the distinction we introduced at the beginning of section 31.1. We will, of course, sidestep these syntactic issues in our core language by using different constructs for boxes and for variables.

31.4.1 The Syntax of Variable Assignment

The first thing to note about variable mutation is that, although it too has two subterms like box mutation (`setboxC`), its syntax is fundamentally different. To understand why, let's return to our Java fragment:

```
x = 3;
```

In this setting, we cannot write an arbitrary expression in place of `x`: we must literally write the name of the identifier itself. That is because, if it were an expression position, then we could evaluate it, yielding a value: for instance, if `x` were previously bound to `1`, this would be tantamout to writing the following statement:

```
1 = 3;
```

But this is, of course, nonsensical! We can't assign a new value to `1`, and indeed `1` is pretty much the definition of immutable. What we instead want is to find *where* `x` is in the store, and change the value held over there.

Here's another way to see this. Suppose, in Java, the local variable `o` is already bound to some `String` object:

```
o = new String("an initial string");
```

Say the program now executes

```
o = new String("a new string");
```

Is it trying to change the content of the original string (`"an initial string"`)? Certainly not: the second assignment intends to leave that original string alone; it only wants to change the value that `o` is referring to, so that subsequent references evaluate to this new string (`"a new string"`) object instead.

31.4.2 Interpreting Variables

We'll start by reflecting this in our syntax:

```
data ExprC:
  | numC(n :: Number)
  | plusC(l :: ExprC, r :: ExprC)
  | multC(l :: ExprC, r :: ExprC)
  | varC(s :: String)
  | appC(f :: ExprC, a :: ExprC)
  | fdC(arg :: String, body :: ExprC)
  | setC(v :: String, b :: ExprC)
  | seqC(b1 :: ExprC, b2 :: ExprC)
end
```

Observe that we've jettisoned the box operations, but kept sequencing because it's handy around mutation. Importantly, we've now added the `setC` case, and its first sub-term is not an expression but the literal name of a variable. We've also renamed `idC` to `varC`.

Because we've gotten rid of boxes, we can also get rid of the special box values. When the only kind of mutation you have is variables, you don't need new kinds of values.

```
data Value:
  | numV (n :: Number)
  | closV (f :: ExprC, e :: List<Binding>)
end
```

As you might imagine, to support variables we need the same store-passing style that we've seen before [section 31.3.5], and for the same reasons. What differs is in precisely how we use it. Because sequencing is interpreted in just the same way (observe that the verb for it does not depend on boxes versus variables), that leaves us just the variable mutation case to handle.

First, we might as well evaluate the value expression and obtain the updated store:

<mut-var-interp/setC> ::=

```
  | setC(v, b) =>
    new-val = interp(b, nv, st)
    <mut-var-interp/setC/core>
```

What now? Remember we just said that we don't want to fully evaluate the variable, because that would just give the value it is bound to. Instead, we want to know which memory location it corresponds to, and update what is stored at that memory location; this *latter* part is just the same thing we did when mutating boxes:

<mut-var-interp/setC/core> ::=

```
  var-loc = lookup(v, nv)
  ret(new-val.v,
    xtnd-sto(cell(var-loc, new-val.v), new-val.st))
```

The very interesting new pattern we have here is this. When we added boxes, in the `idC` case (<*mut-str-interp/idC*>), we looked up an identifier in the environment (just as above), *and then immediately fetched the value at that location from the store*; the composition yielded a value, just as it used to before we added stores. Now, however, we have a new pattern: looking up an identifier in the environment *without* subsequently fetching its value from the store, i.e., we have "half" of a variable's evaluation. The result of invoking just `lookup` is traditionally called an *l-value*, for "left-hand-side (of an assignment) value". This is a fancy way of saying "memory address", and stands in contast to the actual values that the store yields: observe that it does not directly correspond to anything in the type `Value`.

And we're done! We did all the hard work when we implemented store-passing style (and also in that application allocated new locations for variables).

31.4.3 Reference Parameter Passing

Let's return to the parenthetical statement above: that every application allocates a fresh location in the store for the parameter.

Do Now!

Why does this matter? Consider the following Pyret program:

```
fun f(x):
  x := 3
end

var y = 5
f(y)
```

After this runs, what do we expect to be the value of `y`?

In the example above, `y` evaluates to `5`, not `3`. That is because the value of the formal parameter `x` is held at a different location than that of the actual parameter `y`, so the mutation affects the location of `x`, leaving `y` unscathed.

Now suppose, instead, that application behaved as follows. When the actual parameter is a variable, and hence has a location in memory, instead of allocating a new location for the value, it simply passes along the existing one for the variable. Now the formal parameter is referring to the *same store location* as the actual: i.e., they are *variable aliases*. Thus any mutation on the formal will leak back out into the calling context; the above program would evaluate to `3` rather than `5`. These is called a *call-by-reference* parameter-passing strategy.

Instead, our interpreter implements *call-by-value*, and this is the same strategy followed by languages like Java. This causes confusion because *when the value is itself mutable*, changes made to the value in the callee are observed by the caller. However, that is simply an artifact of mutable values, not of the calling strategy. Please avoid this confusion!

For some years, this power was considered a good idea. It was useful because programmers could write abstractions such as `swap`, which swaps the *value of two* *variables* in the caller. However, the disadvantages greatly outweigh the advantages:

- A careless programmer can alias a variable in the caller and modify it without realizing they have done so, and the caller may not even realize this has happened until some obscure condition triggers it.
- Some people thought this was necessary for efficiency: they assumed the alternative was to *copy* large data structures. However, call-by-value is compatible with passing just the address of the data structure. You only need make a copy if (a) the data structure is mutable, (b) you do not want the caller to be able to mutate it, and (c) the language does not itself provide immutability annotations or other mechanisms.
- It can force non-local and hence non-modular reasoning. For instance, suppose we have the procedure:

  ```
  fun f(g):
    var x = 10
    g(x)
    ...
  end
  ```

  If the language were to permit by-reference parameter passing, then the programmer cannot locally—i.e., just from the above code—determine what the value of `x` will be in the ellipses, because it depends on precisely who the callee (which is being passed in as a parameter) will be, and what it might do, which in turn may depend on dynamic conditions (including the phase of the moon).

At the very least, then, if the language is going to permit by-reference parameters, it should let the *caller* determine whether to pass the reference—i.e., let the callee share the memory address of the caller's variable—or not. However, even this option is not quite as attractive as it may sound, because now the callee faces a symmetric problem, not knowing whether its parameters are aliased or not. In traditional, sequential programs this is less of a concern, but if the procedure is *reentrant*, the callee faces precisely the same predicaments.

At some point, therefore, we should consider whether any of this fuss is worthwhile. Instead, callers who want the callee to perform a mutation could simply send a boxed value to the callee. The box signals that the caller accepts—indeed, invites—the callee to perform a mutation, and the caller can extract the value when it's done. This does obviate the ability to write a simple swapper, but that's a small price to pay for genuine software engineering concerns.

31.5 The Design of Stateful Language Operations

Though most programming languages include one or both kinds of state we have studied, their admission should not be regarded as a trivial or foregone matter. On the one hand, state brings some vital benefits:

- State provides a form of *modularity*. As our very interpreter demonstrates, without explicit stateful operations, to achieve the same effect:

  - We would need to add explicit parameters and return values that pass the equivalent of the store around.
  - These changes would have to be made to *all* procedures that may be involved in a communication path between producers and consumers of state.

  Thus, a different way to think of state in a programming language is that it is an *implicit parameter already passed to and returned from all procedures*, without imposing that burden on the programmer. This enables procedures to communicate "at a distance" without all the intermediaries having to be aware of the communication.

- State makes it possible to construct dynamic, cyclic data structures, or at least to do so in a relatively straightforward manner [chapter 20].

- State gives procedures *memory*, such as `new-loc` above. If a procedure could not remember things for itself, the callers would need to perform the remembering on its behalf, employing the moral equivalent of (at least local) store-passing. This is not only unwieldy, it creates the potential for a caller to interfere with the memory for its own nefarious purposes (e.g., a caller might purposely send back an old store, thereby obtaining a reference already granted to some other party, through which it might launch a correctness or security attack).

On the other hand, state imposes real costs on programmers as well as on programs that process programs (such as compilers). One is "aliasing", which we discuss later [REF]. Another is "referential transparency", which too we hope to return to [REF]. Finally, we have described above how state provides a form of modularity. However, this same description could be viewed as that of a back-channel of communication that the intermediaries did not know and could not monitor. In some (especially security and distributed system) settings, such back-channels can lead to collusion, and can hence be extremely dangerous and undesirable.

Because there is no optimal answer, it is probably wise to include mutation operators but to carefully delinate them. In Standard ML, for instance, there is no variable mutation, because it is considered unnecessary. Instead, the language has the equivalent of boxes (called `ref`s). One can easily simulate variables using boxes, so no expressive power is lost, though it does create more potential for aliasing than variables alone would have ([REF aliasing]) if the boxes are not used carefully.

In return, however, developers obtain expressive *types*: every data structure is considered immutable unless it contains a `ref`, and the presence of a `ref` is a warning to both developers and programs (such as compilers) that the underlying value may keep changing. Thus, for instance, suppose `b` is a box and `v` is bound to the unboxing of `b`. A developer should be aware that replacing all instances of the unboxing `b` with references to `v` is not safe, because the former always fetches the *current* value in the box while the latter holds the value only at the instant when `v` was computed, and may now be inconsistent. The declaration that a field is mutable provides this information to both the developer and to programming tools (such as compilers); in particular, the absence of such a declaration permits caching of values, thereby trading computation time for space.

This same argument applies to Pyret, where the absence of a `ref` declaration means that a field is immutable, and the absence of a `var` declaration means an identifier is immutable, i.e., not a variable.

31.6 Typing State

Adding stateful operations to a type-checker is easy: the only safe thing to do is make sure the type of the new value is exactly the same as that of the old one. If that is true, the behavior of the program will be indistinguishable to the type system before and after the mutation. That is, it is safest to follow invariance [section 27.2].

31.6.1 Mutation and Polymorphism

Later, once we encounter subtyping [section 32.6.1], we will find it is no longer trivial to type mutation. Already, however, mutation is complex in the presence of polymorphism. Let us first give polymorphic types to boxes, where `Box` is a type constructor for boxes:

```
box<T> :: T -> Box(T)
unbox<T> :: Box(T) -> T
set-box<T> :: Box(T), T -> Box(T)
```

(assuming that the mutation operation, `set-box`, returns the box as its result).

Exercise

Implement the above three functions.

Now let us consider a simple example that uses these:

```
let f = box(lam(x): x end):
  set-box(f, lam(x): x + 5 end)
  unbox(f)(true)
end
```

Initially, the type of `f` is `Box((A -> A))` where `(A -> A)` represents the type of the polymorphic identity function. When performing inference, a copy of this type certainly unifies with `Number -> Number`, the type of `lam(x): x + 5 end`. Another copy is used at the application site; the argument type unifies with that of `true`, giving the whole expression the type `Boolean`. However, when the actual function is applied, it attempts to add `true` to `5`, resulting in a run-time error. If the compiler had assumed the type-system was sound and had not compiled in checks, this program could even result in a segmentation fault.

There are many ways to try to understand this problem, which is beyond the scope of this study. The simplest way is that polymorphism and mutation do not work together well. The essence of polymorphism is to imagine a quantified type is instantiated at *each* use; however, the essence of mutation is to silently transmit values from one part of the program to the other. Thus, the values being unified at two different sites are only guaranteed to be compatible with the `let`-bound identifier—*not with each other*. This is normally not a problem because the two do not communicate directly, except where mutation is involved. Therefore, a simple solution in this case is to prevent polymorphic generalization for mutable `let`-bound identifiers.

31.6.2 Typing the Initial Value

There is one last issue we should address where mutation is concerned: the typing of cycle creation, and in particular dealing with the problem of section 21.3.3. We have discussed several approaches to handling the initial value; each of these has consequences for typing:

1. Using a fixed initial value of a standard type means the value subsequently mutated into place may not be type-compatible, thereby failing invariance.

2. Using a different initial value of the type that will eventually be put into the mutable has the problem that prematurely observing it is even more deadly, because it may not be distinguishable from the eventual value.

3. Using a new value just for this case works provided there is one *of each type*. Otherwise, again, we violate invariance. But having one of each type is a problem in itself, because now the run-time system has to check for all of them.

4. Syntactically restricting recursion to functions is the safest, because the initial value is never seen. As a result, there is no need to provide any meaningful type for it.

In short, this is a place where we have to confront unsurmountable trade-offs. The first option sacrifices typability; the second option sacrifices program reliability (because the dummy values are of the right type, and may hence be inadvertently used without noticing they are wrong); the third sacrifices run-time simplicity; and the fourth sacrifices programmer flexibility.

Chapter 32

Objects: Interpretation and Types

When a language admits functions as values, it provides developers the most natural way to represent a unit of computation. Suppose a developer wants to parameterize some function `f`. Any language lets `f` be parameterized by *passive* data, such as numbers and strings. But it is often attractive to parameterize it over *active* data: a datum that can *compute* an answer, perhaps in response to some information. Furthermore, the function passed to `f` can—assuming lexically-scoped functions—refer to data from the caller without those data having to be revealed to `f`, thus providing a foundation for security and privacy. Thus, lexically-scoped functions are central to the design of many secure programming techniques.

While a function is a splendid thing, it suffers from excessive terseness. Sometimes we might want multiple functions to all close over to the same *shared* data; the sharing especially matters if some of the functions mutate it and expect the others to see the result of those mutations. In such cases, it becomes unwieldly to send just a single function as a parameter; it is more useful to send a group of functions. The recipient then needs a way to choose between the different functions in the group. This grouping of functions, and the means to select one from the group, is the essence of an *object*. We are therefore perfectly placed to study objects having covered functions [section 26.3], mutation [chapter 31], and recursion [section 21.3].

We cannot hope to do justice to the enormous space of object systems. Please read *Object-Oriented Programming Languages: Application and Interpretation* by Éric Tanter, which goes into more detail and covers topics ignored here.

Let's add this notion of objects to our language. Then we'll flesh it out and grow it, and explore the many dimensions in the design space of objects. We'll first show how to add objects to the core language, but because we'll want to prototype many different ideas quickly, we'll soon shift to a desugaring-based strategy. Which one you use depends on whether you think understanding them is critical to understanding the essence of your language. One way to measure this is how complex your desugaring strategy becomes, and whether by adding some key core language enhancements, you can greatly reduce the complexity of desugaring.

32.1 Interpreting Objects

The simplest notion of an object—pretty much the only thing everyone who talks about objects agrees about—is that an object is

- a value, that
- maps names to
- stuff: either other values or “methods”.

From a minimalist perspective, methods seem to be just functions, and since we already have those in the language, we can put aside this distinction.

We’re about to find out that “methods” are awfully close to functions but differ in important ways in how they’re called and/or what’s bound in them.

Starting from the language with variables, let’s define this very simple notion of objects by adding it to the core language. We clearly have to extend our notion of values:

```
data Value:
  | numV(n :: Number)
  | closV(f :: ExprC, e :: List<Binding>)
  | objV(ns :: List<String>, vs :: List<Value>)
end
```

We’ll extend the expression grammar to support literal object construction expressions:

Observe that this is already a design decision. In some languages, like JavaScript, a developer can write literal objects: a notion so popular that a subset of the syntax for it in JavaScript has become a Web standard, JSON. In other languages, like older versions of Java, objects can only be created by invoking a constructor on a class. We can simulate both by assuming that to model the latter kind of language, we must write object literals only in special positions following a stylized convention, as we do when desugaring below.

```
| objC(ns :: List<String>, vs :: List<ExprC>)
```

Evaluating such an object expression is easy: we just evaluate each of its expression positions. In the presence of state, however, we have to be careful to thread the store:

```
| objC(ns, vs) =>
  obj-vs = eval-obj-vs(vs, nv, st)
  ret(objV(ns, obj-vs.exprs), obj-vs.final-store)
```

Exercise

Write `eval-obj-vs`, which evaluates each expression in `vs` while threading the store. Assume it returns an object with two fields: `exprs` is the list of evaluated expressions, while `final-store` is the final store ensuing from these evaluations.

Unfortunately, we can't actually *use* an object, because we have no way of obtaining its content. For that reason, we should add an operation to extract members:
<msgC-def> ::=
```
  | msgC(o :: ExprC, n :: String)
```
whose behavior is intuitive:

```
| msgC(o, n) =>
  o-val = interp(o, nv, st)
  msg = lookup-msg(n, o-val.v)
  ret(msg, o-val.st)
```

Exercise

Implement `lookup-msg`.

In principle, `msgC` can be used to obtain any kind of member but for simplicity, we need only assume that we have functions. To use them, we must apply them to values. This is cumbersome to write directly, so let's assume desugaring has taken care of it for us: that is, the user can write `(msg o m v)`—where `o` evaluates to an object, `m` names a method, and `v` evaluates to an argument value—and this desugars into using `msgC` to obtain the method and regular application to apply it.

For illustration, we'll assume methods take only one argument. This is easy to relax. Note that in a Lispy language we could have instead written `(define (msg o m . a) (appl` which would have let `msg` take any number of arguments.

With this we have a full first language with objects. For instance, here is an object definition and invocation:
```
(let o (obj (add1 (lambda x (+ x 1)))
            (sub1 (lambda x (+ x -1))))
  (msg o sub1 2))
```
and this evaluates to `(numV 1)`.

32.2 Objects by Desugaring

While defining objects in the core language is good to really understand their essence, it's an unwieldy way to go about studying them. Instead, we'll use Pyret to represent objects, sticking to the parts of the language we already know how to implement in our interpreter. That is, we'll assume that we are looking at the *output of desugaring*. (For this reason, we'll also stick to stylized code, potentially writing unnecessary expressions on the grounds that this is what a simple program generator would produce.)

Exercise

The code that follows largely drops type annotations. Go back in and add these annotations wherever possible; where you can't, explain what problems you encounter. See section 32.6.

32.2.1 Objects as Named Collections

Let's begin by reproducing the object language we had above. An object is just a value that dispatches on a given name. For simplicity, we'll use anonymous functions to represent the object and conditionals to implement the dispatching.

Observe that basic objects are a generalization of anonymous functions to have multiple "entry-points". Conversely, an anonymous functions is an object with just one entry-point, so it doesn't need a "method name" to disambiguate.

<obj-o-1> ::=

```
  o-1 =
    lam(m):
      if m == "add1":
        lam(x): x + 1 end
      else if m == "sub1":
        lam(x): x - 1 end
      else:
        raise("message not found: " + m)
      end
    end
```

This is the same object we defined earlier, and we use its method in the same way:

```
check: o-1("add1")(5) is 6 end
```

Of course, writing method invocations with these nested function calls is unwieldy (and is about to become even more so), so we'd be best off equipping ourselves with a convenient syntax for invoking methods, which we can define here as a function:

```
fun msg(o, m, a): o(m)(a) end
```

This enables us to rewrite our test:

```
check: msg(o-1, "add1", 5) is 6 end
```

Do Now!

Something very important changed when we switched to the desugaring strategy. Do you see what it is?

Recall the syntax definition we had earlier: *<msgC-def>*. The "name" position of a message was very explicitly a *syntactic string*. That is, the user had to write the literal name of the method there. In our desugared version, the name position is just an expression that must evaluate to a string; this permits the user to write the following:

```
check: msg(o-1, "add" + "1", 5) is 6 end
```

which may very much not be intended [section 32.3].

This is a general problem with desugaring: the target language may allow computations that have no counterpart in the source, and hence cannot be mapped back to it. Fortunately we don't often need to perform this inverse mapping, though it does arise in some debugging and program comprehension tools. More subtly, however, we must ensure that the target language does not produce *values* that have no corresponding equivalent in the source.

Now that we have basic objects, let's start adding the kinds of features we've come to expect from most object systems. But before we proceed, it's unwieldy to define an object as an explicit conditional; we would rather write a more declarative mapping from names to methods, and leave the implementation of the lookup to the language. This, after all, is one of the key primitives provided by every definition of object-orientation. That is, we wish to write the previous object (<*obj-o-1*>) as

```
o-1-1 = mk-object(
  [list:
    mtd("add1", lam(x): x + 1 end),
    mtd("sub1", lam(x): x - 1 end) ] )
```

and to support this, we define the datatype

```
data Mtd:
  | mtd(name :: String, value)
end
```

and the corresponding function

```
fun mk-object(n-vs):
  lam(m):
    fun lookup(locals):
      cases (List) locals:
        | empty => raise("message not found: " + m)
        | link(f, r) =>
          if f.name == m: f.value else: lookup(r) end
      end
    end
    lookup(n-vs)
  end
```

```
end
```

With this much simpler notation—which does not even require desugaring to implement—we are now better equipped to handle the study of object system features.

32.2.2 Constructors

A constructor is simply a function that is invoked at object construction time. We currently lack such a function. By turning an object from a literal into a function that takes constructor parameters, we achieve this effect:

```
o-constr-1 =
  lam(x):
    mk-object( [list: mtd("addX", lam(y): x + y end) ])
  end

check:
  msg(o-constr-1(5), "addX", 3) is 8
  msg(o-constr-1(2), "addX", 3) is 5
end
```

In the first example, we pass `5` as the constructor's argument, so adding `3` yields `8`. The second is similar, and shows that the two invocations of the constructors don't interfere with one another.

32.2.3 State

Alan Kay, who won a Turing Award for inventing Smalltalk and modern object technology, disagrees. In *The Early History of Smalltalk*, he says, "[t]he small scale [motivation for OOP] was to find a more flexible version of assignment, and then to try to eliminate it altogether". He adds, "It is unfortunate that much of what is called 'object-oriented programming' today is simply old style programming with fancier constructs. Many programs are loaded with 'assignment-style' operations now done by more expensive attached procedures."

Many people believe that objects primarily exist to encapsulate state. We certainly haven't lost that ability. If we desugar to a language with variables (we could equivalently use boxes, in return for a slight desugaring overhead), we can easily have multiple methods mutate common state, such as a constructor argument:

```
o-state-1 =
  lam(count):
    var mut-count = count
    mk-object(
      [list:
        mtd("inc", lam(n): mut-count := mut-count + n end),
        mtd("dec", lam(n): mut-count := mut-count - n end),
        mtd("get", lam(_): mut-count end) ] )
  end
```

For instance, we can test a sequence of operations:

```
check:
  o = o-state-1(5)
  msg(o, "inc", 1)
  msg(o, "dec", 1)
  msg(o, "get", "dummy") is 5
end
```

and also notice that mutating one object doesn't affect another:

```
check:
  o1 = o-state-1(5)
  o2 = o-state-1(5)
  msg(o1, "inc", 1)
  msg(o1, "inc", 1)
  msg(o1, "get", "dummy") is 7
  msg(o2, "get", "dummy") is 5
end
```

32.2.4 Private Members

Another common object language feature is private members: ones that are visible only inside the object, not outside it. These may seem like an additional feature we need to implement, but we already have the necessary mechanism in the form of locally-scoped, lexically-bound variables, such as `mut-count` above: there is no way for surrounding code to access `mut-count` directly, because lexical scoping ensures that it remains hidden to the world.

Except that, in Java, instances of other classes of the same type are privy to "private" members. Otherwise, you would simply never be able to implement an approximation to an Abstract Data Type.

32.2.5 Static Members

Another feature often valuable to users of objects is *static* members: those that are common to all instances of the "same" type of object. This, however, is merely a lexically-scoped identifier (making it private) that lives outside the constructor (making it common to all uses of the constructor), such as `counter` below:

We use quotes because there are many notions of sameness for objects. And then some.

```
mk-bank-account =
  block:
    var counter = 0
    lam(amount):
      var balance = amount
      counter := counter + 1
      mk-object(
        [list:
          mtd("deposit", lam(m): balance := balance + m end),
          mtd("withdraw", lam(m): balance := balance - m end),
          mtd("balance", lam(_): balance end),
          mtd("how-many-accounts", lam(_): counter end) ])
    end
  end
```

We've written the counter increment where the "constructor" for this object would go, though it could just as well be manipulated inside the methods. This obeys the following tests:

```
check:
  acc-1 = mk-bank-account(0)
  msg(acc-1, "how-many-accounts", "dummy") is 1
  acc-2 = mk-bank-account(100)
  msg(acc-1, "how-many-accounts", "dummy") is 2
  msg(acc-2, "how-many-accounts", "dummy") is 2
  msg(acc-1, "deposit", 100)
  msg(acc-1, "withdraw", 50)
  msg(acc-2, "deposit", 10)
  msg(acc-1, "balance", "dummy") is 50
  msg(acc-2, "balance", "dummy") is 110
  msg(acc-1, "how-many-accounts", "dummy") is 2
  msg(acc-2, "how-many-accounts", "dummy") is 2
end
```

Note that the different objects each affect the result seen by the other.

32.2.6 Objects with Self-Reference

Until now, our objects have simply been packages of named functions grouped together and hence given different, named entry-points. We've seen that many of the features considered important in object systems are actually simple patterns over functions and scope, and have indeed been used—without names assigned to them—for decades by programmers armed with anonymous functions (with lexical scope).

One characteristic that actually distinguishes object systems is that each object is automatically equipped with a reference to the same object, often called `self` or `this`. Can we implement this easily?

We prefer this slightly dry way of putting it to the anthropomorphic "knows about itself" terminology often adopted by object advocates. Indeed, note that we have gotten this far into object system properties without ever needing to resort to anthropomorphism.

Self-Reference Using Mutation

Yes, we can, because we have seen just this very pattern when we implemented recursion; we'll just adapt it to refer not to the same box or function but to the same object.

```
o-self =
  block:
    var self = "dummy"
    self :=
      mk-object(
          [list:
            mtd("first",
              lam(v): msg(self, "second", v + 1) end),
            mtd("second",
              lam(v): v + 1 end )])
    self
  end
```

Observe that this is precisely the recursion pattern [section 21.3], adapted slightly. We've tested it having `"first"` invoke its own `"second"` method. Sure enough, this produces the expected answer:

```
check: msg(o-self, "first", 5) is 7 end
```

Self-Reference Without Mutation

If you know how to implement recursion without mutation [REF Y/omega], you'll notice that the same solution applies here, too. Observe:

```
o-self-no-state =
  mk-object(
    [list:
      mtd("first",
        lam(self, v): smsg(self, "second", v + 1) end),
      mtd("second",
        lam(self, v): v + 1 end )])
```

Each method now takes `self` as an argument. That means method invocation must be modified to pass along the object as part of the invocation:

```
fun smsg(o, m, a): o(m)(o, a) end
```

That is, when invoking a method on `o`, we must pass `o` as a parameter to the method. Obviously, this approach is dangerous because we can potentially pass a *different* object as the "`self`". Exposing this to the developer is therefore probably a bad idea; if this implementation technique is used, it should only be done in desugaring.

Nevertheless, Python exposes just this *in its surface syntax*.

Just to be sure, we can check this using essentially the same code as before:

```
check: smsg(o-self, "first", 5) is 7 end
```

32.2.7 Dynamic Dispatch

Finally, we should make sure our objects can handle a characteristic attribute of object systems, which is the ability to invoke a method without the caller having to know or decide which object will handle the invocation. Suppose we have a binary tree data structure, where a tree consists of either empty nodes or leaves that hold a value. In traditional functions, we are forced to implement the equivalent some form of conditional that exhaustively lists and selects between the different kinds of trees. If the definition of a tree grows to include new kinds of trees, each of these code fragments must be modified. Dynamic dispatch solves this problem by eliminating this conditional branch from the user's program and instead handling it by the method selection code *built into the language*. The key feature that this provides is an *extensible conditional*. This is one dimension of the extensibility that objects provide.

This property—which appears to make systems more *black-box extensible* because one part of the system can grow without the other part needing to be modified to accommodate those changes—is often hailed as a key benefit of object-orientation. While this is indeed an advantage objects have over functions, there is a dual advantage that functions have over objects, and indeed many object programmers end up contorting their code—using the Visitor pattern—to make it look more like a function-based organization. Read *Synthesizing Object-Oriented and Functional Design to Promote Re-Use* for a running example that will lay out the problem in its full glory. Try to solve it in your favorite language, and see the Racket solution.

Let's now defined our two kinds of tree objects:

```
mt =
  lam():
    mk-object(
      [list:
        mtd("add",
          lam(self, _): 0 end) ])
  end

node =
  lam(v, l, r):
    mk-object(
      [list:
        mtd("add",
          lam(self, _):
            v
              + smsg(l, "add", "dummy")
              + smsg(r, "add", "dummy") end) ] )
  end
```

With these, we can make a concrete tree:

```
a-tree =
  node(10,
    node(5, mt(), mt()),
    node(15, node(6, mt(), mt()), mt()))
```

And finally, test it:

```
check: smsg(a-tree, "add", "dummy") is (10 + 5 + 15 + 6) end
```

Observe that both in the test and in the `"add"` method of `node`, there is a reference to `"add"` without checking whether the recipient is a `mt` or a `node`. Instead, the run-time system extracts the recipient's `"add"` method and invokes it. This *missing conditional in the user's source program provided automatically by the system* is the essence of dynamic dispatch.

32.3 Member Access Design Space

We already have two orthogonal dimensions when it comes to the treatment of member names. One dimension is whether the name is provided statically or computed, and the other is whether the set of names is fixed or variable:

	Name is Static	**Name is Computed**
Fixed Set of Members	As in base Java.	As in Java with reflection to compute the nam
Variable Set of Members	Difficult to envision (what use would it be?).	Most scripting languages.

Only one case does not quite make sense: if we force the developer to specify the member name in the source file explicitly, then no new members would be accessible (and some accesses to previously-existing, but deleted, members would fail). All other points in this design space have, however, been explored by languages.

The lower-right quadrant corresponds closely with languages that use hash-tables to represent objects. Then the name is simply the index into the hash-table. Some languages carry this to an extreme and use the same representation even for numeric indices, thereby (for instance) conflating objects with dictionaries and even arrays. Even when the object only handles "member names", this style of object creates significant difficulty for type-checking [chapter 27] and is hence not automatically desirable.

Therefore, in the rest of this section, we will stick with "traditional" objects that have a fixed set of names and even static member name references (the top-left quadrant). Even then, we will find there is much, much more to study.

32.4 What (Goes In) Else?

So far, the "else clause" of method lookup (which is currently implemented by `mk-object`)—namely, what to do when the list of methods is empty—has signaled a "method not found" error. What else might happen instead? One possibility, adopted by many programming languages, is to "chain" control to one or more *parent* object(s). This is called *inheritance*.

Let's return to our model of desugared objects above. To implement inheritance, the object must be given "something" to which it can delegate method invocations that it does not recognize. A great deal will depend on what that "something" is.

One answer could be that it is simply another object: where currently we have

```
| empty => raise("message not found: " + m)
```

we could instead have

```
| empty => parent-object(m)
```

Due to our representation of objects, this application effectively searches for the method in the parent object (and, presumably, recursively in its parents). If a method matching the name is found, it returns through this chain to the original call that sought the method. If none is found, the final parent object presumably signals the same "message not found" error.

Exercise

Observe that the application `parent-object(m)` is like "half a `msg`", just like an l-value was "half" a variable's evaluation [section 31.4.2]. Is there any connection?

Let's try this by extending our trees to implement another method, `"size"`. We'll write an "extension" (you may be tempted to say "sub-class", but hold off for now!) for each `node` and `mt` to implement the `size` method. We intend these to extend the existing definitions of `node` and `mt`, so we'll use the extension pattern described above.

We're not editing the existing definitions because that is supposed to be the whole point of object inheritance: to reuse code in a black-box fashion. This also means different parties, who do not know one another, can each extend the same base code. If they had to edit the base, first they have to find out about each other, and in addition, one might dislike the edits of the other. Inheritance is meant to sidestep these issues entirely.

32.4.1 Classes

Immediately we see a design choice. Is this the constructor pattern?

```
node-size-ext =
  fun(parent-object, v, l, r):
    ...
```

That is, we pass the parent object to `node-size-ext` along with the constructor parameters. Since the parent object will be an instance of `node`, and the two objects should presumably have the same values for the parameters, this means we would have had to specify those values twice (which violates the DRY principle). As an alternative, we can simply pass the *constructor* of the parent to `node-size-ext` and let it construct the parent object:

```
node-size-ext =
  lam(parent-maker, v, l, r):
    parent-object = parent-maker(v, l, r)
    mk-ext-object(parent-object,
      [list:
        mtd("size",
          lam(self, _):
            1
              + smsg(l, "size", "dummy")
              + smsg(r, "size", "dummy") end) ] )
  end
```

Using this, we can make a more user-friendly interface to nodes with the size method:

```
fun node-size(v, l, r): node-size-ext(node, v, l, r) end
```

Do Now!

Did you notice that instead of `mk-object` we've used `mk-ext-object` above? Do you see that it takes one extra parameter? Try to define it for yourself.

The entire difference in `mk-ext-object` is that, if it cannot find a method in the current object, it chains to the parent:

```
fun mk-ext-object(parent, n-vs):
  lam(m):
    fun lookup(locals):
      cases (List) locals:
        | empty => parent(m)
        | link(f, r) =>
          if f.name == m: f.value else: lookup(r) end
      end
    end
    lookup(n-vs)
  end
end
```

With this, we can similarly create an extension of empty tree nodes:

```
mt-size-ext =
  lam(parent-maker):
    parent-object = parent-maker()
    mk-ext-object(parent-object,
      [list:
        mtd("size",
          lam(self, _): 0 end) ])
  end

fun mt-size(): mt-size-ext(mt) end
```

Finally, we can use these objects to construct a tree:

```
a-tree-size =
  node-size(10,
    node-size(5, mt-size(), mt-size()),
    node-size(15, node-size(6, mt-size(), mt-size()), mt-size()))
```

When testing, we should make sure both old and new behavior work:

```
check:
  smsg(a-tree-size, "add", "dummy") is (10 + 5 + 15 + 6)
  smsg(a-tree-size, "size", "dummy") is 4
end
```

Exercise

Earlier, we commented that chaining method-lookup to parents presumably bottoms out at some sort of "empty object", which might look like this:

```
fun empty-object(m):
  raise("message not found: " + m)
end
```

However, we haven't needed to define or use this despite the use of `mk-ext-object`. Why is that, and how would you fix that?

What we have done is capture the essence of a *class*. Each function parameterized over a parent is...well, it's a bit tricky, really. Let's call it a *blob* for now. A blob corresponds to what a Java programmer defines when they write a `class`:

```
class NodeSize extends Node { ... }
```

Do Now!

So why are we going out of the way to not call it a “class”?

When a developer invokes a Java class’s constructor, it in effect constructs objects all the way up the inheritance chain (in practice, a compiler might optimize this to require only one constructor invocation and one object allocation). These are private copies of the objects corresponding to the parent classes (private, that is, up to the presence of static members). There is, however, a question of how much of these objects is visible. Java chooses that—unlike in our implementation above—only one method of a given name (and signature) remains, no matter how many there might have been on the inheritance chain, whereas every field remains in the result, and can be accessed by casting. The latter makes some sense because each field presumably has invariants governing it, so keeping them separate (and hence all present) is wise. In contrast, it is easy to imagine an implementation that also makes all the methods available, not only the ones lowest (i.e., most refined) in the inheritance hierarchy. Many scripting languages take the latter approach.

Exercise

In the implementation above, we have relied on the self-application semantics for recursive access to an object, rather than using state. The reason is because the behavior of inheritance would be subtly wrong if we used state naively, as we have shown above. Can you construct an example that illustrates this?

By examining values carefully, you will notice that the `self` reference is to the most refined object at all times. This demonstrates the other form of extensibility we get from traditional objects: *extensible recursion*. The extensible conditional can be viewed as free extension across “space”, namely, the different variants of data, whereas extensible recursion can be viewed as free extension across “time”, namely, the different extensions to the code. Nevertheless, as this paper points out, there’s no free lunch.

32.4.2 Prototypes

In our description above, we’ve supplied each class with a description of its parent *class*. Object construction then makes instances of each as it goes up the inheritance chain. There is another way to think of the parent: not as a class to be instantiated but, instead, directly as an object itself. Then all children with the same parent would observe the very same object, which means changes to it from one child object would be visible to another child. The shared parent object is known as a *prototype*.

The archetypal prototype-based language is Self. Though you may have read that languages like JavaScript are "based on" Self, there is value to studying the idea from its source, especially because Self presents these ideas in their purest form.

Some language designers have argued that prototypes are more primitive than classes in that, with other basic mechanisms such as functions, one can recover classes from prototypes—but not the other way around. That is essentially what we have done above: each "class" function contains inside it an object description, so a class is an object-returning-function. Had we exposed these are two different operations and chosen to inherit directly from an object, we would have something akin to prototypes.

Exercise

Modify the inheritance pattern above to implement a Self-like, prototype-based language, instead of a class-based language. Because classes provide each object with distinct copies of their parent objects, a prototype-language might provide a *clone* operation to simplify creation of the operation that simulates classes atop prototypes.

32.4.3 Multiple Inheritance

Now you might ask, why is there only one fall-through option? It's easy to generalize this to there being many, which leads naturally to *multiple inheritance*. In effect, we have multiple objects to which we can chain the lookup, which of course raises the question of what order in which we should do so. It would be bad enough if the ascendants were arranged in a tree, because even a tree does not have a canonical order of traversal: take just breadth-first and depth-first traversal, for instance (each of which has compelling uses). Worse, suppose a blob `A` extends `B` and `C`; but now suppose `B` and `C` each extend `D`. Now we have to confront this question: will there be one or two `D` objects in the instance of `A`? Having only one saves space and might interact better with our expectations, but then, will we visit this object once or twice? Visiting it twice should not make any difference, so it seems unnecessary. But visiting it once means the behavior of one of `B` or `C` might change. And so on. As a result, virtually every multiple-inheritance language is accompanied by a subtle algorithm merely to define the lookup order—and each language's designer argues why their algorithm is more intuitive.

This infamous situation is called *diamond inheritance*. If you choose to include multiple inheritance in your language you can lose yourself for days in design decisions on this. Because it is highly unlikely you will find a canonical answer, your pain will have only begun.

Multiple inheritance is only attractive until you've thought it through.

32.4.4 Super-Duper!

Many languages have a notion of super-invocations, i.e., the ability to invoke a method or access a field higher up in the inheritance chain. This includes doing so at the point of object construction, where there is often a requirement that all constructors be invoked, to make sure the object is properly defined.

Note that we say "the" and "chain". When we switch to multiple inheritance, these concepts are replaced with something much more complex.

We have become so accustomed to thinking of these calls as going "up" the chain that we may have forgotten to ask whether this is the most natural direction. Keep in mind that constructors and methods are expected to enforce *invariants*. Whom should we trust more: the super-class or the sub-class? One argument would say that the sub-class is most refined, so it has the most global view of the object. Conversely, each super-class has a vested interest in protecting its invariants against violation by ignorant sub-classes.

These are two fundamentally opposed views of what inheritance means. Going up the chain means we view the extension as *replacing* the parent. Going down the chain means we view the extension as *refining* the parent. Because we normally associate sub-classing with refinement, why do our languages choose the "wrong" order of calling? Some languages have, therefore, explored invocation in the downward direction by default.

gbeta is a modern programming language that supports `inner`, as well as many other interesting features. It is also interesting to consider combining both directions.

32.4.5 Mixins and Traits

Let's return to our "blobs".

When we write a `class` in Java, what are we really defining between the opening and closing braces? It is not the entire class: that depends on the parent that it extends, and so on recursively. Rather, what we define inside the braces is a *class extension*. It only becomes a full-blown class because we *also* identify the parent class in the same place.

Naturally, we should ask: Why? Why not separate the act of *defining an extension* from *applying the extension to a base class*? That is, suppose instead of

```
class C extends B { ... }
```

we instead write:

```
classext E { ... }
```

and separately

```
class C = E(B);
```

where `B` is some already-defined class.

This recovers what we had before, but the function-application-like syntax is meant to be suggestive: we can "apply" this extension to several different base classes. Thus:

```
class C1 = E(B1);
class C2 = E(B2);
```

and so on. What we have done by separating the definition of `E` from that of the class it extends is to *liberate class extensions from the tyranny of the fixed base class*. We have a name for these extensions: they're called *mixins*.

The term "mixin" originated in Common Lisp, where it was a particular pattern of using multiple inheritance. Lipstick on a pig.

Mixins make class definition more compositional. They provide many of the benefits of multiple-inheritance (reusing multiple fragments of functionality) but within the aegis of a single-inheritance language (i.e., no complicated rules about lookup order). Observe that when desugaring, it's actually quite easy to add mixins to the language. A mixin is primarily a "function over classes"; because we have already determined how to desugar classes, and our target language for desugaring also has functions, and classes desugar to expressions that can be nested inside functions, it becomes almost trivial to implement a simple model of mixins.

This is a case where the greater generality of the target language of desugaring can lead us to a *better* construct, if we reflect it back into the source language.

In a typed language, a good design for mixins can actually improve object-oriented programming practice. Suppose we're defining a mixin-based version of Java. If a mixin is effectively a class-to-class function, what is the "type" of this "function"? Clearly, a mixin ought to use *interfaces* to describe what it expects and provides. Java already enables (but does not require) the latter, but it does not enable the former: a class (extension) extends another *class*—with all its members visible to the extension—not its *interface*. That means it obtains all of the parent's behavior, not a specification thereof. In turn, if the parent changes, the class might break.

In a typed mixin language, we can instead write

```
mixin M extends I { ... }
```

where `I` is an interface. Then `M` can only be applied to a class that satisfies the interface `I`, and in turn the language can *ensure that only members specified in `I` are visible in `M`*. This follows one of the important principles of good software design.

"Program to an interface, not an implementation." —*Design Patterns*

A good design for mixins can go even further. A class can only be used once in an inheritance chain, by definition (if a class eventually referred back to itself, there would be a cycle in the inheritance chain, causing potential infinite loops). In contrast, when we compose functions, we have no qualms about using the same function twice (e.g.: `(map ... (filter ... (map ...)))`). Is there value to using a mixin twice?

There certainly is! See sections 3 and 4 of *Classes and Mixins*.

Mixins solve an important problem that arises in the design of libraries. Suppose we have a dozen features that can be combined in different ways. How many classes should we provide? It is obviously impractical to generate the entire combinatorial explosion of classes. It would be better if the devleoper could pick and choose the features they care about. This is precisely the problem that mixins solve: they provide class extensions that the developers can combine, in an interface-preserving way, to create just the classes they need.

Mixins are used extensively in the Racket GUI library. For instance, `color:text-mixin` consumes basic text editor interfaces and implements the colored text editor interface. The latter is iself a basic text editor interface, so additional basic text mixins can be applied to the result.

Exercise

How does your favorite object-oriented library solve this problem?

Mixins do have one limitation: they enforce a linearity of composition. This strictness is sometimes misplaced, because it puts a burden on programmers that may not be necessary. A generalization of mixins called *traits* says that instead of extending a single mixin, we can extend a *set* of them. Of course, the moment we extend more than one, we must again contend with potential name-clashes. Thus traits must be equipped with mechanisms for resolving name clashes, often in the form of some name-combination algebra. Traits thus offer a nice complement to mixins, enabling programmers to choose the mechanism that best fits their needs. A handful of languages, such as Racket, therefore provide both traits and mixins.

32.5 Object Classification and Object Equality

Previously [section 21.6], we have seen three different kinds of equality operations. For the purpose of this discussion, we will ignore the distinction between `equal-now` and `equal-always`, focusing on the fact that both are primarily structural (`equal-now` being purely so). Extended to objects, this would check each member recursively, perhaps ignoring methods in languages that cannot compare them for equality, or comparing them using reference equality.

This leaves us with the very fine-grained and unforgiving `identical`, and the very coarse-grained and perhaps overly forgiving `equal-now`. Why is structural equality overly forgiving? Because two completely unrelated objects that just happened to have the same member names and types could end up being regarded equal: as a famous example in the objects community has it, `draw` is a meaningful method of both user interfaces and cowhands.

Therefore, some systems provide an equality predicate "in the middle": it is still fundamentally structural, but it discriminates between objects that were not "made the same way". The typical notion of construction is associated with a class: all objects made from a certain class are considered to be candidates for (structural) equality, but objects made from different classes (for some notion of "different") are immediately ruled unequal independent of their structure (which may in fact be identical).

In the special case where classes are named, first-order entities, this is called *nominal* equality: an equality based on names. However, it does not have to depend on names, nor even on first-order classes. Some languages have dynamic tag creators—known to the language—called *brands*. Each branding operation places a tag on an object. The built-in equality primitives then check for brands being

In keeping with the cowhand theme.

identical; when this condition is met, they revert to structural equality (which may involve additional brand-checking during recursion).

32.6 Types for Objects

Having studied various programming mechanisms, we now turn our focus to types for them. First [section 32.6.1] we will relax the notion of invariance for substitutability [section 27.2]. Then, we will discuss how new notions of equality [section 32.5] can impact subtyping to create a new class of types [section 32.6.3].

32.6.1 Subtyping

Consider two object types. The first we will call `Add1Sub1`:

```
type Add1Sub1 =
  { add1 :: (Number -> Number),
    sub1 :: (Number -> Number) }
```

This is a type for objects that have two members, `add1` and `sub1`, of the given types. The question we need to answer is, precisely what objects can be given this type?

To understand this, let us consider another, related type, which we will call `Arith`:

```
type Arith =
  { add1  :: (Number -> Number),
    sub1  :: (Number -> Number),
    plus  :: (Number, Number -> Number),
    mult  :: (Number, Number -> Number) }
```

Notice that two members have the same name and the same type, but there are two more members (`plus` and `mult`).

Consider a function designed to work with `Arith` values:

```
fun f(a :: Arith) -> Number:
  a.plus(2, 3)
end
```

Is it okay to pass a value of type `Add1Sub1` to `f`? Of course not: the function invokes the member `plus`, which the type annotation on `a` says it can expect to find; but if the value passed in does not have this member, this would result in a run-time *member not found* error, which is precisely what the type system was trying to avoid. Therefore, we cannot *substitute* a value of type `Add1Sub1` in a context expecting a `Arith`.

But how about in the other direction? This is entirely reasonable: the context is expecting a `Add1Sub`—and hence not using any more than what that type promises. Because `Arith` supplies everything expected by `Add1Sub1`, it is okay to provide a `Arith` value for a `Add1Sub1`.

This is our first example of *subtyping*. We say that `Arith` is a subtype of `Add1Sub1` because we can supply an `Arith` value in any context that expected a `Add1Sub1` value. Specifically, because this involves dropping some members—i.e., making the object "less wide"—this is called *width subtyping*.

The essence of subtyping is a relation, conventionally written as `<:`, between pairs of types. We say `S <: T` if a value of type `S` can be given where a value of type `T` is expected, and call `S` the *subtype* and `T` the *supertype*. Therefore, in the above example, `Arith <: Add1Sub1` and `Arith` is a subtype of `Add1Sub1`. It is useful (and usually accurate) to take a subset interpretation: if the values of `S` are a subset of `T`, then an expression expecting `T` values will not be unpleasantly surprised to receive only `S` values.

Later [section 32.6.3], we will talk about how sub*types* correspond to sub*classes*. But for now observe that we're talking only about objects, without any reference to the existence of classes.

Exercise

Why is subtyping a relation and not a function?

In other words:

```
{ add1  : (Number -> Number),              { add1 : (Number -> Number),
  sub1  : (Number -> Number),            <:  sub1 : (Number -> Number) }
  plus  : (Number, Number -> Number),
  mult  : (Number, Number -> Number) }
```

This may momentarily look confusing: we've said that subtyping follows set inclusion, so we would expect the smaller set on the left and the larger set on the right. Yet, it looks like we have a "larger type" (certainly in terms of character count) on the left and a "smaller type" on the right.

To understand why this is sound, it helps to develop the intuition that the "larger" the type, the fewer values it can have. Every object that has the four members on the left clearly also has the two members on the right. However, there are many objects that have the two members on the right that fail to have all four on the left. If we think of a type as a constraint on acceptable value shapes, the "bigger" type imposes more constraints and hence admits fewer values. Thus, though the *types* may appear to be of the wrong sizes, everything is well because the sets of values they subscribe are of the expected sizes.

As you might expect, there is another important form of subtyping, which is *within* a given member. This simply says that any particular member can be subsumed to a supertype in its corresponding position. For obvious reasons, this form is called *depth subtyping*.

Exercise

Construct two examples of depth subtyping. In one, give the field itself an object type, and use width subtyping to subtype that field. In the other, give the field a function type.

The combination of width and depth subtyping cover the most interesting cases of object subtyping. A type system that implemented only these two would, however, needlessly annoy programmers. Other convenient rules include the ability to permute names, reflexivity (every type is a subtype of itself, which gives us invariance for free, and lets us interpret the subtype relationship as subset), and transitivity.

Subtyping has a pervasive effect on the type system. We have to reexamine every kind of type and understand its interaction with subtyping. For base types, this is usually quite obvious: disjoint types like `Number`, `String`, etc., are all unrelated to each other. (In languages where one base type is used to represent another—for instance, in some scripting languages numbers are merely strings written with a special syntax, and in other languages, Booleans are merely numbers—there might be subtyping relationships even between base types, but these are not common.) However, we do have to consider how subtyping interacts with every single compound type constructor.

In fact, even our very diction about types has to change. Suppose we have an expression of type `T`. Normally, we would say that it produces values of type `T`. Now, we should be careful to say that it produces values of *up to* or *at most* `T`, because it may only produce values of a subtype of `T`. Thus every reference to a type should implicitly be cloaked in a reference to the potential for subtyping. To avoid pestering you we will refrain from doing this, but be wary that it is possible to make reasoning errors by not keeping this implicit interpretation in mind.

Subtyping Functions

Our examples above have been carefully chosen to mask an important detail: the subtyping of functions. To understand this, we will build up an example.

Consider a hypothetical language with a new type called `Boolean01`, where `true` is just an alias for `1` and `false` is an alias for `0`. Thus, in this language, `Boolean01 <: Number`; all `Boolean01` values are of type `Number`, but not all `Number` values are `Boolean01` (indeed, most are not). In this language, we can write some functions:

```
fun b2n(b :: Boolean01) -> Number:
  if b == 0:  # alias for false
    1
  else if b == 1:  # alias for true
    0
  else:
    raise('not valid number as Boolean01')
  end
end

fun n2b(n :: Number) -> Boolean01:
  if n == 0:
    false  # alias for 0
  else if n == 1:
    true   # alias for 1
  else:
    raise('no valid Boolean01 for number')
  end
end

fun n2n(n :: Number) -> Number:
  n + 1
end

fun b2b(b :: Boolean01) -> Boolean01:
  if b == 0:  # alias for false
    true   # alias for 1
  else if b == 1: # alias for true
    false  # alias 0
  else:
    raise('not valid number as Boolean01')
  end
end
```

Let us also define four types:

```
type N2N = (Number -> Number)
type B2B = (Boolean01 -> Boolean01)
type N2B = (Number -> Boolean01)
type B2N = (Boolean01 -> Number)
```

We can now ask the following question: which of these types are subtypes of the other? In more concrete terms, which of these functions can be safely substituted for which others?

We might expect a rule as follows. Because `Boolean01 <: Number` (in our imaginary system), a `(Boolean01 -> Boolean01)` function is a subtype of a `(Number -> Number)` function. This is a natural conclusion to arrive at...but wrong, as we will soon see.

To make this concrete, assume we have a function `p` that consumes and uses one of these functions. The function could be a member in an object, though for the purposes of understanding the basic problem, we don't need that: we can focus just on the function types. Thus, we have something like

```
fun p(op :: (A -> B)) -> B:
  op(a-value)
end
```

where `A` and `B` are going to be all the combinations of `Number` and `Boolean01`, and assume that `a-value` has whatever the `A` type is. For each type for `op` (column headers), we will ask which of the above functions (row headers) we can safely pass to `p`.

Do Now!

Stop and try to fill out this table first.

	N2N	**N2B**	**B2N**	**B2B**
n2n	yes (identical)	no (range)	yes (domain)	no (range)
n2b	yes (range)	yes (identical)	yes (domain and range)	yes (domain)
b2n	no (domain)	no (domain and range)	yes (identical)	no (range)
b2b	no (domain)	no (domain)	yes (range)	yes (identical)

In each cell, "yes" means the function on the left can be passed in when the type at the top is expected, while "no" means it cannot. Parentheses give the reason: "identical" means they are the same type (so of course they can be passed in); in the "yes" case it says where subtyping needed to apply, while in the "no" case where the type error is.

Let us consider trying to pass `n2n` to a `N2B` annotation (for `op`). Because the return type of `p` is `Boolean01`, whatever uses `p(n2n)` assumes that it gets only `Boolean01` values back. However, the function `n2n` is free to return any numeric value it wants: in particular, given `1` it returns `2`, which does not correspond to either `Boolean01`. Therefore, allowing this parameter can result in an unsound program execution. To prevent that, we must flag this as a type error.

More generally, if the type of the emph formal parameter promises `Boolean01`, the actual function passed had better return only `Boolean01`; but if the type of the formal is `Number`, the actual can safely return `Boolean01` without causing trouble. Thus, in general, for `(A -> B) <: (C -> D)`, we must have that `B <: D`. In other words, the subtyping of the range parallels the subtyping of the function itself, so we say the range position is *covariant* ("co-" meaning "together").

Now we get to the more interesting case: the domain. Consider why we can pass `n2n` where a `B2N` is expected. Inside the body of `op`, `a-value` can only be a `Boolean01`, because that is all the type permits. Because every `Boolean01` is a `Number`, the function `n2n` has no trouble accepting it.

In contrast, consider passing `b2n` where an `N2N` is expected. Inside `op`, `a-value` can evaluate to any number, because `op` is expected (by the type annotation on `p`) to be able to accept it. However, `b2n` can accept only two numbers; everything else results in an error. Hence, if the type-checker were to allow this, we could get a run-time error even though the program passed the type-checker.

From this, the moral we derive is that for the domain position, the *formal must be a subtype of the actual*. The formal parameter bounds what values `op` can expect; so long as the actual can take a set of values at least as large, there will be no problem. Thus, for `(A -> B) <: (C -> D)`, we must have that `C <: A`. The subtyping of the domain goes in the direction opposite to that of the subtyping of the function itself, so we say the range position is *contravariant* ("contra-" meaning "opposite").

Putting together these two rules, `(A -> B) <: (C -> D)` when `C <: A` and `B <: D`.

Subtyping and Information Hiding

Consider an object `o` that implements the `Add1Sub1` type. By the nature of width subtyping, there is absolutely nothing preventing the object from also having members named `plus` and `mult` of the right type. This raises a question: if we write

```
o :: Add1Sub1 = ...
```

where `...` is an object with `plus` and `mult`, and then we attempt to pass `o` to `f`, what should happen?

In a strictly dynamic interpretation—e.g., if this program were written without any annotations at all—it would work perfectly fine. Doing so statically, however, means that we have violated our intuition about what these annotations mean. In general, it is going to be undecidable whether an object of type `Add1Sub1` actually has the additional `Arith` members in it, so it is safest to reject programs that attempt to use `Add1Sub1`-annotated values as `Arith` ones. The natural type system will prevent us from passing `o` to `f`.

In short, the static type system becomes a mechanism for *information hiding*. By leaving out some members in type descriptions, we effectively hide the fact that they exist. For instance, one could create an object

```
crypto =
  { private-key: ... ,
    public-key: ...,
    decrypt: fun(msg): ... end,
    encrypt: fun(plain-text): ... end }
```

and ascribe it the type

```
type PK =
  { public-key: Number,
    encrypt: (String -> String) }
```

as follows:

```
for-dist :: PK = crypto
```

Then all references to `for-dist` can only use the public interface and have no way to access the `private-key` or `decrypt` members, but those that have access to the `crypto` object can use those members. Provided access to `crypto` is provisioned carefully, the *language* will ensure the privacy of these two sensitive members.

However, this becomes more tricky in a system that is not purely statically typed, including ones where a typed language can interoperate with an untyped one. In an untyped language there are no annotations, so there is nothing preventing the `plus` member of `o` or the `decrypt` member of `for-dist` from being accessed: after all, those members really are present in the underlying object. Thus, it is common when "exporting" a typed object to any kind of untrusted or unannotated context to create a *proxy* object; it would be as if the developer wrote the following by hand:

```
fun proxy-for-crypto(c):
  { public-key: c.public-key,
    encrypt: c.encrypt }
end

proxy-dist = proxy-for-crypto(for-dist)
```

It is `proxy-dist` that is then provided to dangerous contexts. Since the resulting object really contains only two fields, and the underlying object is only visible in lexical scope (`c`), so long as the language does not provide a means to inspect or traverse the scope (an assumption not guaranteed by all languages!), the untyped or dangerous context cannot get access to the private content.

Implementing Subtyping

Of course, these rules assume that we have modified the type-checker to respect subtyping. The essence of subtyping is a rule that says, if an expression `e` is of type `S`, and `S <: T`, then `e` also has type `T`. While this sounds intuitive, it is also immediately problematic for two reasons:

- Until now all of our type rules have been syntax-driven, which is what enabled us to write a recursive-descent type-checker. Now, however, we have a rule that applies to *all* expressions, so we can no longer be sure when to apply it.

- There could be many levels of subtyping. As a result, it is no longer obvious when to "stop" subtyping. In particular, whereas before type-checking was able to calculate the type of an expression, now we have many possible types for each expression; if we return the "wrong" one, we might get a type error (due to that not being the type expected by the context) even though there exists some other type that was the one expected by the context.

What these two issues point to is that the description of subtyping we are giving here is fundamentally *declarative*: we are saying what must be true, but not showing how to turn it into an algorithm. For each actual type language, there is a less or more interesting problem in turning this into *algorithmic subtyping*: an actual algorithm that realizes a type-checker (ideally one that types exactly those programs that would have typed under the declarative regime, i.e., one that is both sound and complete).

32.6.2 Types for Self-Reference

Remember that one of the essential features of many object systems is having a reference, inside a method, to the object on which it was invoked: i.e., a self-reference [section 32.2.6]. What is the type of this `self` identifier?

Consider the type `Add1Sub1` we described earlier. To be entirely honest, the implementation of `add1` and `sub1`—to be *methods*—must take an extra parameter that will be a self-reference. What is the nature of this self-referential parameter? It is clearly an object; it clearly has two methods, `add1` and `sub1` (at least up to subtyping); and each of those methods takes two parameters, one a number and...

You see where this is going.

Object types are therefore typically *recursive* types: the type world's equivalent of `rec` [section 21.3.2]. Typically, they are written μ ("mu") instead of `rec`; thus:

```
type Add1Sub1 =
  μ T . { add1 :: (T, Number -> Number),
          sub1 :: (T, Number -> Number) }
```

Read the right-hand side as "construct a recursive type `T` such that it (a) is an object, (b) has two members `add1` and `sub1`, and (c) each member has two parameters, the first of which is the type being defined" (and so on).

Unfortunately, recursive types are not as simple as they look. Note that the above type does not have a "base case"; thus, it is a finite representation of an infinite type (which is exactly what we want, because we can write an infinite number of `self` applications). Therefore, when it comes to checking for the equality of two recursive types, we encounter complications, which are beyond the scope of this study.

See Pierce's *Types and Programming Languages* for details.

32.6.3 Nominal Types

Earlier [section 32.5] we read about nominal equality, where classes are made to aid in equality comparisons. In some typed languages—Java being a poster-child—classes carry an even heavier load: they are also used as the basis for the type system, rather than structural types.

The basic idea is that each class (or other nominal entity) defines an entirely new type, even if the type-structure of its members is exactly the same as that of some other type. Then, type equality mirrors nominal equality, but trivially: if two values have the same type they must have the same structure, and if they have different types then their structure doesn't matter (even if it's identical). Thus, type equality reduces to a constant-time check of whether the classes are the same.

Nominal types have one more advantage. They effectively make it straightforward to write recursive types without wrestling with μ. Consider the following Java class definition:

```
class Add1Sub1 {
  public int add1(int n) { ... }
  public int sub1(int n) { ... }
}
```

Implicit in these two method definitions are `this` parameters. But what is the type of `this`? It's just `Add1Sub1`: the keyword `class` not only introduces a new name but automatically makes it a recursive binding. Thus, programmers can comfortably refer to and use nominal types without having to dwell on their true meaning (as recursive types) or their equality (because it's by name rather than structure). Thus, nominal types, for all their inflexibility, do offer an elegant solution to a particular set of language design constraints.

It is worth noting that in Java, inheritance (unfortunately) corresponds to subtyping. As we go up the inheritance chain a class has fewer and fewer members (width subtyping), until we reach `Object`, the supertype of all classes, which has the fewest. Thus for all class types `C` in Java, `C <: Object`. The interpretation of subtyping as subsets holds: every object that has a type lower in an inheritance hierarchy also has a type higher in the hierarchy, but not vice versa. When it comes to depth subtyping, however, Java prefers types to be *invariant* down the object hierarchy because this is a safe option for conventional mutation.

Somewhat confusingly, the terms *narrowing* and *widening* are sometimes used, but with what some might consider the opposite meaning. To widen is to go from subtype to supertype, because it goes from a "narrower" (smaller) to a "wider" (bigger) set. These terms evolved independently, but unfortunately not consistently.

Chapter 33

Control Operations

The term *control* refers to any programming language instruction that causes evaluation to proceed, because it "controls" the program counter of the machine. In that sense, sequential execution of instructions is "control", as is even an arithmetic expression (and in the presence of state, this control is laid bare through the order in which effects occur); other forms of control found in all ordinary programming languages include function calls and returns. However, in practice we use the term to refer primarily to those operations that cause *non-local* transfer of control beyond that of mere functions and procedures, usually starting with exceptions. We will study such operations in this chapter.

As we study the following control operators, it's worth remembering that even without them, we still have languages that are Turing-complete, so these control operations provide no more "power". Therefore, what control operators do is change and potentially improve the way we express our intent, and therefore enhance the structure of programs. Thus, it pays to being our study by focusing on program structure.

33.1 Control on the Web

Let us begin our study by examining the structure of Web programs. Consider the following program:

```
print(read-number("First number")
  + read-number("Second number"))
```

Henceforth, we'll call this our "addition server". You should, of course, understand this as a stand-in for more sophisticated applications. For instance, the two prompts might ask for starting and ending points for a trip, and in place of addition we might compute a route or compute airfares. There might even be computation between the two steps: e.g., after entering the first city, the airline might prompt us with choices of where it flies from there.

This is based on a hypothetical a `read-number` function that, when run, suspends program execution, prompts for a number and, when the user enters one, resumes computation. You might find it excessively pedantic that I'd mention the suspension and resumption of computation, but this detail will prove to be absolutely central to our study, so don't gloss over these steps!

Now suppose we want to run this on a Web server. We immediately encounter a difficulty: the structure of server-side Web programs is such that they generate a single Web page—such as the one asking for the first number—and then *halt*. As a result, the *rest of the program*—which in this case prompts for the second number, then adds the two, and then prints that result, is lost.

Do Now!

Why do Web servers behave in such a strange way?

There are at least two reasons for this behavior: one perhaps historical, and the other technical. The historical reason is that Web servers were initially designed to serve *pages*, i.e., static content. Any program that ran had to generate its output to a file, from which a server could offer it. Naturally, developers wondered why that same program couldn't run on demand. This made Web content *dynamic*. Terminating the program after generating a single piece of output was the simplest incremental step in transitioning the Web from "pages" to "programs".

The more important reason—and the one that has stayed with us—is technical. Imagine our addition server has generated its first prompt. The pending computation is not trivial: it must remember the first response, generate the second prompt, perform the addition, and then display the result. This computation must suspend waiting for the user's input. If there are millions of users, then millions of computations must be suspended (imagine threads running in virtual machines, each consuming memory for local data), creating an enormous performance problem. Furthermore, suppose a user does not actually complete the computation—analogous to searching at an on-line bookstore or airline site, but not completing the purchase. How does the server know when or even whether to terminate the computation? Until it does, the resources associated with that computation remain in use.

Conceptually, therefore, the Web protocol was designed to be *stateless*: it would not store state on the server associated with intermediate computations. Instead, Web program developers would be forced to maintain all necessary state elsewhere, and each request would need to be able to resume the computation in full. In practice the Web has not proven to be stateless, but it still hews in this direction, and studying the structure of such programs is very instructive.

Now consider client-side Web programs: those that run inside the browser, written in or compiled to JavaScript. Suppose such a computation needs to communicate with a server. The primitive for this is called `XMLHttpRequest`. The user makes an instance of this primitive and invokes its `send` method to send a message to the server.

Communicating with a server is not, however, instantaneous: it takes some time; if the server faces a heavy load, it could take a long time; and indeed, it may never complete at all, depending on the state of the network and the server. (These are the same problems faced above by `get-number`, with a user taking the place of a server: the user may take a long time to enter a number, or may never do so at all.) If the `send` method suspended program execution, the entire (client-side) application would be blocked, indefinitely. You would not want to use such a program.

To keep the application responsive, the designers of `XMLHttpRequest` therefore had a choice. They could make JavaScript multi-threaded, but because the language also has state, programmers would have to confront all the problems of combining state with concurrency. In particular, beginners would have to wrestle with a combination of features that even experienced programmers do not use well, probably resulting in numerous deadlocked Web sites.

Instead, JavaScript is *single-threaded*: i.e., there is only one thread of execution at a time. When the `send` method is invoked, JavaScript instead suspends the current computation and returns control to an event loop, which can now invoke other suspended computations. Devlopers associate a *callback* with the `send`. When (and if) a response returns, this callback is added to the queue of suspended computations, thereby enabling it to resume.

Due to the structuring problems this causes, there are now various proposals to, in effect, add “safe” threads to JavaScript. The ideas described in this chapter can be viewed as an alternative that offer similar structuring benefits.

This callback needs to embody the *rest of the processing* of that request. Thus, for entirely different reasons—not performance, but avoiding the problems of synchronization, non-atomicity, and deadlocks—the client-side Web has evolved to impose essentially the same problems of program structure on developers as the server-side Web. Let us now better understand that structure.

33.1.1 Program Decomposition into Now and Later

Let us consider what it takes to make our addition program work in a stateless setting, such as on a Web server. First we have to determine the *first* interaction. This is the prompt for the first number, because Pyret evaluates arguments from left to right. It is instructive to divide the program into two parts: what happens to generate the first interaction (which can run right now), and what needs to happen after it (which must be “remembered” somehow). The former is easy:

```
read-number("First number")
```

We’ve already explained in prose what’s left, but now it’s time to write it *as a program*. It seems to be something like:

```
print(<the result from the first interaction>
  + read-number("Second number"))
```

A Web server can't execute the above, however, because it evidently isn't a *program*. We instead need some way of writing this as one.

Let's observe a few characteristics of this computation:

- It needs to be a syntactically valid program.
- It needs to stay suspended until the request comes in.
- It needs a way—such as a parameter—to refer to the value from the first interaction.

Put together these characteristics and we have a clear representation—a *function*:

```
lam(v1):
  print(v1 + read-number("Second number"))
end
```

33.1.2 A Partial Solution

On the Web, there is an additional wrinkle: each Web page with input elements needs to refer to a program stored on the Web, which will receive the data from the form and process it. This program is named in the `action` field of a form. Thus, imagine that the server generates a fresh label, stores the above function in a table associated with that label, and refers to the label in the `action` field. When (and if) the client actually submits the form the server extracts the associated function, supplies it with the form's values, and thus resumes execution.

Do Now!

Is the solution above stateless?

Let's imagine that we have a custom Web server that maintains the above table. In such a server, we might have a special version of `read-number`—call it `read-number-suspend`—that records the rest of the program:

```
read-number-suspend("First number",
  lam(v1):
    print(v1 + read-number("Second number"))
  end)
```

Unfortunately, this is not sufficient. The moment we perform the *second* `read-number`, we're back to having forgotten the rest of the computation. Therefore, the second one needs to be converted to use `read-number-suspend`, too. What is the rest of *its* computation?

```
lam(v2):
  print(v1 + v2)
end
```

where `v1` is the value from the first computation. Putting together the pieces, the fully-translated program is

```
read-number-suspend("First number",
  lam(v1):
    read-number-suspend("Second number",
      lam(v2):
        print(v1 + v2)
      end)
  end)
```

Notice how the inner closure depends on being nested inside the outer one, so that `v1` is bound in the addition. Also observe how the addition and printing got moved from initiating "immediately" after the first number was provided to waiting until the second number was also available.

Exercise

Ascribe types to the above computation. Also determine the type of the Web server and of the table holding these procedures.

33.1.3 Achieving Statelessness

We haven't actually achieved statelessness yet, because we have this large table residing on the server, with no clear means to remove entries from it. It would be better if we could avoid the server state entirely. This means we have to move the relevant state to the client.

There are actually two ways in which the server holds state. One is that we have reserved the right to create as many entries in the hash table as we wish. This makes the server storage space proportional to the number of interactions—a dynamic value—rather than the size of the program, a static value with a clear bound. The other is what we're storing in the table: honest-to-goodness closures, each of which might be different and closed over copious amounts of state.

Let's start by eliminating the closure. Instead, let's have each of the functions be named and at the top-level (which immediately forces us to have only a fixed number of them, bounded by the size of the program):

```
read-number-stateless("First number", prog-1)

fun prog-1(v1):
  read-number-stateless("Second number", prog-2)
end

fun prog-2(v2):
  print(v1 + v2)
end
```

Observe how each code block refers only to the *name* of the next procedure, rather than to a real closure. The value of the argument comes from the form. There's just one problem: `v1` in `prog-2` is a free identifier!

The way to fix this problem is, instead of creating a closure after one step, to send `v1` to the client to be stored there. Where do we store this? The browser offers two mechanisms for doing this: *cookies* and *hidden fields*. Which one do we use?

33.1.4 Interaction with State

One way to avoid this problem is to find a channel of communication between what follows the first and second prompts. Recall that we have noted that *state* provides such a channel of communication [section 31.5]. Therefore, we could use a top-level variable to communicate the value of `v1`. To be suggestive, we'll call this variable `cookie`:

```
var cookie = "dummy initial value"

read-number-suspend("First number",
  lam(v1):
    cookie := v1
    read-number-suspend("Second number",
      lam(v2):
        print(cookie + v2)
      end)
  end)
```

from which we can eliminate closures easily:

```
var cookie = "dummy initial value"

read-number-stateless("First number", prog-1)

fun prog-1(v1):
  cookie := v1
  read-number-stateless("Second number", prog-2)
end

fun prog-2(v2):
  print(cookie + v2)
end
```

Unfortunately, this means *every* intermediate computation will share the same `cookie` variable. If we open up two concurrent windows and try to add different first numbers, the latest first number will always reside in `cookie`, so the other window is going to see unpredictable results.

This, of course, is precisely what happens on the Web. The browser's *cookies* are merely a client-side implementation of the store. Thus, Web sites that store their information in cookies are susceptible to exactly this problem: two concurrent interactions with the site will end up interfering with one another. Therefore, the pervasive use of cookies on Web sites, induced by Web programming traditions, results in actively less usable sites.

These problems are not hypothetical. For instance, see Section 2 of *Modeling Web Interactions and Errors*.

In contrast, the Web offers another mechanism for storing information on the client: the *hidden field*. Because they are local to each page, and each page corresponds to a closure, they are precisely analogous to a closure's environment! Thus, instead of storing the value of `v1` in a single, global cookie, if we were to store it in a hidden field in the response page, then two different response pages would have different values in their hidden field, which would be sent back to the server on the next request—thereby avoiding the interference problem entirely.

33.2 Conversion to Continuation-Passing Style

The style of functions we've been writing has a name. Though we've presented ideas in terms of the Web, we're relying on a much older idea: the functions are called *continuations*, and this style of programs is called *continuation-passing style* (CPS). This is worth studying in its own right, because it is the basis for studying a variety of other non-trivial control operations—such as generators.

We will take the liberty of using CPS as both a noun and verb: a particular structure of code and the process that converts code into it.

Earlier, we converted programs so that no Web input operation was nested inside another. The motivation was simple: when the program terminates, all nested computations are lost. A similar argument applies, in a more local sense, in the case of `XMLHttpRequest`: any computation depending on the result of a response from a Web server needs to reside in the callback associated with the request to the server.

In fact, we don't need to transform *every* expression. We only care about expressions that involve actual Web interaction. For example, if we computed a more complex mathematical expression than just addition, we wouldn't need to transform it. If, however, we had a function call, we'd either have to be absolutely certain the function didn't have any Web invocations either inside it, or in the functions in invokes, or the ones *they* invoke...or else, to be defensive, we should transform them all. Therefore, we have to transform *every expression that we can't be sure performs no Web interactions*.

The heart of our transformation is therefore to turn every function, `f`, into one with an extra argument. This extra argument is the continuation, which represents the *rest of the computation*. `f`, instead of *returning* a value, instead *passes* the value it would have returned to its continuation. Thus, the continuation is itself a function of one argument; this argument represents the value that *would have been returned* by `f`. A function returns a value to "pass it to the rest of the computation"; CPS makes this explicit, because invoking a continuation (in place of returning a value) precisely passes it to the function representing the rest of the computation.

CPS is a general transformation, which we can apply to any program. Because it's a program transformation, we can think of it as a special kind of desugaring that transforms programs *within* the same language: from the full language to a more restricted version that obeys the pattern we've been discussing. As a result, we can reuse an evaluator for the full language to also evaluate programs in the CPS subset.

33.2.1 Implementation by Desugaring

Let us therefore implement CPS as a source-to-source transformation. Thought of as a function, it consumes and returns `ExprC` expressions, but the output expressions will have the peculiar structure we have seen above, and will therefore be a strict subset of all `ExprC` expressions.

Exercise

Put differently, the comment about "strict subset" above means that certain `ExprC` expressions are not legal in the output that CPS generates. Provide examples.

<cps-trans> ::=

```
fun cps(e :: ExprC) -> ExprC:
  cases (ExprC) e:
    <cps-trans-numC>
    <cps-trans-plusC>
    <cps-trans-idC>
    <cps-trans-fdC>
    <cps-trans-appC>
  end
end
```

Our representation in CPS will be to turn *every* expression into a procedure of one argument, the continuation. The converted expression will eventually either supply a value to the continuation or will pass the continuation on to some other expression that will—by preserving this invariant inductively—supply it with a value. Applied to Pyret, all output from CPS will look like `fun (k): ... end`. Since we are applying CPS to Paret instead, it will look like `fdC("k", ...)`. Either way, note that lexical scope keeps these `k`'s from clashing with any other identifiers of the same name.

First let's dispatch with the easy case, which is atomic values. Because we already have a value, we are ready to "return" it, which we do by supplying it to the continuation:

<cps-trans-numC> ::=

```
  | numC(_) => fdC("k", appC(idC("k"), e))
```

and similarly:

<cps-trans-idC> ::=

```
  | idC(_) => fdC("k", appC(idC("k"), e))
```

Exercise

Extend the language to handle conditionals.

Exercise

Extend the language to support mutable state as well. Does this have any impact on the CPS process, i.e., does it change the pattern of conversion?

Next, let's handle binary operators. We have seen the essence of this transformation earlier, applied to the Web:

<cps-trans-plusC> ::=

```
  | plusC(l, r) =>
    fdC("k",
      appC(cps(l),
        fdC("l-v",
          appC(cps(r),
            fdC("r-v",
              appC(idC("k"), plusC(idC("l-v"), idC("r-v"))))))))
```

This assumes that the primitive operator, in this case addition, does not itself need to be transformed; on the Web, for instance, it's a safe bet that performing arithmetic does not involve any Web interactions.

Unless, of course, the arithmetic is part of a cryptographic algorithm, in which case it may be necessary to notify the NSA of the results.

Finally, we have function definition and application.

Do Now!

It's tempting to think that, because function are just values, they too can be passed unchanged to the continuation. Why is this not true?

Exercise

Before proceeding, alter the underlying language to also permit two-argument function definitions and, correspondingly, applications. Name the definitions `fd2C` and the applications `app2C`.

For an application we have to evaluate both the function and argument expressions. Once we've obtained these, we are ready to apply the function. Therefore, it is tempting to write

<cps-trans-appC-try-1> ::=

```
  | appC(f, a) =>
    fdC("k",
      appC(cps(f),
        fdC("f-v",
          appC(cps(a),
            fdC("a-v",
              appC(idC("k"), appC(idC("f-v"), idC("a-v"))))))))
```

Do Now!

Do you see why this is wrong?

The problem is that, though the function is a value, that value is a closure with a potentially complicated body: evaluating the body can, for example, result in further Web interactions, at which point the rest of the function's body, as well as the pending `k(...)` (i.e., the rest of the program), will all be lost. To avoid this,

we have to supply `k` to the function's value, and let the inductive invariant ensure that `k` will eventually be invoked with the value of applying `f-v` to `a-v`:
<cps-trans-appC> ::=

```
  | appC(f, a) =>
    fdC("k",
      appC(cps(f),
        fdC("f-v",
          appC(cps(a),
            fdC("a-v",
              app2C(idC("f-v"), idC("a-v"), idC("k")))))))
```

A function is itself a value, so it should be returned to the pending computation. The application case above, however, shows that we have to transform functions to take an extra argument, namely the continuation at the point of invocation. This leaves us with a quandary: which continuation do we supply to the body?

<cps-trans-fdC-try-1> ::=

```
  | fdC(v, b) =>
    fdC("k",
      appC(idC("k"),
        fd2C(v, "dyn-k",
          appC(cps(b), ???))))
```

That is, in place of `???`, which continuation do we supply: `k` or `dyn-k`?

Do Now!

Which continuation should we supply?

The former is the continuation *at the point of closure creation*. The latter is the continuation *at the point of closure invocation*. In other words, the former is "static" and the latter is "dynamic". In this case, we need to use the dynamic continuation, otherwise something very strange would happen: the program would return to the point where the closure was created, rather than where it is being used! This would result in seemingly very strange program behavior, so we wish to avoid it. Observe that we are consciously choosing the dynamic continuation just as, where scope was concerned we chose the static environment but where state was concerned we chose the "dynamic" (namely, most recent) store. Thus continuations are more like state than they are like lexical binding, a similarity we will return to later [REF].

<cps-trans-fdC> ::=

```
  | fdC(v, b) =>
    fdC("k",
      appC(idC("k"),
        fd2C(v, "dyn-k",
          appC(cps(b), idC("dyn-k")))))
```

Do Now!

After you have understood this material, replace `"dyn-k"` with `"k"`, predict what should change, and check that it does.

Testing any code converted to CPS is slightly annoying because all CPS terms expect a continuation. In a real environment, the initial continuation is one that simply either (a) consumes a value and returns it, or (b) consumes a value and prints it, or (c) consumes a value, prints it, and gets ready for another computation (as the prompt in a REPL does). All three of these are effectively just the identity function in various guises. Thus, the following definition is helpful for testing:

```
fun icps(e):
  id-cps = fdC("v", idC("v"))
  interp(appC(cps(e), id-cps), mt-env)
end
```

For instance,

```
icps(plusC(numC(5), appC(quad, numC(3)))) is numV(17)
icps(multC(appC(c5, numC(3)), numC(4))) is numV(20)
icps(plusC(numC(10), appC(c5, numC(10)))) is numV(15)
```

33.2.2 Understanding the Output

The output of this transformation takes some getting used to. Consider a very simple example:

```
cps(plusC(numC(1), numC(2)))
```

This evaluates to

```
fdC("k",
  appC(fdC("k",
      appC(idC("k"), numC(1))),
    fdC("l-v",
      appC(fdC("k",
          appC(idC("k"), numC(2))),
        fdC("r-v",
          appC(idC("k"),
            plusC(idC("l-v"), idC("r-v"))))))))
```

For (slightly more) readability, let's transform this from Paret to Pyret and give it a name:

```
f1 =
  lam(k):
    (lam(shadow k):
        k(1)
      end)(lam(l-v):
        (lam(shadow k):
              k(2)
          end)(lam(r-v):
              k(l-v + r-v)
          end)
      end)
  end
```

We can then apply it to the identity function to observe that it produces the expected answer:

We had to insert the `shadow` declarations to confirm to Pyret that we really did mean to shadow these identifiers.

```
check:
  f1(lam(x): x end) is 3
end
```

We can also rename the different `k`s to better tell them apart:

```
f2 =
  lam(k):
    (lam(k1):
        k1(1)
      end)(lam(l-v):
        (lam(k2):
            k2(2)
          end)(lam(r-v):
            k(l-v + r-v)
          end)
      end)
  end

check:
  f2(lam(x): x end) is 3
end
```

Terms like

```
(lam(k1): k1(1) end)(...)
```

would seem a significant decrease in code readability, but that's only until you learn how to "read" such a program. This is equivalent to saying: "`k1` represents the rest of the program after evaluating `1`. Evaluate `1`, and send its result—the value `1`—to the rest of the computation, namely `k1`." This value (`1`) is bound to `l-v`, the identifier representing the left-hand-side value of the addition...which, of course, is precisely what `1` is.

There is an active line of research in creating better CPS transformations that produce fewer intermediate function terms; we've actually used one of the very oldest and least sophisticated. The trade-off is in simplicity of desugaring versus simplicity of output, with the two roughly inversely correlated.

33.2.3 An Interaction Primitive by Transformation

At this point we have identified a problem in program structure; we hypothesized a better API for it; we transformed an example to use such an API; and then we generalized that transformation. But now we have a program structure so complex that it is unclear what use it could possibly be. The point of this transformation was so that *every* sub-expression would have an associated continuation, which a interaction-friendly primitive can use. Let's see how to do that.

To enable this, we will now add two primitives: `read-numC` and `read-num-webC`. The idea is that user programs (pre-CPS) will use the former, which the transformation will convert into uses of the latter. Here are the two new language forms:

```
| read-numC(p :: ExprC)
| read-num-webC(p :: ExprC, k :: ExprC)
```

The prompt `p` is assumed to be an expression that evaluates to what we want to print to the user. In our impoverished language this is a number, which is sufficient for illustration.

We will assume that `cps` does not need to handle `read-num-webC` (because the end-user is not expected to write this directly), while `interp` does not need to handle `read-numC` (because we want this interpreter to function even in a setting that periodically terminates input, so it cannot block waiting for a response).

Transforming `read-numC` into `read-num-webC` in `cps` is now easy, because the continuation argument now gives us exactly what we need:

```
| read-numC(p) =>
  fdC("k",
    appC(cps(p),
      fdC("p-v",
        read-num-webC(idC("p-v"), idC("k")))))
```

Now let us build an implementation of `read-num-webC` in the interpreter that properly simulates a program that halts.

First we need a way to record the current resumption point. In a real system this might be remembered on a server or marshaled into a value sent to the client. Here, we'll just record it in a global variable:

See this paper for more on how to marshal the continuation to the client.

```
var web-continuation = "nothing here yet"
```

Sure enough, something interesting will be there once we start running the program.

Now let us modify the interpreter:

```
| read-num-webC(p, k) =>
  prompt = num-to-string(interp(p, nv).n)
  cont = interp(k, nv)
  print('Web interaction: ' + prompt)
  web-continuation := cont
  raise('Program halted waiting for user input')
```

First we evaluate the prompt expression to obtain an actual prompt. We then print this to the screen. Crucially, we then store the current continuation to the global variable. Finally, we *halt the program's execution*; this step is vital in keeping us honest, so that we don't accidentally rely on Pyret to resume our computation.

Exercise

Introduce an error in `cps` and show how halting the program highlights it, while not doing so silently masks it.

Suppose we run this on the following input program:

```
››› icps(plusC(read-numC(numC(1)), read-numC(numC(2))))
```

Pyret prints

```
Error:

"Program halted waiting for user input"
```

and the program halts.

At this point, `web-continuation` contains a genuine, run-time closure (a `closV` value). This represents a continuation: a program value representing the

Due to a bug in the current implementation, you can't inspect the value of `web-continuation` directly; but you can access it from a function that closes over it.

rest of the computation. The user now supplies an input in the imagined Web form; this is provided as the actual argument to the continuation.

We can do this as follows. We extract the function and environment from the closure, and apply the function to the provided parameter, evaluating this in the context of the closure:

```
››› fun run-wc(n):
      wc = web-continuation
      interp(appC(wc.f, numC(n)), wc.e)
    end
```

Sure enough, using this with `3` as the first input yields:

```
››› run-wc(3)
```

```
Error:

"Program halted waiting for user input"
```

Oooh, promising! Now we try this again with, say, `4` as the second input, and we get:

```
››› run-wc(4)
```

```
numV(7)
```

Et voilà!

Here, then, is the key lesson. By transforming the program into CPS we were able to write a normal-looking program—`read-numC(numC(1)), read-numC(numC(2))`—and run it on an intepreter that truly terminated after each interaction, and were still able to resume the computation successfully, running to completion without losing track of computations or of scope errors. That is, we can write the program in *direct style*, with properly nested expressions, and a compiler—in this case, the CPS converter—takes care of making it work with a suitable underlying API. This is what good programming languages ought to do!

Exercise

Modify the program to store each previous continuations with some kind of unique tag. Now that you have access to multiple continuations, simulate the effect of different browser actions such as reloading the page (re-invoking a continuation), going back (using a prior continuation), cloning a page (re-using a continuation), etc. Does your implementation still work?

33.3 Implementation in the Core

Now that we've seen how CPS can be implemented through desugaring, we should ask whether it can be put in the core instead.

Recall that we've said that CPS applies to all programs. We have one program we are especially interested in: the interpreter. Sure enough, we can apply the CPS transformation to it, making available what are effectively the same continuations.

33.3.1 Converting the Interpreter

Rather than mindlessly applying the transformation, which would result in a very unwieldy (and unreadable) intepreter, we'll clean things up a little as we go. Note first of all that the interpreter needs to take an additional argument, representing the rest of the computation:

```
<cps-interp> ::=
  fun interp(e :: ExprC, nv :: List<Binding>, k):
    cases (ExprC) e:
      <cps-interp-numC>
      <cps-interp-plusC>
      <cps-interp-idC>
      <cps-interp-fdC/fd2C>
      <cps-interp-appC>
      <cps-interp-app2C>
    end
  end
```

Exercise

Note that we have not annotated `k`, and we've dropped the return annotation on `interp`. Fill them in.

When we have values, we simply "return" them through the continuation:

<cps-interp-numC> ::=

```
  | numC(n) =>
    k(numV(n))
```
<cps-interp-idC> ::=
```
  | idC(s) =>
    k(lookup(s, nv))
```
For binary operations where the operator is a primitive, we have to follow the CPS pattern:

<cps-interp-plusC> ::=
```
  | plusC(l, r) =>
    interp(l, nv,
      lam(l-v):
        interp(r, nv,
          lam(r-v):
            k(plus-v(l-v, r-v))
        end)
      end)
```
Note that CPS also ends up enforcing an order-of-evalation (in this case, left-to-right) just as mutation did.

For function definitions, we have to be careful. Earlier (*<cps-trans-fdC>*), we added a continuation parameter to closures. However, the `fdC` data structures *are merely data*; it is functions like `interp` that need to be given the extra parameter. Therefore, we can leave these alone:

<cps-interp-fdC/fd2C> ::=
```
  | fdC(_, _) =>
    k(closV(e, nv))
  | fd2C(_, _, _) =>
    k(closV(e, nv))
```
Finally, applications have to be converted to CPS as we have seen before:

<cps-interp-appC> ::=
```
  | appC(f, a) =>
    interp(f, nv,
      lam(clos-v):
        interp(a, nv,
          lam(arg-v):
            interp(clos-v.f.body,
              xtnd-env(bind(clos-v.f.arg, arg-v), clos-v.e),
              k)
          end)
      end)
```
and similarly when there are two arguments:

<cps-interp-app2C> ::=

```
  | app2C(f, a1, a2) =>
    interp(f, nv,
      lam(clos-v):
        interp(a1, nv,
          lam(arg1-v):
            interp(a2, nv,
              lam(arg2-v):
                interp(clos-v.f.body,
                  xtnd-env(bind(clos-v.f.arg1, arg1-v),
                    xtnd-env(bind(clos-v.f.arg2, arg2-v),
                      clos-v.e)),
                k)
              end)
          end)
      end)
```

By converting the interpreter to CPS we have given it access to an extra parameter: `k`, the continuation *of the interpreter*. Because the interpreter's execution mimics the intended behavior of the interpreted program, the continuation of the interpreter reflects the rest of the behavior of the interpreted program: i.e., applying `interp` to an expression `e` with continuation `k` will result in `k` being given the value of `e`. We can therefore put `k` to work by exposing it to programs being interpreted.

33.3.2 An Interaction Primitive in the Core

We can now lift our previous solution [section 33.2.3] to this modified interpreter. This time, instead of the continuation being created *by the program*, it's created *by the interpreter itself*, with the program oblivious to this activity. Thus:

```
| read-numC(p) =>
  interp(p, nv,
    lam(p-v):
      prompt = num-to-string(p-v.n)
      print('Web interaction: ' + prompt)
      web-continuation := k
      raise('Program halted waiting for user input')
    end)
```

Note that we must first evaluate the prompt expression to obtain its value. Now, however, there is no longer a continuation expression to evaluate: the interpreter has the continuation at the ready. It is this continuation that we store in `web-continuation`.

Observe that this value is now a genuine closure *in Pyret*, not a closure data structure we have constructed. Therefore, to apply it we can no longer extract its fields; instead, the only thing we can do with it is to apply it:

```
fun run-wc(n):
  web-continuation(numC(n))
end
```

With this in place, if we evaluate the expression

```
››› plusC(read-numC(numC(1)), read-numC(numC(2)))
```

we observe the same behavior as before:

```
Error:

"Program halted waiting for user input"
```

```
››› run-wc(3)
```

```
Error:

"Program halted waiting for user input"
```

```
››› run-wc(4)
```

```
numV(7)
```

Despite their similarities, there are two major differences between the two strategies:

1. When using CPS, the hard work was actually done in the program transformation. The interpreter as a whole was essentially unchanged from before; indeed, the main addition to the interpreter was effectively debugging support in the form of halting its execution, so we could make sure the continuation strategy was correct. Here, the transformation is of the interpreter itself, done one time, and the interpreter works to generate the continuations.

2. In particular, the continuation now closes over the rest of the behavior, not of the interpret*ed* program but the interpret*ing* one. Because the latter's job, however, is to precisely mimic that of the former, we cannot observe this difference.

In the latter case, static scope (in Pyret) ensures that the correct computations are resumed, even if we have multiple continuations stored. Both strategies implicitly point out that continuations are themselves *statically scoped*.

33.4 Generators

Many programming languages now have a notion of *generators*. A generator is like a procedure, in that one can invoke it in an application. Whereas a regular procedure always begins execution at the beginning, a generator *resumes* from where it last left off. Of course, that means a generator needs a notion of "exiting before it's done". This is known as *yielding*, namely returning control to whatever called it.

There are many variations between generators. The points of variation, predictably, have to do with how to enter and exit a generator:

- In some languages a generator is an object that is instantiated like any other object, and its execution is resumed by invoking a method (such as `next` in Python). In others it is just like a procedure, and indeed it is re-entered by applying it like a function.

In languages where values in addition to regular procedures can be used in an application, all such values are collectively called *applicables*.

- In some languages the yielding operation—such as Python's `yield`—is available only inside the syntactic body of the generator. In others, such as Racket, `yield` is an applicable value bound in the body, but by virtue of being a value, it can be passed to abstractions, stored in data structures, and so on.

Python's design represents an extreme point in that a generator is simply *any function that contains the keyword* `yield` *in its body*. In addition, Python's `yield` cannot be passed as a parameter to another function that performs the yielding on behalf of the generator.

There is also a small issue of naming. In many languages with generators, the yielder is *automatically* called `yield` (as in Python). Another possibility is that the user of the generator must indicate in the generator expression what name to give the yielder; for example, in Racket,

Curiously, Python expects users to determine what to call `self` or `this` in objects, but it does not provide the same flexibility for `yield`, because it has no other way to determine which functions are generators!

```
(generator (yield) (from)
           (rec (f (lambda (n)
                     (begin
                       (yield n)
                       (f (+ n 1)))))
             (f from)))
```

but it might equivalently be

```
(generator (y) (from)
          (rec (f (lambda (n)
                    (begin
                      (y n)
                      (f (+ n 1)))))
            (f from)))
```

and if the yielder is an actual value, a user can also abstract over yielding:

```
(generator (y) (from)
          (rec (f (lam (n)
                    (seq
                      ((yield-helper y) n)
                      (f (+ n 1)))))
            (f from)))
```

where `yield-helper` will presumably perform the actual yielding.

There are actually two more design decisions:

1. Is `yield` a statement or expression? In many languages it is actually an expression, meaning it has a value: the one supplied when resuming the generator. This makes the generator more flexible because the user of a generator can use the parameter(s) to alter the generator's behavior, rather than being *forced* to use state to communicate desired changes.

2. What happens at the end of the generator's execution? In many languages, a generator raises an exception to signal its completion.

To implement generators, it will be especially useful to work from our CPS interpreter. Why? Remember how generators work: to yield, a generator must

- remember where in its execution it currently is, and
- know where in its caller it should return to.

while, when invoked, it should

- remember where in its execution its caller currently is, and
- know where in its body it should return to.

Observe the duality between invocation and yielding.

As you might guess, these "where"s correspond to continuations.

Exercise

Add generators to a CPS interpreter.

Exercise

How do generators differ from coroutines and threads? Implement coroutines and threads using a similar strategy.

Exercise

We have seen that Python's generators do not permit any abstraction over yielding, whereas Racket's do. Assuming this was intentional, why might Python have made such a design decision?

33.5 Continuations and Stacks

Surprising as it may seem, CPS conversion actually provides tremendous insight into the nature of the program execution *stack*. The first thing to understand is that every continuation is actually *the stack itself*. This might seem odd, given that stacks are low-level machine primitives while continuations are seemingly complex procedures. But what is the stack, really?

- It's a record of what remains to be done in the computation. So is the continuation.
- It's traditionally thought of as a list of stack *frames*. That is, each frame has a reference to the frames remaining after it finishes. Similarly, each continuation is a small procedure that refers to—and hence closes over—its own continuation. If we had chosen a different representation for program instructions, combining this with the data structure representation of closures, we would obtain a continuation representation that is essentially the same as the machine stack.
- Each stack frame also stores procedure parameters. This is implicitly managed by the procedural representation of continuations, whereas this was done explicitly in the data stucture representation (using `bind`).

- Each frame also has space for "local variables". In principle so does the continuation, though by desugaring local binding, we've effectively reduced everything to procedure parameters. Conceptually, however, some of these are "true" procedure parameters while others are local bindings turned into procedure parameters by desugaring.

- The stack has references to, but does not close over, the heap. Thus changes to the heap are visible across stack frames. In precisely the same way, closures refer to, but do not close over, the store, so changes to the store are visible across closures.

Therefore, traditionally the stack is responsible for maintaining lexical scope, which we get automatically because we are using closures in a statically-scoped language.

Now we can study the conversion of various terms to understand the mapping to stacks. For instance, consider the conversion of a function application (*<cps-trans-appC>*). How do we "read" this? As follows:

- Let's use `k` to refer to the stack present before the function application begins to evaluate.

- When we begin to evaluate the function position (`f`), create a new stack frame (`fdC("f-v"): ...;`. This frame has one free identifier: `k`. Thus its closure needs to record one element of the environment, namely the rest of the stack.

- The code portion of the stack frame represents what is left to be done once we obtain a value for the function: evaluate the argument, and perform the application, and return the result to the stack expecting the result of the application: `k`.

- When evaluation of `f` completes, we begin to evaluate `a`, which also creates a stack frame: `fdC("a-v"): ...;`. This frame has *two* free identifiers: `k` and `f-v`. This tells us:

  - We no longer need the stack frame for evaluating the function position, but
  - we now need a *temporary* that records the value—hopefully a function value—of evaluating the function position.

- The code portion of this second frame also represents what is left to be done: invoke the function value with the argument, in the stack expecting the value of the application.

Similarly, examining the CPS conversion of conditionals would tell us that we have to create a new frame to evaluate the conditional expression we have to create a new stack frame. This frame closes over the stack expecting the value of the entire conditional. This frame makes a decision based on the value of the conditional expression, and invokes one of the other expressions. Once we have examined this value the frame created to evaluate the conditional expression is no longer necessary, so evaluation can proceed in the original continuation.

Viewed through this lens, we can more easily provide an operational explanation for generators. Each generator has its own private stack, and when execution attempts to return past its end, our implementation raises an error. On invocation, a generator stores a reference to the stack of the "rest of the program", and resumes its own stack. On yielding, the system swaps references to stacks. Coroutines, threads, and generators are all conceptually similar: they are all mechanisms to create "many little stacks" instead of having a single, global stack.

33.6 Tail Calls

Observe that the stack patterns above add a frame to the current stack, perform some evaluation, and eventually always return to the current stack. In particular, observe that in an application, we need stack space to evaluate the function position and then the arguments, but once all these are evaluated, we resume computation using the stack we started out with before the application. In other words, *function calls do not themselves need to consume stack space*: we only need space to compute the arguments.

However, not all languages observe or respect this property. In languages that do, programmers can use *recursion* to obtain *iterative behavior*: i.e., a sequence of function calls can consume no more stack space than no function calls at all. This removes the need to create special looping constructs; indeed, loops can simply be expressed as a syntactic sugar.

Of course, this property does not apply in general. If a call to `f` is performed to compute an argument to a call to `g`, the call to `f` is still consuming space relative to the context surrounding `g`. Thus, we should really speak of a relationship between expressions: one expression is in *tail position* relative to another if its evaluation requires no additional stack space beyond the other. In our CPS desugaring, every expression that uses `k` as its continuation—such as a function application after all the sub-expressions have been evaluated, or the then- and else-branches of a conditional—are all in tail position relative to the enclosing application (and perhaps recursively further up). In contrast, every expression that has to create a new stack frame is not in tail position.

Some languages have special support for tail *recursion*: when a procedure calls itself in tail position relative to its body. This is obviously useful, because it enables recursion to efficiently implement loops. However, it hurts “loops” that cannot be squeezed into a single recursive function. For instance, when implementing a scanner or other state machine, it is most convenient to have a set of functions each representing one state, and transitioning to other states by making (tail) function calls. It is onerous (and misses the point) to turn these into a single recursive function. If, however, a language recognizes tail calls as such, it can optimize these cross-function calls just as much as it does intra-function ones.

Scheme and Racket, in particular, promise to implement tail calls without allocating additional stack space. Though some people refer to this as “tail call optimization”, this term is misleading: an optimization is optional, whereas whether or not a language promises to properly implement tail calls is a *semantic* feature. Developers need to know how the language will behave because it affects how they program: they need to know how to structure their loops!

Because of this feature, observe something interesting about the program after CPS transformation: all of its function applications are themselves tail calls! Assuming the program might terminate at any call is tantamount to not using any stack space at all (because the stack would get wiped out).

Exercise

Any program that consumes some amount of stack, when converted to CPS and run, suddenly consumes no stack space at all. Why?

As a corollary, does conversion to CPS reduce the overall memory footprint of the program?

Exercise

Java’s native security model employs a mechanism called stack inspection (look it up if you aren’t familiar with it). What is the interaction between CPS and stack inspection? That is, if we were to CPS a program, would this affect its security behavior?

If not, why not?

If so, how, and what would you suggest doing to recover security assuming the CPS conversion was necessary?

Chapter 34

Pyret for Racketeers and Schemers

Contents

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.

34.1 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`

34.2 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`.

34.3 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

```
(define (dist x y)
  (sqrt (+ (* x x)
           (* y y))))
```

Pyret

```
fun dist(x, y):
  num-sqrt((x * x) +
           (y * y))
end
```

34.4 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

```
(check-expect 1 1)
```

Pyret

```
check:
  1 is 1
end
```

Note that multiple tests can be put into a single block:

RSW

```
  (check-expect 1 1)
  (check-expect 2 2)
```

Pyret

```
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.

34.5 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. Thus:

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”.

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.

34.6 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 contast, 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.

34.7 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:
 (if full-moon     "howl"
     "howl"      else:
     "meow")       "meow"
                 end
```

RSW **Pyret**

```
                    if full-moon:
 (if full-moon        "howl"
     "howl"         else if new-moon:
     (if new-moon     "bark"
         "bark"     else:
         "meow"))     "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.

34.8 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`

```
(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").

34.9 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`

34.10 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.

34.11 What Else?

If there are other parts of Scheme or Racket syntax that you would like to see translated, please let us know.

Chapter 35

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 f is idempotent if, for all x in its domain, $f(f(x)) = f(x)$ (and by induction this holds for additional applications of f).

☞ *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 get 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 `e1` and `e2`, the sum `e1 + e2`", we do not mean the programmer literally wrote "`e1`" in the program; rather we are using `e1` to refer to whatever the programmer might write on the left of the addition sign. Therefore, `e1` 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 bytes 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 chacters on an input stream, and the output might be expected to be a rich, 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 [section 22.3], using a ☞ *cache*, environments [section 26.2], etc. are all 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 [section 23.2].