Creative Scala

Dave Gurnell and Noel Welsh

March 2020

Creative Scala

March 2020

Published by Underscore Consulting LLP, Brighton, UK.

Copies of this, and related topics, can be found at http://underscore.io/training.

Team discounts, when available, may also be found at that address.

Contact the author regarding this text at: hello@underscore.io.

Contents

Foreword

Creative Scala is aimed at developers who have no prior experience in Scala. It is designed to give you a fun introduction to functional programming. We assume you have some very basic familiarity with another programming language but little or no experience with Scala or other functional languages.

We have three goals with this book:

1. To give an introduction to functional programming so that you can calculate and reason about programs, and pick up and understand other introductory books on functional programming.
2. To teach you enough Scala that you can explore your own interests in and using Scala.
3. To present all this in a fun, gentle, and interesting way via two-dimensional computer graphics.

Our motivation comes from our own experience learning programming, studying functional programming, and teaching Scala to commercial developers.

Firstly, we believe that functional programming is the future. Since we're assuming you have little programming experience we won't go into the details of the differences between functional programming and object-oriented programming that you may have already experienced. Suffice to say there are different ways to think about and write computer programs, and we've chosen the functional programming approach.

The reason for choosing functional programming are more interesting. It's common to teach programming by what we call the "bag of syntax" approach. In this approach a programming language is taught a collection of syntactical features (variables, for loops, while loops, methods) and students are left to figure out on their own when to use each feature. We've seen this method fail both when we were undergraduates learning programming, and as postgraduates teaching programming, as students simply have no systematic way to break down a problem and turn it into code. The result is that many students dropped out due to the poor quality of teaching. The students that remained tended to, like us, already have extensive programming experience.

Let's think back to primary school maths, specifically column addition. This is the basic way we're taught to add up numbers when they're too big to do in our head. So, for example, adding up 266 + 385, we would line up the columns, carry the tens and so on. Now maybe maths wasn't your favorite subject but there are some important lessons here. The first is that we're given a systematic way to arrive at the solution. We can *calculate* the solution once we realise this is a problem that requires column addition. The second point is that we don't even have to understand why column addition works (though it helps) to use it. So long as we follow the steps we'll get the correct answer.

The remarkable thing about functional programming is that it works like column addition. We have recipes that are guaranteed to give us the correct answer if we follow them correctly. We call this *calculating* a program. This is not to say that programming is without creativity, but the challenge is to understand the structure of the problem and once we've done that the recipe we should use follows immediately. The code itself is not the interesting part.

We're teaching functional programming using Scala, but not Scala itself. Scala is a language that is in demand right now. Scala programmers can relatively easily get jobs in a variety of industries, and this is an important

motivation for learning Scala. One of the reasons for Scala's popularity is that is straddles object-oriented programming, the old way of programming, and functional programming. There is a lot of code written in an object-oriented style, and a lot of programmers who are used to that style. Scala gives a gentle way from object-oriented programming to functional programming. However this means Scala is a large language, and the interaction between the object-oriented and functional parts can be confusing. We believe that functional programming is much more effective than object-oriented programming, and for new programmers there is no need to add the confusion of learning object-oriented techniques at the same time. That can come later. Therefore this book is exclusively using the functional programming parts of Scala.

Finally, we've chosen what we hope is a fun method to explore functional programming and Scala: computer graphics. There are many introductions to Scala, but the majority use examples that either relate to business or mathematics. For example, one of the first exercises in the very popular Coursera course is to implement sets via indicator functions. We feel if you're the type of person who likes directly working with these sort of concepts you already have plenty of content available. We want to target a different group: people who perhaps thought that maths was not for them but nonetheless have an interest or appreciation in the visual arts. We won't lie: there is maths in this book, but we hope we manage to motivate and indeed visualise the concepts in a way that makes them less intimidating.

Although this book will give you the basic mental model required to become competent with Scala, you won't finish knowing *everything* you need to be self-sufficient. For further advancement we recommend considering one of the many excellent Scala textbooks out there, including our own Essential Scala.

If you are working through the exercises on your own, we highly recommend joining our Gitter chat room to provide get help with the exercises and provide feedback on the book.

The text of Creative Scala is open source, as is the source code for the Doodle drawing library used in the exercises. You can grab the code from our GitHub account. Contact us on Gitter or by email if you would like to contribute.

Thanks for downloading and happy creative programming!

—Dave and Noel

Notes on the Early Access Edition

> This is an *early access* release of Creative Scala. There may be typos and other errors in the text and examples.
>
> If you spot any mistakes or would like to provide feedback, please let us know via our Gitter chat room or by email:
>
> - Dave Gurnell (dave@underscore.io)
> - Noel Welsh (noel@underscore.io)

Acknowledgements

Creative Scala was written by Dave Gurnell and Noel Welsh. Many thanks to Richard Dallaway, Jonathan Ferguson, and the team at Underscore for their invaluable contributions and extensive proof reading.

Thanks also to the many people who pointed out errors or made suggestions to improve the book: Neil Moore; Kelley Robinson, Julie Pitt, and the other ScalaBridge organizers; d43; Matt Kohl; Alexa Kovachevich and all the other students who worked through Creative Scala at ScalaBridge, at another event, or on their own; and the

many awesome members of the Scala community who gave us comments and suggestions in person. Finally, we have large amount of gratitude for Bridgewater, and particularly Lauren Cipicchio, who perhaps unknowingly funded a good portion of the initial development of the second version of the Creative Scala, and provided the first few rounds of students.

Finally, Creative Scala owes a large intellectual debt to the work of many researchers working in programming language theory and Computer Science education. In particular we'd like to highlight:

- the work of the PLT research group, and in particular the book "How to Design Programs", by Matthew Flatt, Matthias Felleisen, Robert Bruce Findler, and Shriram Krishnamurthi; and
- the "creative coding" approach to introductory programming pioneered by Mark Guzdial, Dianna Xu, and others.

Chapter 1

Getting Started

Our first step is to install the software we need to work with Creative Scala. We describe two pathways here:

1. Working with a text editor and a terminal. *We recommend this setup to people completely new to programming* as there are fewer moving parts.
2. Working with IntelliJ IDEA. We recommend this setup to people who are used to using an IDE or are uncomfortable with the terminal.

If you're an experienced developer with a setup you are happy with just keep the tools you know and adapt the instructions below as needed.

If all this stuff is new to you, the rest of the chapter has some background.

1.1 Installing Terminal Software and a Text Editors

This section is our recommended setup for people new to programming, and describes how to setup Creative Scala with the terminal and a text editor. We need to install:

- the JVM;
- Git;
- a text editor; and
- the template project for Creative Scala.

1.1.1 OS X

Open the terminal. (Click the magnifying glass icon on the top righthand side of the toolbar. Type in "terminal".)

Install Java. Type into the terminal

```
java
```

If this runs you already have Java installed. Otherwise it will prompt you to install Java.

Install homebrew. Paste into the terminal

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Install `git` using homebrew. At the terminal, type

```
brew install git
```

Now install the text editor Atom. Again type at the terminal

```
brew install Caskroom/cask/atom
```

Install Scala support inside Atom: Settings > Install > language-scala

Now we will use Git to get an SBT project that will work with Creative Scala. Type

```
git clone https://github.com/underscoreio/creative-scala-template.git
```

> **Sharing Your Work**
>
> There is an alternative setup that involves first forking the Creative Scala template project, and then cloning it to your computer. This is the setup to choose if you want to share your work with other people; for example you might be taking Creative Scala with a remote instructor or you might just (quite rightfully) be proud of your work.
>
> In this setup you first *fork* the Creative Scala template. Then you make a clone of *your* fork. This alternative setup is described in more detail in the section on GitHub later in this chapter.

Now change to the directory we just created and run SBT.

```
cd creative-scala-template
./sbt.sh
```

SBT should start. Within SBT type `console`. Finally type

```
Example.image.draw()
```

and an image of three circles should appear!

If you've made it this far you've successfully installed all the software you need for work through Creative Scala.

The final step is to load Atom and use it to open `Example.scala`, which you can find in `src/main/scala`.

1.1.2 Windows

Download and install Java. Search for the "JDK" (Java development kit). This will take you to Oracle's site. Accept their license and download the JDK. Run the installer you just downloaded.

Download and install Atom. Go to `https://atom.io/` and download Atom for Windows. Run the installer you've just downloaded.

Download and install Git. Go to `https://git-scm.com/` and download Git for Windows. Run the installer you've just downloaded. At the very end it gives you the option to open Git. Select that option. A window will open up with a command prompt. Type

```
git clone https://github.com/underscoreio/creative-scala-template.git
```

> **Sharing Your Work**
>
> There is an alternative setup that involves first forking the Creative Scala template project, and then cloning it to your computer. This is the setup to choose if you want to share your work with other people; for example you might be taking Creative Scala with a remote instructor or you might just (quite rightfully) be proud of your work.
>
> In this setup you first *fork* the Creative Scala template. Then you make a clone of *your* fork. This alternative setup is described in more detail in the section on GitHub later in this chapter.

Open a normal command-prompt. Click on the Windows icon on the bottom left of the screen. In the search box enter "cmd" and run the program it finds. In the window that is opened up type

```
cd creative-scala-template
```

which will change into the directory of the Creative Scala template project we just downloaded. Type

```
sbt.bat
```

to start SBT. Within SBT type `console`. Finally type

```
Example.image.draw()
```

and an image of three circles should appear!

If you've made it this far you've successfully installed all the software you need for work through Creative Scala.

The final step is to load Atom and use it to open `Example.scala`, which you can find in the directory `src\main\scala`.

1.1.3 Linux

Follow the OS X instructions, using your distributions package manager to install software in place of Homebrew.

1.2 IntelliJ

IntelliJ is an integrated development environment (IDE) for Scala and other programming languages. It integrates an number of programming tools into one application, and we recommend it for people who are used to using other IDEs such as Visual Studiol or Eclipse.

Start by downloading and installing IntelliJ. You can use the free community edition for Creative Scala. When installing IntelliJ you will be asked a lot of questions. You can accept the defaults for the most part. When you are asked about "featured plugins", *make sure you install the Scala plug-in*.

Once you have completed the configuration you will be presented with a dialog asking if you want to create a new project, import a project, open a file, or check out from version control. Choose "checkout from version control", and select GitHub.

In the dialog box that opens change "Auth type" to Token. Now visit GitHub in a web browser. Select your account (top right hand of the page). Choose "Settings" and then choose "Personal access tokens". Generate a token. Call it "intellij" and select the "repo" check box. Copy the long string of numbers and text and paste it into the "Token" box. Now login to GitHub.

Install the SBT console add-on.

1.3 Background

This section gives some background information on some of the tools we'll be using. If you're an experienced developer a lot of this will be old hat, and you can skip it. If you're not, this will hopefully give some useful context to the software we'll be working with.

1.3.1 The Terminal

Back when the world was young and computing was in its infancy, the common user interface of graphical windows, a cursor controlled by a mouse, and interaction by *direct manipulation* didn't exist. Instead users typed in commands at a device called a *terminal*. The direct manipulation interface is superior for most uses, but there are some cases for which the terminal or *command line* is preferable. For example, if we wanted to work out how much space was used by all the files which names starting with `data` in Linux or OS X we can execute the command

```
du -hs data*
```

We can break this down into three components:

- the command `du` means disk usage;
- the flags `-hs` mean to print a human readable summary; and
- the pattern `data*` means all the files whose names begin with `data`.

Doing this with a direct manipulation interface would be much more time consuming.

The command line has a steep learning curve, but the reward is an extremely powerful tool. Our usage of the terminal will be very limited, so don't worry if you find the example above intimidating!

1.3.2 Text Editors

You're probably used to writing documents in a word processor. A word processor allows us to write text and control the formatting of how it appears on the (increasingly rare) printed page. A word processor includes powerful commands, such as a spell checker and automatic table of contents generation, to make editing prose easier.

A *text editor* is like a word processor for code. Whereas a word processor is concerned about visual presentation of text, a text editor has many programming specific functions. Typical examples include powerful tools to search and replace text, and the ability to quickly jump between the many different files that make up a project.

Text editors date back to the days of terminals and perhaps surprisingly some of these tools are still in use. The two main ancient and glorious text editors that survive are called Emacs and Vim. They have very different approaches (except when they don't) and developers tend to use one or the other. I've been using Emacs for about twenty years, and thus I know in my bones that Emacs is the greatest of all possible text editors and Vim users are knuckle-draggers lumbered with poor taste and an inferior tool. Vim users no doubt think the same about me.

If there is one thing that unites Vim and Emacs users it's the sure knowledge that new-fangled text editors like Sublime Text and Atom are bringing about the downfall of our civilization. Nonetheless we recommend using Atom if you're new to this text editing game. Both Vim and Emacs were created before the common interfaces in use today were created, and using them requires learning a very different way of working.

1.3.3 The Compiler

The code we write in a text editor is not in a form that a computer can run. A *compiler* translates it into something the computer can run. As it does this it performs certain checks on the code. If these checks don't pass the code won't be compiled and the compiler will print an error message instead. We'll learn more about what the compiler can check and what it can't in the rest of this book.

When we said the compiler translates the code is something the computer can run, this is not the complete truth in the case of Scala. The output of the compiler is something called bytecode, and another program, called the Java Virtual Machine (JVM), runs this code[1].

1.3.4 Integrated Development Environments

Integrated development environments (IDEs) are an alternative approach that combine a text editor, a compiler, and several other programmer tools into a single program. Some people swear by IDEs, while some people prefer to use the terminal and a text editor. Our recommendation if you're new to programming is to take the terminal-and-text-editor approach. If you're already used to an IDE then IntelliJ IDEA is currently the best IDE for Scala development.

1.3.5 Version Control

Version control is the final tool we'll use. A version control system is a program that allows us to keep a record of all the changes that have been made to a group of files. It's very useful for allowing multiple people to work on a project at the same time, and it ensures people don't accidentally overwrite each others changes. This is not a huge concern in Creative Scala, but it is good to get some exposure to version control now.

The version control software we'll use is called Git. It's powerful but complex. The good news is we don't need to learn much about Git. Most of our use of Git will be via a website called GitHub, which allows people to share software that is stored in Git. We use GitHub to share the software used in Creative Scala.

1.3.6 Onward!

Now that we've got some background, let's move on to installing the software we need to write Scala code.

[1] This is not itself the entire truth! We usually run Scala code on the JVM, but we can actually compile Scala to three different formats. The first and most common is JVM bytecode. We can also compiled to Javascript, another programming language, which allows us to run Scala code in a web browser. Finally, Scala Native will compile Scala to something a computer *can* run directly without requiring the JVM.

1.4 GitHub

We have created a template for you that will get you set up with all the code you need to work through Creative Scala. This template is stored on GitHub, a website for sharing code.

You can copy the template onto your computer, which Git calls cloning, but this means you won't be able to save any changes you make back to GitHub where other people can view them.

If you want to be able to share your changes, you need to make a copy of the template project on GitHub that you own. Git calls this forking. You fork the repository on GitHub and then clone *your fork* to your computer. Then you can save your changes back to your fork on GitHub.

To start this process you need to create a GitHub account, if you do not have one already.

Once you have an account, visit the template project in your browser. At the top right is button called "Fork". Press this button to create your own copy of the template. You will be taken to a web page displaying your own fork of the template. Remember the name of this repository. It should be something like `yourname/creative-scala-template` where `yourname` is your GitHub user name.

Now cloning your fork is as simple as running this command and replacing `yourname` with your actual GitHub user name.

```
git clone git@github.com:yourname/creative-scala-template.git
```

Now any changes you make can be sent back to your fork on GitHub. The process for doing this in Git is a bit involved. When you've made a change you must:

- `add` the change to what's called Git's index;
- `commit` the change; and finally
- `push` the change to the fork.

Here's an example of using the command line to do this.

```
git add
git commit -m "Explain here what you did"
git push
```

GitHub makes a nice free graphical tool for using Git, called GitHub Desktop. It's probably the easiest way to use Git when you're getting started.

Chapter 2

Expressions, Values, and Types

Scala programs have three fundamental building blocks: *expressions*, *values*, and *types*. In this section, we explore these concepts.

Here's a very simple expression:

```
1 + 2
```

An *expression* is a fragment of Scala code. We can write expressions in a text editor, or on a piece of paper, or on a wall; expressions are like writing. Just like writing must be read for it to have any effect on the world (and the reader has to understand the language the writing is written in), the computer must *run* an expression for it to have an effect. The result of running an expression is a *value*. Values live in the computer's memory, in the same way that the result of reading some writing lives in the reader's head. We will also say expressions are *evaluated* or *executed* to describe the process of transforming them into values.

We can evaluate expressions immediately by writing them at the console and pressing "Enter" (or "Return"). Try it now.

```
1 + 2
// res1: Int = 3
```

The console responds with the value the expression evaluates to, and the type of the expression.

The expression `1 + 2` evaluates to the value 3. We can write down the number three here on the page, but the real value is something stored in the computer's memory. In this case, it is a 32-bit integer represented in two's-complement. The meaning of "32-bit integer represented in two's-complement" is not important. We just mention it to emphasize the fact the computer's representation of the value 3 is the true value, not the numeral written here or displayed by the console.

Types are the final piece of the puzzle. A type describes a set of values. The expression `1 + 2` has the type `Int`, which means the value the expression evaluates to will be one of the over four billion values the computer understands to be integers. We determine the type of an expression *without* running it, which is why the type `Int` doesn't tell us which specific value the expression evalutes to.

Before a Scala program is run, it must be *compiled*. Compilation checks that a program makes sense. It must be syntactically correct, meaning it must be written according to the rules of Scala. For example `(1 + 2)` is syntactically correct, but `(1 + 2` is not because there is no `)` to match the `(`. It must also *type check*, meaning the types must be correct for the operations we're trying to do. `1 + 2` type checks (we are adding integers), but `1.toUpperCase` does not (there is no concept of upper and lower case for integers.)

Only programs that successfully compile can be run. We can think of compilation as being analogous to the rules of grammar in writing. The sentence "FaRf fjrmn;l df.fd" is syntactically incorrect in English. The arrangement of letters doesn't form any words. The sentence "dog fly a here no" is made out of valid words but their arrangement breaks the rules of grammar—analogous to the type checks that Scala performs.

It is important to remember that type checking is done before a program runs. If you have used a language like Python or Javascript, which are sometimes called "dynamically typed", there is no type checking done before a program runs. In a "statically typed" language like Scala the type checking catches some potential errors for us before we run the code. What is sometimes called a type in a dynamically typed language is *not* a type as we understand it. Types, for us, exist at at the time when a program is compiled, which we will call *compile time*. At time when a program runs, which we call *run time*, we have only values. Values may record some information about the type of the expression that created them. If they do we call these *tags*, or sometimes *boxes*. Not all values are tagged or boxed. Avoiding tagging, which is also called *type erasure*, allows for more efficient programs.

2.1 Literal Expressions

We'll now start to explore the various forms of expressions in Scala, starting with the simplest expressions, *literals*. Here's a literal expression:

```
3
// res0: Int = 3
```

A literal evaluates to "itself." How we write the expression and how the console prints the value are the same. Remember though, there is a difference between the written representation of a value and its actual representation in the computer's memory.

Scala has many different forms of literals. We've already seen `Int` literals. There is a different type, and a different literal syntax, for what are called *floating point numbers*. This corresponds to a computer's approximation of the real numbers. Here's an example:

```
0.1
// res1: Double = 0.1
```

As you can see, the type is called `Double`.

Numbers are well and good, but what about text? Scala's `String` type represents a sequence of characters. We write literal strings by putting their contents in double quotes.

```
"To be fond of dancing was a certain step towards falling in love."
// res2: String = "To be fond of dancing was a certain step towards falling in love."
```

Sometimes we want to write strings that span several lines. We can do this by using triple double quotes, as below.

```
"""
A new, a vast, and a powerful language is developed for the future use of analysis,
in which to wield its truths so that these may become of more speedy and accurate
practical application for the purposes of mankind than the means hitherto in our
possession have rendered possible.

  -- Ada Lovelace, the world's first programmer
"""
// res3: String = """
// A new, a vast, and a powerful language is developed for the future use of analysis,
```

```
// in which to wield its truths so that these may become of more speedy and accurate
// practical application for the purposes of mankind than the means hitherto in our
// possession have rendered possible.
//
//   -- Ada Lovelace, the world's first programmer
// """
```

A `String` is a sequence of characters. Characters themselves have a type, `Char`, and character literals are written in single quotes.

```
'a'
// res4: Char = 'a'
```

Finally we'll look at the literal representations of the `Boolean` type, named after English logician George Boole. This fancy name just means a value that can be either `true` or `false`, and this indeed is how we write boolean literals.

```
true
// res5: Boolean = true
false
// res6: Boolean = false
```

With literal expressions, we can create values, but we won't get very far if we can't somehow manipulate the values we've created. We've seen a few examples of more complex expressions like `1 + 2`. In the next section, we'll learn about objects and methods, which will allow us to understand how this, and more interesting expressions, work.

2.2 Values are Objects

In Scala all values are *objects*. An object is a grouping of data and operations on that data. For example, 2 is an object. The data is the integer 2, and the operations on that data are familiar operations like +, -, and so on. We call operations of an object the object's *methods*.

2.2.1 Method Calls

We interact with objects by *calling* or *invoking* methods. For example, we can get the uppercase version of a `String` by calling its `toUpperCase` method.

```
"Titan!".toUpperCase
// res0: String = "TITAN!"
```

Some methods accept *parameters* or *arguments*, which control how the method works. The `take` method, for example, takes characters from a `String`. We must pass a parameter to `take` to specify how many characters we want.

```
"Gilgamesh went abroad in the world".take(3)
// res1: String = "Gil"
"Gilgamesh went abroad in the world".take(9)
// res2: String = "Gilgamesh"
```

A method call is an expression, and thus evaluates to an object. This means we can chain method calls together to make more complex programs:

```
"Titan!".toUpperCase.toLowerCase
// res3: String = "titan!"
```

Method Call Syntax

The syntax for a method call is

```
anExpression.methodName(param1, ...)
```

or

```
anExpression.methodName
```

where

- `anExpression` is any expression (which evaluates to an object)
- `methodName` is the name of the method
- the optional `param1, ...` are one or more expressions evaluating to the parameters to the method.

2.2.2 Operators

We have said that all values are objects, and we call methods with the syntax `object.methodName(parameter)`. How then do we explain expressions like `1 + 2`?

In Scala, and expression written `a.b(c)` can be written `a b c`. So these are equivalent:

```
1 + 2
// res4: Int = 3
1.+(2)
// res5: Int = 3
```

This first way of calling a method is known an *operator* style.

Infix Operator Notation

Any Scala expression written `a.b(c)` can also be written `a b c`.

Note that `a b c d e` is equivalent to `a.b(c).d(e)`, not `a.b(c, d, e)`.

2.3 Types

Now that we can write more complex expressions, we can talk a little more about types.

One use of types is stopping us from calling methods that don't exist. The type of an expression tells the compiler what methods exist on the value it evaluates to. Our code won't compile if we try to call to a method that doesn't exist. Here are some simple examples.

```
"Brontë" / "Austen"
1.take(2)
// error: value / is not a member of String
// "Brontë" / "Austen"
// ^^^^^^^^^^
// error: value take is not a member of Int
// 1.take(2)
// ^^^^^^
```

It really is the type of the expression that determines what methods we can call, which we can demonstrate by calling methods on the result of more complex expressions.

```
(1 + 3).take(1)
// error: value take is not a member of Int
// (1 + 3).take(1)
//  ^^^^^^^^^^^
```

This process of *type checking* also applies to the parameter of methods.

```
1.min("zero")
// error: type mismatch;
//  found   : String("zero")
//  required: Int
// 1.min("zero")
//       ^^^^^^
```

Types are a property of expressions and thus exist at compile time (as we have discussed before.) This means we can determine the type of an expression even if evaluating it results in an error at run time. For example, dividing an `Int` by zero causes a run-time error.

```
1 / 0
// java.lang.ArithmeticException: / by zero
//  at repl.Session$App$$anonfun$4.apply$mcI$sp(types.md:27)
//  at repl.Session$App$$anonfun$4.apply(types.md:27)
//  at repl.Session$App$$anonfun$4.apply(types.md:27)
```

The expression `1 / 0` still has a type, and we can get that type from the console as shown below.

```
:type 1 / 0
// Int
```

We can also write a compound expression including a sub-expression that will fail at run-time.

```
(2 + (1 / 0) + 3)
// java.lang.ArithmeticException: / by zero
//  at repl.Session$App$$anonfun$5.apply$mcI$sp(types.md:35)
//  at repl.Session$App$$anonfun$5.apply(types.md:35)
//  at repl.Session$App$$anonfun$5.apply(types.md:35)
```

This expression also has a type.

```
:type (2 + (1 / 0) + 3)
// Int
```

2.4 Exercises

2.4.0.1 Arithmetic

Write an expression using integer literals, addition, and subtraction that evaluates to 42.

See the solution

2.4.0.2 Appending Strings

Join together two strings (known as *appending* strings) using the ++ method. Write equivalent expressions using both the normal method call style and the operator style.

See the solution

2.4.0.3 Precedence

In mathematics we learned that some operators take *precedence* over others. For example, in the expression `1 + 2 * 3` we should do the multiplication before the addition. Do the same rules hold in Scala?

See the solution

2.4.0.4 Types and Values

Which of the following expressions will not compile? Of the expressions that will compile, what is their type? Which expressions fail at run-time?

```
1 + 2

"3".toInt

"Electric blue".toInt
// java.lang.NumberFormatException: For input string: "Electric blue"
//  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
//  at java.lang.Integer.parseInt(Integer.java:580)
//  at java.lang.Integer.parseInt(Integer.java:615)
//  at scala.collection.immutable.StringLike.toInt(StringLike.scala:304)
//  at scala.collection.immutable.StringLike.toInt$(StringLike.scala:304)
//  at scala.collection.immutable.StringOps.toInt(StringOps.scala:33)
//  at repl.Session$App$$anonfun$9.apply$mcI$sp(exercises.md:48)
//  at repl.Session$App$$anonfun$9.apply(exercises.md:48)
//  at repl.Session$App$$anonfun$9.apply(exercises.md:48)

"Electric blue".take(1)

"Electric blue".take("blue")

1 + ("Moonage daydream".indexOf("N"))

1 / 1 + ("Moonage daydream".indexOf("N"))

1 / (1 + ("Moonage daydream".indexOf("N")))
// java.lang.ArithmeticException: / by zero
//  at repl.Session$App$$anonfun$14.apply$mcI$sp(exercises.md:80)
```

```
//  at repl.Session$App$$anonfun$14.apply(exercises.md:80)
//  at repl.Session$App$$anonfun$14.apply(exercises.md:80)
```

See the solution

2.4.0.5 Floating Point Failings

When we introduced Doubles, I said they are an approximation to the real numbers. Why do you think this is? Think about representing numbers like ⅓ and π. How much space would it take to represent these numbers in decimal?

See the solution

2.4.0.6 Beyond Expressions

In our current model of computation there are only three components: expressions (program text) with types, that evaluate to values (something within the computer's memory). Is this sufficient? Could we write a stock market or a computer game with just this model? Can you think of ways to extend this model?

See the solution

Chapter 3

Computing With Pictures

So far we have computed using numbers, strings, and other simple objects. From here on out we will focus our attention on computing with pictures, and later with animations. Pictures offer us more immediate creative opportunities, and they make our program output tangible in a way that other methods can't deliver.

We'll use a library called Doodle to help us with creating graphics. In this chapter we will learn the basics of Doodle.

> If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.
>
> ```
> import doodle.core._
> import doodle.image._
> import doodle.image.syntax._
> import doodle.image.syntax.core._
> import doodle.java2d._
> ```

3.1 Images

Let's start with some simple shapes, programming at the console as we've done before.

```
Image.circle(10)
// res0: Image = Circle(10.0)
```

What is happening here? `Image` is an object and `circle` a method on that object. We pass to `circle` a parameter, `10` that gives the diameter of the circle we're constructing. Note the type of the result—an `Image`.

```
Image.circle(10)
// res1: Image = Circle(10.0)
```

We draw the circle by calling the `draw` method.

```
Image.circle(10).draw()
```

A window should appear as shown in fig. 3.1.

Doodle supports a handful of "primitive" images: circles, rectangles, and triangles. Let's try drawing a rectangle.

Figure 3.1: A circle

Figure 3.2: A rectangle

```
Image.rectangle(100, 50).draw()
```

The output is shown in fig. 3.2.

Finally let's try a triangle, for which the output is shown in fig. 3.3.

```
Image.triangle(60, 40).draw()
```

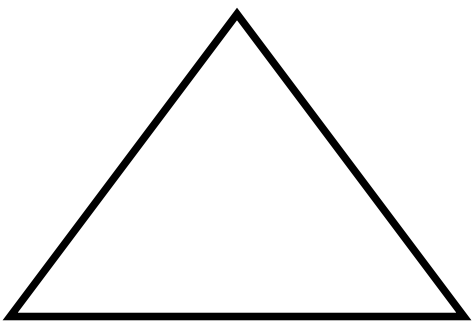

Figure 3.3: A triangle

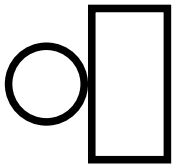

Figure 3.4: A circle beside a rectangle

Exercises

I Go Round in Circles

Create circles that are 1, 10, and 100 units wide. Now draw them!

See the solution

My Type of Art

What is the type of a circle? A rectangle? A triangle?

See the solution

Not My Type of Art

What's the type of *drawing* an image? What does this mean?

See the solution

3.2 Layout

We can seen how to create primitive images. We can combine together images using layouts methods to create more complex images. Try the following code—you should see a circle and a rectangle displayed beside one another, as in fig. 3.4.

```
(Image.circle(10).beside(Image.rectangle(10, 20))).draw()
```

`Image` contains several layout methods for combining images, described in tbl. 3.1. Try them out now to see what they do.

Table 3.1: Layout methods available in Doodle

Method	Parameter	Description	Example
`beside`	`Image`	Places images horizontally next to one another	`Image.circle(10) .beside(Image.circle(10))`
`above`	`Image`	Places images vertically next to one another	`Image.circle(10) .above(Image.circle(10))`
`below`	`Image`	Places images vertically next to one another	`Image.circle(10) .below(Image.circle(10))`

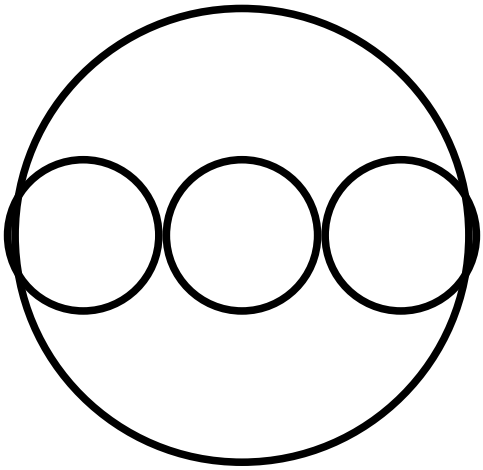

Figure 3.5: The width of a circle

Method	Parameter	Description	Example
`on`	`Image`	Places images centered on top of one another	`Image.circle(10) .on(Image.circle(10))`
`under`	`Image`	Places images centered on top of one another	`Image.circle(10) .under(Image.circle(10))`

Exercises

The Width of a Circle

Create the picture fig. 3.5 using the layout methods and basic images we've covered so far.

See the solution

3.3 Color

In addition to layout, Doodle has some simple operators to add a splash of colour to our images. Try these out the methods described in tbl. 3.2 to see how they work.

Table 3.2: Some of the methods to add color to images in Doodle.

Method	Parameter	Description	Example
`fillColor`	`Color`	Fills the image with the specified color.	`Image.circle(10) .fillColor(Color.red)`
`strokeColor`	`Color`	Outlines the image with the specified color.	`Image.circle(10) .strokeColor(Color.blue)`
`strokeWdith`	`Double`	Sets the width of the image outline.	`Image.circle(10) .strokeWidth(3)`
`noFill`	None	Removes any fill from the image.	`Image.circle(10).noFill`

Method	Parameter	Description	Example
`noStroke`	None	Removes any stroke from the image.	`Image.circle(10).noStroke`

Doodle has various ways of creating colours. The simplest are the predefined colours in CommonColors.scala. Some of the most commonly used are described in tbl. 3.3.

Table 3.3: Some of the most common predefined colors.

Color	Type	Example
`Color.red`	`Color`	`Image.circle(10).fillColor(Color.red)`
`Color.blue`	`Color`	`Image.circle(10).fillColor(Color.blue)`
`Color.green`	`Color`	`Image.circle(10).fillColor(Color.green)`
`Color.black`	`Color`	`Image.circle(10).fillColor(Color.black)`
`Color.white`	`Color`	`Image.circle(10).fillColor(Color.white)`
`Color.gray`	`Color`	`Image.circle(10).fillColor(Color.gray)`
`Color.brown`	`Color`	`Image.circle(10).fillColor(Color.brown)`

Exercises

Evil Eye

Make the image in fig. 3.6, designed to look like a traditional amulet protecting against the evil eye. I used `cornflowerBlue` for the iris, and `darkBlue` for the outer color, but experiment with your own choices!

See the solution

3.4 Creating Colors

We've seen how to use predefined colors in our images. What about creating our own colors? In this section we will see how to create colors of our own, and transform existing colors into new ones.

3.4.1 RGB Colors

Computers work with colors defined by mixing together different amounts of red, green, and blue. This "RGB" model is an additive model of color, which means adding more colors gets us closer to white. This is the oppositive of paint, which is a subtractive model where adding more paints gets us closer to black. Each red, green, or blue component can have a value between zero and 255. If all three components are set to the maximum of 255 we get pure white. If all components are zero we get black.

We can create our own RGB colors using the `rgb` method on the `Color` object. This method takes three parameters: the red, green, and blue components. These are numbers between 0 and 255, called an `UnsignedByte`[1]. There is no literal expression for `UnsignedByte` like there is for `Int`, so we must convert an `Int` to `UnsignedByte`. We can do this with the `uByte` method. An `Int` can take on more values that an `UnsignedByte`, so if

[1]A byte is a number with 256 possible values, which takes 8 bits within a computer to represent. A signed byte has integer values from -128 to 127, while an unsigned byte ranges from 0 to 255.

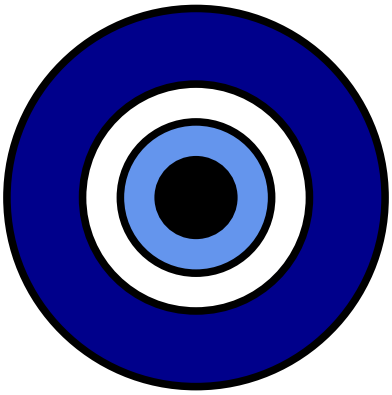

Figure 3.6: No evil eyes here!

the number is too small or too large to be represented as a `UnsignedByte` it will be converted to the closest values is the range 0 to 255. These examples illustrate the conversion.

```
0.uByte.get
// res0: Int = 0
255.uByte.get
// res1: Int = 255
128.uByte.get
// res2: Int = 128
-100.uByte.get // Too small, is transformed to 0
// res3: Int = 0 // Too small, is transformed to 0
1000.uByte.get // Too big, is transformed to 255
// res4: Int = 255
```

(Note that `UnsignedByte` is a feature of Doodle. It is not something provided by Scala.)

Now we know how to construct `UnsignedBytes` we can make RGB colors.

```
Color.rgb(255.uByte, 255.uByte, 255.uByte) // White
Color.rgb(0.uByte, 0.uByte, 0.uByte) // Black
Color.rgb(255.uByte, 0.uByte, 0.uByte) // Red
```

3.4.2 HSL Colors

The RGB color representation is not very easy to use. The hue-saturation-lightness (HSL) format more closely corresponds to how we perceive color. In this representation a color consists of:

- *hue*, which is an angle between 0 and 360 degrees giving a rotation around the color wheel.
- *saturation*, which is a number between 0 and 1 giving the intensity of the color from a drab gray to a pure color; and
- *lightness* between 0 and 1 giving the color a brightness varying from black to pure white.

fig. 3.7 shows how colors vary as we change hue and lightness, and fig. 3.8 shows the effect of changing saturation.

We can construct a color in the HSL representation using the `Color.hsl` method. This method takes as parameters the hue, saturation, and lightness. The hue is an `Angle`. We can convert a `Double` to an `Angle` using the `degrees` (or `radians`) methods.

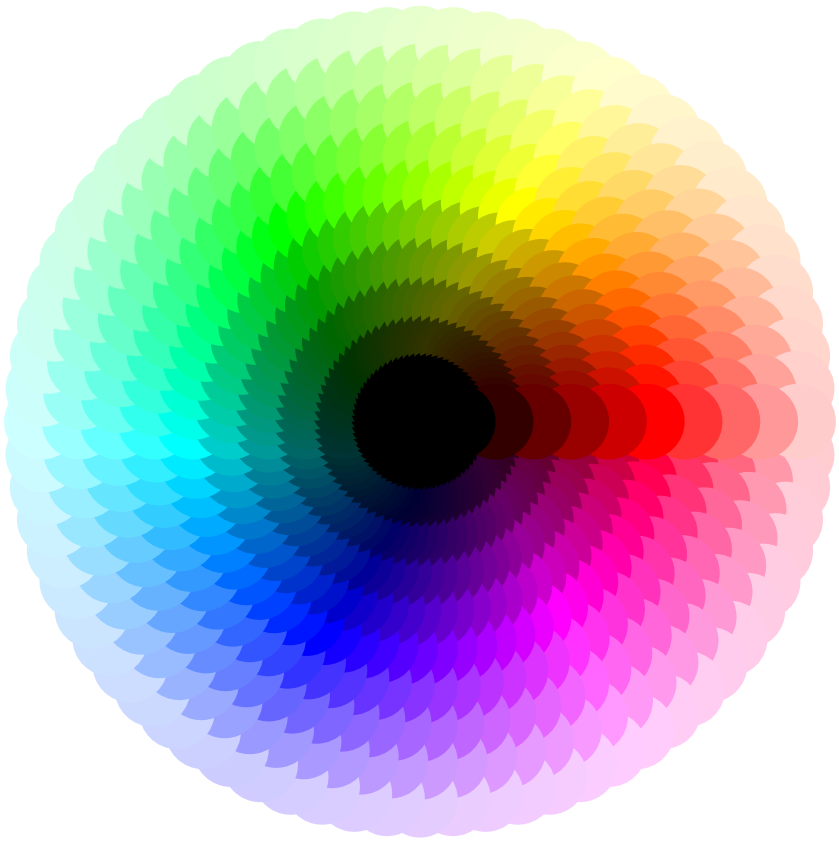

Figure 3.7: A color wheel showing changes in hue (rotations) and lightness (distance from the center) with saturation fixed at 1.

Figure 3.8: A gradient showing how changing saturation effects color, with hue and lightness held constant. Saturation is zero on the left and one on the right.

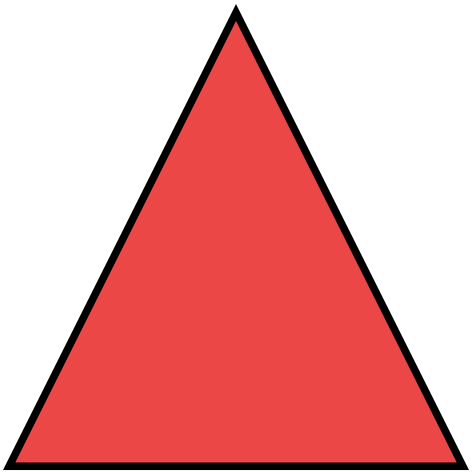

Figure 3.9: Rendering pastel red in a triangle

```
0.degrees
// res8: Angle = Angle(0.0)
180.degrees
// res9: Angle = Angle(3.141592653589793)
3.14.radians
// res10: Angle = Angle(3.14)
```

Saturation and lightness are both `Doubles` that should be between 0.0 and 1.0. Values outside this range will be converted to the closest number within the range.

We can now create colors using the HSL representation.

```
Color.hsl(0.degrees, 0.8, 0.6) // A pastel red
```

To view this color we can render it in a picture. See fig. 3.9 for an example.

3.4.3 Manipulating Colors

The effectiveness of a composition often depends as much on the relationships between colors as the actual colors used. Colors have several methods that allow us to create a new color from an existing one. The most commonly used ones are:

- `spin`, which rotates the hue by an `Angle`;
- `saturate` and `desaturate`, which respectively add and subtract a `Normalised` value from the color; and
- `lighten` and `darken`, which respectively add and subtract a `Normalised` value from the lightness.

For example,

```
Image.circle(100)
     .fillColor(Color.red)
     .beside(Image.circle(100).fillColor(Color.red.spin(15.degrees)))
```

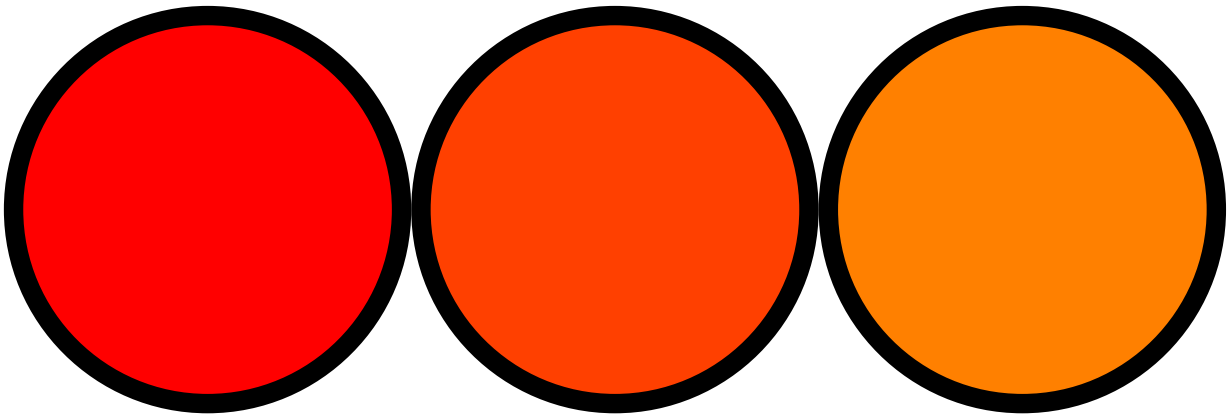

Figure 3.10: Three circles, starting with Color.red and spinning by 15 degrees for each successive circle

```
    .beside(Image.circle(100).fillColor(Color.red.spin(30.degrees)))
    .strokeWidth(5.0)
```

produces fig. 3.10.

Here's a similar example, this time manipulating saturation and lightness, shown in fig. 3.11.

```
Image.circle(40)
   .fillColor(Color.red.darken(0.2.normalized))
   .beside(Image.circle(40).fillColor(Color.red))
   .beside(Image.circle(40).fillColor((Color.red.lighten(0.2.normalized))))
   .above(Image.rectangle(40, 40).fillColor(Color.red.desaturate(0.6.normalized))
               .beside(Image.rectangle(40,40).fillColor(Color.red.desaturate(0.3.normalized)))
               .beside(Image.rectangle(40,40).fillColor(Color.red)))
```

3.4.4 Transparency

We can also add a degree of transparency to our colors, by adding an *alpha* value. An alpha value of 0.0 indicates a completely transparent color, while a color with an alpha of 1.0 is completely opaque. The methods `Color.rgba` and `Color.hsla` have a fourth parameter that is a `Normalized` alpha value. We can also create a new color with a different transparency by using the `alpha` method on a color. Here's an example, shown in fig. 3.12.

```
Image.circle(40)
    .fillColor(Color.red.alpha(0.5.normalized))
    .beside(Image.circle(40).fillColor(Color.blue.alpha(0.5.normalized)))
    .on(Image.circle(40).fillColor(Color.green.alpha(0.5.normalized)))
```

Exercises

Analogous Triangles

Create three triangles, arranged in a triangle, with analogous colors. Analogous colors are colors that are similar in hue. See a (fairly elaborate) example in fig. 3.13.

See the solution

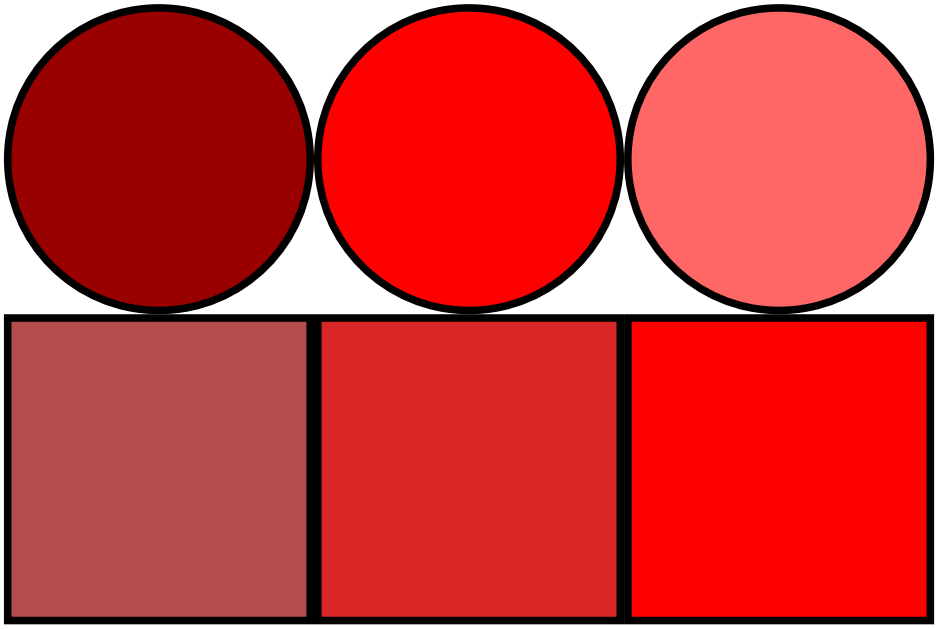

Figure 3.11: The top three circles show the effect of changing lightness, and the bottom three squares show the effect of changing saturation.

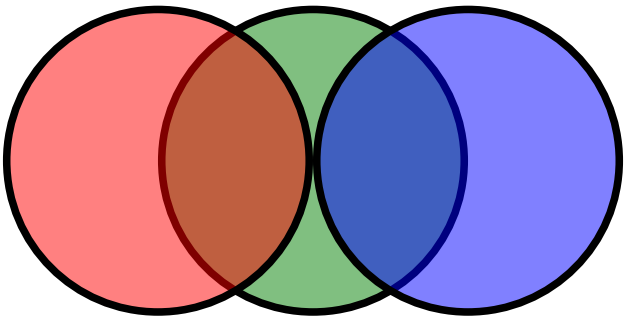

Figure 3.12: Circles with alpha of 0.5 showing transparency

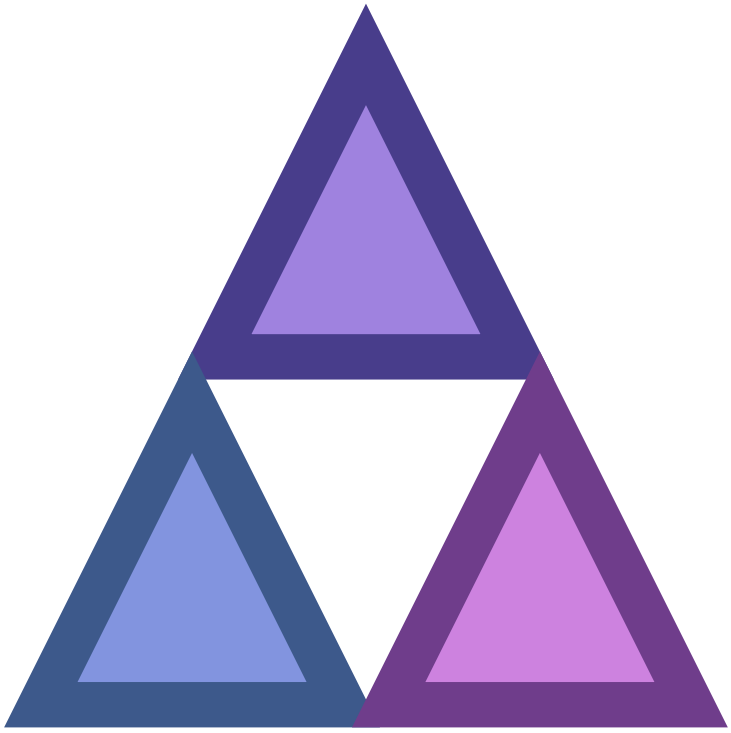

Figure 3.13: Analogous triangles. The colors chosen are variations on `darkSlateBlue`

3.5 Exercises

3.5.1 Compilation Target

Create a line drawing of an archery target with three concentric scoring bands, as shown in fig. 3.14.

For bonus credit add a stand so we can place the target on a range, as shown in fig. 3.15.

See the solution

3.5.2 Stay on Target

Colour your target red and white, the stand in brown (if applicable), and some ground in green. See fig. 3.16 for an example.

See the solution

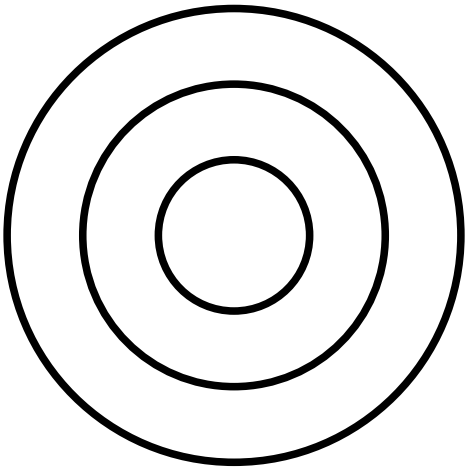

Figure 3.14: Simple archery target

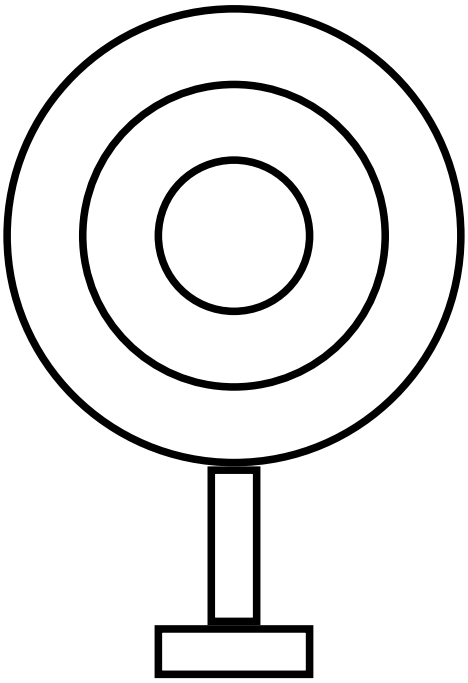

Figure 3.15: Archery target with a stand

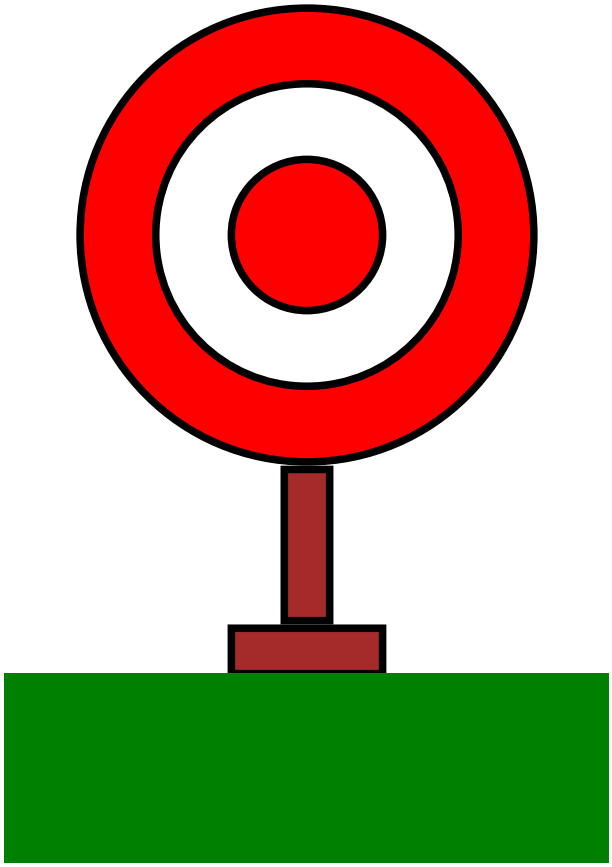

Figure 3.16: Colour archery target

Chapter 4

Writing Larger Programs

We're getting to the point where it's inconvenient to type programs into the console. In this chapter we'll learn about two tools for writing larger programs:

- saving programs to a file so we don't have to type code over and over again;
- giving names to values so we can reuse them.

> If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.
>
> ```
> import doodle.core._
> import doodle.image._
> import doodle.image.syntax._
> import doodle.image.syntax.core._
> import doodle.java2d._
> ```

4.1 Working Within the Console

Your text editor or IDE will allow you to save code to a file, but we need to save files in the right place so the Scala compiler can find them. If you're working from the Doodle template you should save your code in the directory `src/main/scala/`.

How do we use code that we saved to a file from the console? There is a special command, that only works from the console, that allows us to run code saved in a file. This command is called `:paste`[1]. We follow `:paste` with the name of the file we want to run. For example, if we save in the file `src/main/scala/Example.scala` the expression

```
Image.circle(100).fillColor(Color.paleGoldenrod).strokeColor(Color.indianRed)
```

we can then run this code by writing at the console

[1] There is also a command called `:load` which works in a slightly different way to `:paste`. It compiles and runs each line in the file on its own, while `:paste` compiles and runs the whole file in one go. They have subtly different semantics. The way `:paste` works is closer to how Scala code works outside the console, so we'll use it in preference to `:load`.

```
:paste src/main/scala/Example.scala
// res0: doodle.core.Image = ContextTransform(<function1>,ContextTransform(<function1>,Circle(100.0)
    ))
```

Note the value has been given the name `res0` in the example above. If you're following along, the name in your console might end with a different number depending on what you've already typed into the console. We can draw the image by evaluating `res0.draw` (or the correct name for your console).

4.1.1 Tips for Using the Console

Here are a few tips for using the console more productively:

- If you press the up arrow you'll get the last thing you typed into the console. Handy to avoid having to type in those long file names over and over again! You can press up multiple times to go through the history of your interactions at the console.
- You can press the `Tab` key to get the console to suggest completions for code, but unfortunately not file names, you're typing. For example, if you type `Stri` and then press `Tab`, the console will show possible completions. Type `Strin` and the console will complete `String` for you.

> Once we start saving code to a file, we'll likely find the compiler doesn't like our code next time we start SBT. Read the next section to see how we can fix this problem.

4.2 Coding Outside the Console

The code we've been writing inside the console will cause problems running outside the console. For example, put the following code into `Example.scala` in the `src/main/scala`.

```
Image.circle(100).fillColor(Color.paleGoldenrod).strokeColor(Color.indianRed)
```

Now restart SBT and try to enter the console. You should see an error similar to

```
[error] src/main/scala/Example.scala:1: expected class or object definition
[error] circle(100) fillColor Color.paleGoldenrod strokeColor Color.indianRed
[error] ^
[error] one error found
```

You'll see something similar if you're using an IDE.

The problem is this:

- Scala is attempting to compile all our code before the console starts; and
- there are restrictions on code written in files that don't apply to code written directly in the console.

We need to know about these restrictions and change how we write code in files accordingly.

The error message gives us some hint: `expected class or object definition`. We don't yet know what a class is, but we do know about objects—all values are objects. In Scala all code in a file must be written inside an object or class. We can easily define an object by wrapping an expression like the below.

```
object Example {
  Image.circle(100).fillColor(Color.paleGoldenrod).strokeColor(Color.indianRed).draw()
}
```

Now the code won't compile for a different reason. You should see a lot of errors similar to

```
[error] doodle/shared/src/main/scala/doodle/examples/Example.scala:1: not found: value Image
[error]   Image.circle(100).fillColor(Color.paleGoldenrod).strokeColor(Color.indianRed).draw()
[error]   ^
```

The compiler is saying that we've used a name, `circle`, but the compiler doesn't know what value this name refers to. It will have a similiar issue with `Color` in the code above. We'll talk in more details about names in just a moment. Right now let's tell the compiler where it can find the values for these names by adding some `import` statements. The name `Color` is found inside a *package* called `doodle.core`, and the name `circle` is within the object `Image` that is in `doodle.image`. We can tell the compiler to use all the name in `doodle.core`, and all the names in `doodle.image` by writing

```
import doodle.core._
import doodle.image._
```

There are a few other names that the compiler will need to find for the complete code to work. We can import these with the lines

```
import doodle.image.syntax._
import doodle.image.syntax.core._
import doodle.java2d._
```

We should place all these imports at the top of the file, so the complete code looks like

```
import doodle.core._
import doodle.image._
import doodle.image.syntax._
import doodle.image.syntax.core._
import doodle.java2d._

object Example {
  Image.circle(100).fillColor(Color.paleGoldenrod).strokeColor(Color.indianRed).draw()
}
```

With this in place the code should compile without issue.

Now when we go to the console within SBT we can refer to our code using the name, `Example`, that we've given it.

```
Example // draws the image
```

Exercise

If you haven't done so already, save the code above in the file `src/main/scala/Example.scala` and check that the code compiles and you can access it from the console.

4.3 Names

In the previous section we introduced a lot of new concepts. In this section we'll explore one of those concepts: naming values.

We use names to refer to things. For example, "Professeur Emile Perrot" refers to a very fragrant rose variety, while "Cherry Parfait" is a highly disease resistant variety but barely smells at all. Much ink has been spilled, and many a chin stroked, on how exactly this relationship works in spoken language. Programming languages are much more constrained, which allows us to be much more precise: names refer to values. We will sometimes say names are *bound* to values, or a name introduces a *binding*. Wherever we would write out a value we can instead use its name, if the value has a name. In other words, a name evaluates to the value it refers to. This naturally raises the question: how do we give names to values? There are several ways to do this in Scala. Let's see a few.

4.3.1 Object Literals

We have already seen an example of declaring an object literal.

```
object Example {
  Image.circle(100).fillColor(Color.paleGoldenrod).strokeColor(Color.indianRed).draw()
}
```

This is a literal expression, like other literals we've seen so far, but in this case it creates an object with the name `Example`. When we use the name `Example` in a program it evaluates to that object.

```
Example
// Example.type = Example$@76c39258
```

Try this in the console a few times. Do you notice any difference in uses of the name? You might have noticed that the *first* time you entered the name `Example` a picture was drawn, but on subsequent uses this didn't happen. The first time we use an object's name the body of the object is evaluated and the object is created. On subsequent uses of the name the object already exists and is not evaluated again. We can tell there is a difference in this case because the expression inside the object calls the `draw` method. If we replaced it with something like `1 + 1` (or just dropped the call to `draw`) we would not be able to tell the difference. We'll have plenty more to say about this in a later chapter.

We might wonder about the type of the object we've just created. We can ask the console about this.

```
:type Example
// Example.type
```

The type of `Example` is `Example.type`, a unique type that no other value has.

4.3.2 `val` Declarations

Declaring an object literal mixes together object creation and defining a name. It would be useful if we could separate the two, so we could give a name to a pre-existing object. A `val` declaration allows us to do this.

We use `val` by writing

```
val <name> = <value>
```

replacing <name> and <value> with the name and the expression evaluating to the value respectively. For example

```
val one = 1
val anImage = Image.circle(100).fillColor(Color.red)
```

These two declarations define the names `one` and `anImage`. We can use these names to refer to the values in later code.

```
one
// res0: Int = 1
anImage
// res1: Image = FillColor(
//   Circle(100.0),
//   RGBA(
//     UnsignedByte(127),
//     UnsignedByte(-128),
//     UnsignedByte(-128),
//     Normalized(1.0)
//   )
// )
```

4.3.3 Declarations

We've talked about declarations and definitions above. It's now time to be precise about what these terms mean, and to look in a bit more depth at the differences between `object` and `val`.

We already know about expressions. They are a part of a program that evaluates to a value. A *declaration* or *definition* is another part of a program, but do not evaluate to a value. Instead they give a name to something—not always to a value as you can declare types in Scala, though we won't spend much time on this. Both `object` and `val` are declarations.

One consequence of declarations being separate from expressions is we can't write program like

```
val one = ( val aNumber = 1 )
```

because `val aNumber = 1` is not an expression and thus does not evaluate to a value.

We can however write

```
val aNumber = 1
// aNumber: Int = 1
val one = aNumber
// one: Int = 1
```

4.3.4 The Top-Level

It seems a bit unsatisfactory to have both `object` and `val` declarations, as they both give names to values. Why not just have `val` for declaring names, and make `object` just create objects without naming them? Can you declare an object literal without a name?

See the solution

Scala distinguishes between what is called the *top-level* and other code. Code at the top-level is code that doesn't have any other code wrapped around. In other words it is something we can write in a file and Scala will compile without having to wrap it in an `object`.

We've seen that expressions aren't allowed at the top-level. Neither are `val` definitions. Object literals, however, are.

This distinction is a bit annoying. Some other languages don't have this restriction. In Scala's case it comes about because Scala builds on top of the Java Virtual Machine (JVM), which was designed to run Java code. Java makes a distinction between top-level and other code, and Scala is forced to make this distinction to work with the JVM. The Scala console *doesn't* make this top-level distinction (we can think of everything written in the console being wrapped in some object) which can lead to confusion when we first start using Scala.

If an object literal is allowed at the top-level, but a `val` definition is not, does this mean we can declare a `val` inside an object literal? If we can declare a `val` inside an object literal, can we later refer to that name?

See the solution

4.3.5 Scope

If you did the last exercise (and you did, didn't you?) you'll have seen that a name declared inside an object can't be used outside the object without also referring to the object that contains the name. Concretely, if we declare

```
object Example {
  val hi = "Hi!"
}
```

we can't write

```
hi
// error: not found: value hi
```

We must tell Scala to look for `hi` inside `Example`.

```
Example.hi
// res8: String = "Hi!"
```

We say that a name is *visible* in the places where it can be used without qualification, and we call the places where a name is visible its *scope*. So using our fancy-pants new terminology, `hi` is not visible outside of `Example`, or alternatively `hi` is not in scope outside of `Example`.

How do we work out the scope of a name? The rule is fairly simple: a name is visible from the point it is declared to the end of the nearest enclosing braces (braces are `{` and `}`). In the example above `hi` is enclosed by the braces of `Example` and so is visible there. It's not visible elsewhere.

We can declare object literals inside object literals, which allows us to make finer distinctions about scope. For example in the code below

```
object Example1 {
  val hi = "Hi!"

  object Example2 {
    val hello = "Hello!"
  }
}
```

`hi` is in scope in `Example2` (`Example2` is defined within the braces that enclose `hi`). However the scope of `hello` is restricted to `Example2`, and so it has a smaller scope than `hi`.

What happens if we declare a name within a scope where it is already declared? This is known as *shadowing*. In the code below the definition of `hi` within `Example2` shadows the definition of `hi` in `Example1`

```
object Example1 {
  val hi = "Hi!"

  object Example2 {
    val hi = "Hello!"
  }
}
```

Scala let's us do this, but it is generally a bad idea as it can make code very confusing.

We don't have to use object literals to create new scopes. Scala allows us to create a new scope just about anywhere by inserting braces. So we can write

```
object Example {
  val good = "Good"

  // Create a new scope
  {
    val morning = good ++ " morning"
    val toYou = morning ++ " to you"
  }

  val day = good ++ " day, sir!"
}
```

`morning` (and `toYou`) is declared within a new scope. We have no way to refer to this scope from the outside (it has no name) so we cannot refer to `morning` outside of the scope where it is declared. If we had some secrets that we didn't want the rest of the program to know about this is one way we could hide them.

The way nested scopes work in Scala is called *lexical scoping*. Not all languages have lexical scoping. For example, Ruby and Python do not, and Javascript has only recently acquired lexical scoping. It is the authors' opinion that creating a language without lexical scope is an idea on par with eating a bushel of Guatemalan insanity peppers and then going to the toilet without washing your hands.

Exercises

Test your understanding of names and scoping by working out the value of `answer` in each case below.

```
val a = 1
val b = 2
val answer = a + b
```

See the solution

```
object One {
  val a = 1

  object Two {
    val a = 3
    val b = 2
  }
```

```
  object Answer {
    val answer = a + Two.b
  }
}
```

See the solution

```
object One {
  val a = 5
  val b = 2

  object Answer {
    val a = 1
    val answer = a + b
  }
}
```

See the solution

```
object One {
  val a = 1
  val b = a + 1
  val answer = a + b
}
```

See the solution

```
object One {
  val a = 1

  object Two {
    val b = 2
  }

  val answer = a + b
}
```

See the solution

```
object One {
  val a = b - 1
  val b = a + 1

  val answer = a + b
}
```

See the solution

4.4 Abstraction

We've learned a lot about names in the previous section. If we want to use fancy programmer words, we could say that *names abstract over expressions*. This usefully captures the essence of what defining names does, so let's decode the programmer-talk.

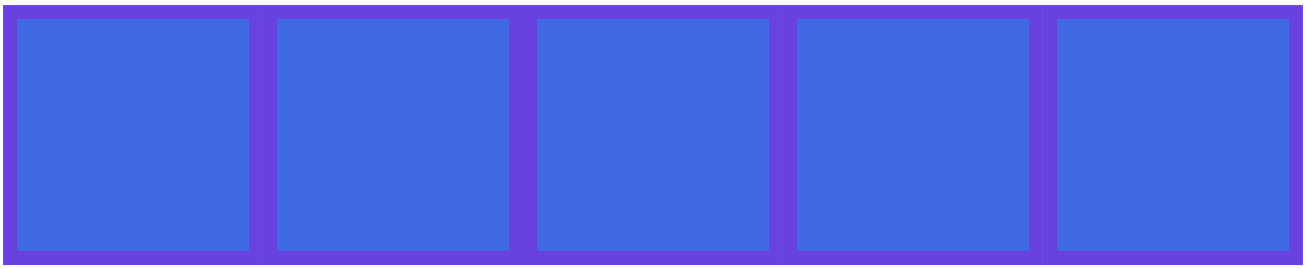

Figure 4.1: Five boxes filled with Royal Blue

To abstract means to remove unnecessary details. For example, numbers are an abstraction. The number "one" is never found in nature as a pure concept. It's always one object, such as one apple, or one copy of Creative Scala. When doing arithmetic the concept of numbers allows us to abstract away the unnecessary detail of the exact objects we're counting and manipulate the numbers on their own.

Similarly a name stands in for an expression. An expression tells us how to construct a value. If that value has a name then we don't need to know anything about how the value is constructed. The expression can have arbitrary complexity, but we don't have to care about this complexity if we just use the name. This is what it means when we say that names abstract over expressions. Whenever we have an expression we can substitute a name that refers to the same value.

Abstraction makes code easier to read and write. Let's take as an example creating a sequence of boxes like shown in fig. 4.1.

We can write out a single expression that creates the picture.

```
Image.rectangle(40, 40)
     .strokeWidth(5.0)
     .strokeColor(Color.royalBlue.spin(30.degrees))
     .fillColor(Color.royalBlue)
     .beside(
       Image.rectangle(40, 40)
         .strokeWidth(5.0)
         .strokeColor(Color.royalBlue.spin(30.degrees))
         .fillColor(Color.royalBlue)
     ).beside(
        Image.rectangle(40, 40)
          .strokeWidth(5.0)
          .strokeColor(Color.royalBlue.spin(30.degrees))
          .fillColor(Color.royalBlue)
     ).beside(
        Image.rectangle(40, 40)
             .strokeWidth(5.0)
             .strokeColor(Color.royalBlue.spin(30.degrees))
            .fillColor(Color.royalBlue)
     ).beside(
        Image.rectangle(40, 40)
             .strokeWidth(5.0)
             .strokeColor(Color.royalBlue.spin(30.degrees))
             .fillColor(Color.royalBlue)
     )
```

In this code it is difficult to see the simple pattern within. Can you really tell at a glance that all the rectangles are exactly the same? If we make the abstraction of naming the basic box the code becomes much easier to read.

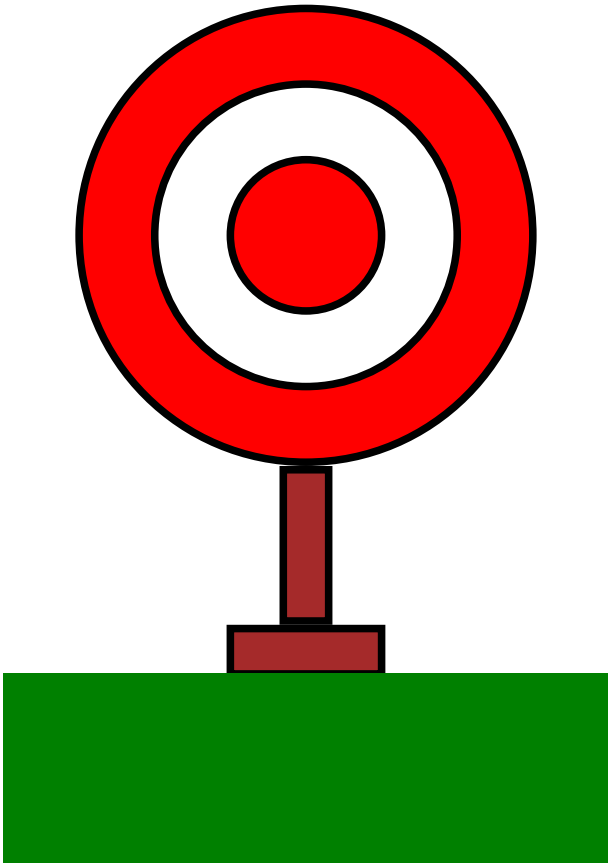

Figure 4.2: The Archery Target

```
val box =
  Image.rectangle(40, 40)
       .strokeWidth(5.0)
       .strokeColor(Color.royalBlue.spin(30.degrees))
       .fillColor(Color.royalBlue)

box.beside(box).beside(box).beside(box).beside(box)
```

Now we can easily see how the box is made, and easily see that the final picture is that box repeated five times.

Exercises

Archery Again

Let's return to the archery target we created in an earlier chapter, shown in fig. 4.2.

Last time we created the image we didn't know how to name values, so we can to write one large expression. This time around, give the components of the image names so that it is easier for someone else to understand how the image is constructed. You'll have to use your own taste to decide what parts should be named and what parts don't warrant names of their own.

See the solution

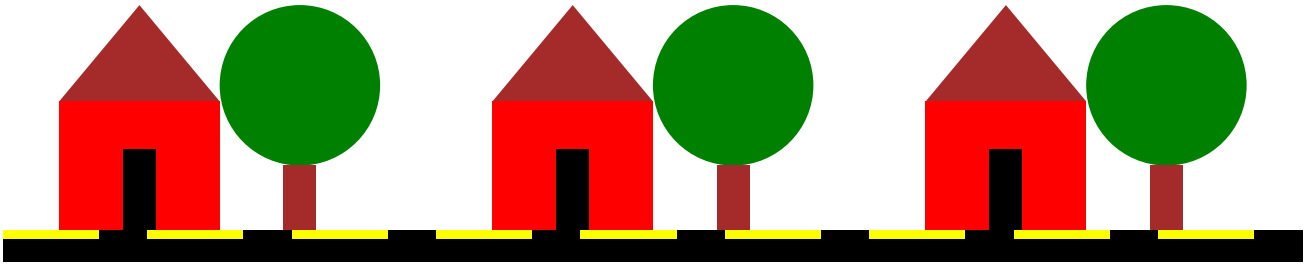

Figure 4.3: A Street Scene

Streets Ahead

For a more compelling use of names, create a street scene like that shown in fig. 4.3. By naming the individual components of the image you should be able to avoid a great deal of repetition.

See the solution

4.5 Packages and Imports

When we changed our code to compile we had to add many *import statements* to it. In this section we learn about them.

We've seen that one name can shadow another. This can cause problems in larger programs as many parts of a program many want to put a common name to different uses. We can create scopes to hide names from the outside, but we must still deal with names defined at the top-level.

We have the same problem in natural language. For example, if both your brother and friend were called "Ziggy" you would have to qualify which one you meant when you used their name. Perhaps you could tell from context, or perhaps your friend was "Ziggy S" and your brother was just "Ziggy".

In Scala we can use *packages* to organise names. A package creates a scope for names defined at the top-level. All top-level names within the same package are defined in the same scope. To bring names in a package into another scope we must *import* them.

Creating a package is simple: we write

```
package <name>
```

at the top of the file, replace `<name>` with the name of our package.

When we want to use names defined in a package we use an `import` statement, specifying the package name followed by `_` for all names, or the just the name we want if we only want one or a few names.

Here's an example.

You can't define packages in the console. To get the following code to work you must put the code within the package `example` into a file and compile it.

Let's start by defining some names within a package.

```
package example

object One {
  val one = 1
}

object Two {
```

```
  val two = 2
}

object Three {
  val three = 3
}
```

Now to bring these names into scope we must import them. We could import just one name.

```
import example.One

One.one
```

Or both `One` and `Two`.

```
import example.{One, Two}

One.one + Two.two
```

Or all the names in `example`.

```
import example._

One.one + Two.two + Three.three
```

In Scala we can also import just about anything that defines a scope, including objects. So the following code brings `one` into scope.

```
import example.One._

one
```

4.5.1 Package Organisation

Packages stop top-level names from colliding, but what about collisions between package names? It's common to organise packages in a hierarchy, which helps to avoid collisions. For example, in Doodle the package `core` is defined within the package `doodle`. When we use the statement

```
import doodle.core._
```

we're indicating we want the package `core` within the package `doodle`, and not some other package that might be called `core`.

Chapter 5

The Substitution Model of Evaluation

We need to build a mental model of how Scala expressions are evaluated so we can understand what our programs are doing. We've been getting by with an informal model so far. In this section we make our model a bit more formal by learning about the *substitution model* of evaluation. Like many things in programming we're using some fancy words for a simple concept. In this case you've probably already learned about substitution in high school algebra, and we're just taking those ideas into a new context.

> If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.
>
> ```
> import doodle.core._
> import doodle.image._
> import doodle.image.syntax._
> import doodle.image.syntax.core._
> import doodle.java2d._
> ```

5.1 Substitution

Substitution says that wherever we see an expression we can replace it with the value it evaluates to. For example, where we see

```
1 + 1
```

we can replace it with 2. This in turn means when we see a compound expression such as

```
(1 + 1) + (1 + 1)
```

we can substitute 2 for `1 + 1` giving

```
2 + 2
```

which evaluates to 4.

This type of reasoning is what we do in high school algebra when we simplify an expression. Naturally computer science has fancy words for this process. In addition to substitution, we can call this *reducing an expression*, or *equational reasoning*.

Substitution gives us a way to reason about our programs, which is another way of saying "working out what they do". We can apply substitution to just about any expression we've seen so far. It's easier to use examples that work with numbers and strings, rather than images, here so we'll return to an example we saw in an earlier chapter:

```
1 + ("Moonage daydream".indexOf("N"))
```

In the previous example we were a bit fast-and-loose. Here we will be a bit more precise to illustrate the steps the computer would have to go through. We are trying to emulate the computer, after all.

The expression containing the + consists of two sub-expressions, `1` and `("Moonage daydream".indexOf("N"))`. We have to decide which to evaluate first: the left or the right. Let's arbitrarily choose the right sub-expression (we'll return to this choice later.)

The sub-expression `("Moonage daydream".indexOf("N"))` again consists of two sub-expressions, `"Moonage daydream"` and `"N"`. Let's again evaluate the right-hand first, remembering that literal expressions are not values so they must be evaluated.

The literal `"N"` evaluates to the value `"N"`. To avoid this confusion let's write the value as `|"N"|`. Now we can substitute the value for the expression given in our first steps

```
1 + ("Moonage daydream".indexOf(|"N"|))
```

Now we can evaluate the left-hand side of the sub-expression, substituting the literal expression `"Moonage daydream"` with its value `|"Moonage daydream"|`. This gives us

```
1 + (|"Moonage daydream"|.indexOf(|"N"|))
```

Now we're in a position to evaluate the entire expression `(|"Moonage daydream"|.indexOf(|"N"|))`, which evaluates to `|-1|` (again differentiating the integer value from the literal expression by using a vertical bar). Once again we perform substitution and now we have

```
1 + |-1|
```

Now we should evaluate the left-hand side literal 1, giving `|1|`. Perform substitution and we get

```
|1| + |-1|
```

Now we can evaluate the entire expression, giving

```
|0|
```

We can ask Scala to evaluate the whole expression to check our work.

```
1 + ("Moonage daydream".indexOf("N"))
// res4: Int = 0
```

Correct!

There are some observations we might make at this point:

- doing substitution rigorously like a computer might involve a lot of steps;
- the shortcut evaluation you probably did in your head probably got to the correct answer; and
- our seemingly arbitrary choice to do evaluation from right-to-left got us the correct answer.

Did we somehow manage to choose the same substitution order that Scala uses (no we didn't, but we haven't investigated this yet) or does it not really matter what order we choose? When exactly can we take shortcuts and still reach the right result, like we did in the first example with addition? We will investigate these questions in just a moment, but first let's talk about how substitution works with names.

5.1.1 Names

The substitution rule for names is to substitute the name with the value it refers to. We've already been using this rule implicitly. Now we're just formalising it.

For example, given the code

```
val name = "Ada"
name ++ " " ++ "Lovelace"
```

we can apply substitution to get

```
"Ada" ++ " " ++ "Lovelace"
```

which evaluates to

```
"Ada Lovelace"
```

We can use names to be a bit more formal with our substitution process. Returning to our first example

```
1 + 1
```

we can give this expression a name:

```
val two = 1 + 1
```

When we see a compound expression such as

```
(1 + 1) + (1 + 1)
```

substitution tells us we can substitute `two` for `1 + 1` giving

```
two + two
```

Remember when we worked through the expression

```
1 + ("Moonage daydream".indexOf("N"))
```

we broke it into sub-expressions which we then evaluated and substituted. Using words, this was quite convoluted. With a few `val` declarations we can make this both more compact and easier to see. Here's the same expression broken into it's components.

```
val a = 1
val b = "Moonage daydream"
val c = "N"
val d = b.indexOf(c)
val e = a + d
```

If we (at this point, arbitrarily) define that evaluation occurs from top-to-bottom we can experiment with different ordering to see what difference they make.

For example,

```
val c = "N"
val b = "Moonage daydream"
val a = 1
val d = b.indexOf(c)
val e = a + d
```

achieves the same result as before. However we can't use

```
val e = a + d
val a = 1
val b = "Moonage daydream"
val c = "N"
val d = b.indexOf(c)
```

because e depends on a and d, and in our top-to-bottom ordering a and d have yet to be evaluated. We might rightly claim that this is a bit silly to even attempt. The complete expression we're trying to evaluate is e but a to d are sub-expressions of e, so of course we have to evaluate the sub-expressions before we evaluate the expression.

5.2 Order of Evaluation

We're now ready to tackle the question of order-of-evaluation. We might wonder if the order of evaluation even matters? In the examples we've looked at so far the order doesn't seem to matter, except for the issue that we cannot evaluate an expression before it's sub-expressions.

To investigate these issues further we need to introduce a new concept. So far we have almost always dealt with *pure* expressions. These are expressions that we can freely substitute in any order without issue[1].

Impure expressions are those where the order of evaluation matters. We have already used one impure expression, the method `draw`. If we evaluate

```
Image.circle(100).draw
Image.rectangle(100, 50).draw
```

and

```
Image.rectangle(100, 50).draw
Image.circle(100).draw
```

the windows containing the images will appear in different orders. Hardly an exciting difference, but it *is* a difference, which is the point.

The key distinguishing feature of impure expressions is that their evaluation causes some change that we can see. For example, evaluating `draw` causes an image to be displayed. We call these observable changes *side effects*, or just *effects* for short. In a program containing side effects we cannot freely use substitution. However we can use side effects to investigate the order of evaluation. Our tool for doing so will be the `println` method.

The `println` method displays text on the console (a side effect) and evaluates to unit. Here's an example:

[1] This is not entirely true. There are some corner cases where the order of evaluation does make a difference even with pure expressions. We're not going to worry about these cases here. If you're interested in learning more, and this is interesting and useful stuff, you can read up on "eager evaluation" and "lazy evaluation".

```
println("Hello!")
// Hello!
```

The side-effect of `println`—printing to the console—gives us a convenient way to investigate the order of evaluation. For example, the result of running

```
println("A")
// A
println("B")
// B
println("C")
// C
```

indicates to us that expressions are evaluated from top to bottom. Let's use `println` to investigate further.

Exercises

No Substitute for Println

In a pure program we can give a name to any expression and substitute any other occurrences of that expression with the name. Concretely, we can rewrite

```
(2 + 2) + (2 + 2)
```

to

```
val a = (2 + 2)
a + a
```

and the result of the program doesn't change.

Using `println` as an example of an impure expression, demonstrates that this is *not* the case for impure expressions, and hence we can say that impure expressions, or side effects, break substitution.

See the solution

Madness to our Methods

When we introduced scopes we also introduced block expressions, though we didn't call them that at the time. A block is created by curly braces (`{}`). It evaluates all the expressions inside the braces. The final result is the result of the last expression in the block.

```
// Evaluates to three
{
  val one = 1
  val two = 2
  one + two
}
// res12: Int = 3
```

We can use block expressions to investigate the order in which method parameters are evaluated, by putting `println` expression inside a block that evaluates to some other useful value.

For example, using `Image.rectangle` or `Color.hsl` and block expressions, we can determine if Scala evaluates method parameters in a fixed order, and if so what that order is.

Note that you can write a block compactly, on one line, by separating expressions with semicolons (`;`). This is generally not good style but might be useful for these experiments. Here's an example.

```
// Evaluates to three
{ val one = 1; val two = 2; one + two }
// res13: Int = 3
```

See the solution

The Last Order

In what order are Scala expressions evaluated? Perform whatever experiments you need to determine an answer to this question to your own satisfaction. You can reasonably assume that Scala uses consistent rules across all expressions. There aren't special cases for different expressions.

See the solution

5.3 Local Reasoning

We've seen that the order of evaluation is only really important when we have side effects. For example, if the following expressions produce side effects

```
disableWarheads()
launchTheMissles()
```

we really want to ensure that the expressions are evaluated top to bottom so the warheads are disabled before the missles are launched.

All useful programs must have some effect, because effects are how the program interacts with the outside world. The effect might just be printing out something when the program has finished, but it's still there. Minimising side effects is a key goal of functional programming so we will spend a few more words on this topic.

Substitution is really easy to understand. When the order of evaluation doesn't matter it means any other code cannot change the meaning of the code we're looking at. `1 + 1` is always 2, no matter what other code we have in our program, but the effect of `launchTheMissles()` depends on whether we have already disabled the warheads or not.

The upshot of this is that pure code can be understood in isolation. Since no other code can change its meaning, if we're only interested in one fragment we can ignore the rest of the code. The meaning of impure code, on the other hand, depends on all the code that will have run before it is evaluated. This property is known as *local reasoning*. Pure code has it, but impure code does not.

As programs get larger it becomes harder and harder to keep all the details in our head. Since the size of our head is a fixed quantity the only solution is to introduce abstraction. Remember that an abstraction is the removal of irrelevant details. Pure code is the ultimate abstraction, because it tells us that everything else is an irrelevant detail. This is one of the properties that gets functional programmers really excited: the ability to make large programs understandable. Functional programming doesn't mean avoiding effects, because all useful programs have effects. It does, however, mean controlling effects so the majority of the code can be reasoned about using the simple model of substitution.

5.3.1 The Meaning of Meaning

So far, we've talked a lot about the meaning of code, where we've taken "meaning" to mean to the result it evaluates to, and perhaps the side effects it performs.

In substitution, the meaning of a program is exactly what it evaluates to. Thus two programs are equal if they evaluate to the same result. This is precisely why side effects break substitution: the substitution model has no notion of side effects and therefore cannot distinguish two programs that differ by their effects.

There are other ways in which programs can differ. For example, one program may take longer than another to produce the same result. Again, substitution does not distinguish them.

Substitution is an abstraction, and the details it throws away are everything except for the value. Side effects, time, and memory usage are all irrelevant to substitution, but perhaps not to the people writing or running the program. There is a tradeoff here. We can employ richer models that capture more of these details, but they are much harder to work with. For most people most of the time substitution makes the right tradeoff of being dead simple to use while still being useful.

Chapter 6

Methods

We've already used methods—methods are the way we interact with objects. In this chapter we'll learn how to write our own methods.

Names allow us to abstract over expressions. Methods allow us to abstract over and *generalise* expressions. By generalisation we mean the ability to express a group of related things, in this case expressions. A method captures a template for an expression, and allows the caller to fill in parts of that template by passing the method parameters.

> If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.
>
> ```
> import doodle.core._
> import doodle.image._
> import doodle.image.syntax._
> import doodle.image.syntax.core._
> import doodle.java2d._
> ```

6.1 Methods

In a previous chapter we created the image shown in fig. 6.1 using the program

```
val box =
  Image.rectangle(40, 40).
    strokeWidth(5.0).
    strokeColor(Color.royalBlue.spin(30.degrees)).
    fillColor(Color.royalBlue)
```

Figure 6.1: Five boxes filled with Royal Blue

```
box beside box beside box beside box beside box
```

Imagine we wanted to change the color of the boxes. Right now we would have to write out the expression again for each different choice of color.

```
val paleGoldenrod = {
  val box =
    Image.rectangle(40, 40).
      strokeWidth(5.0).
      strokeColor(Color.paleGoldenrod.spin(30.degrees)).
      fillColor(Color.paleGoldenrod)

  box beside box beside box beside box beside box
}

val lightSteelBlue = {
  val box =
    Image.rectangle(40, 40).
      strokeWidth(5.0).
      strokeColor(Color.lightSteelBlue.spin(30.degrees)).
      fillColor(Color.lightSteelBlue)

  box beside box beside box beside box beside box
}

val mistyRose = {
  val box =
    Image.rectangle(40, 40).
      strokeWidth(5.0).
      strokeColor(Color.mistyRose.spin(30.degrees)).
      fillColor(Color.mistyRose)

  box beside box beside box beside box beside box
}
```

This is tedious. Each expression only differs in a minor way. It would be nice if we could capture the general pattern and allow the color to vary. We can do exactly this by declaring a method.

```
def boxes(color: Color): Image = {
  val box =
    Image.rectangle(40, 40).
      strokeWidth(5.0).
      strokeColor(color.spin(30.degrees)).
      fillColor(color)

  box beside box beside box beside box beside box
}

// Create boxes with different colors
boxes(Color.paleGoldenrod)
boxes(Color.lightSteelBlue)
boxes(Color.mistyRose)
```

Try this yourself to see that you get the same result using the method as you did writing everything out by hand.

Now that we've seen an example of declaring a method, we need to explain the syntax of methods. Next, we'll look at how to write methods, the semantics of method calls, and how they work in terms of substitution.

6.2 Method Syntax

We've already seen an example of declaring a method.

```
def boxes(color: Color): Image = {
  val box =
    Image.rectangle(40, 40).
      strokeWidth(5.0).
      strokeColor(color.spin(30.degrees)).
      fillColor(color)

  box beside box beside box beside box beside box
}
```

Let's use this as a model for understanding the syntax of declaring a method. The first part is the *keyword* `def`. A keyword is a special word that indicates something important to the Scala compiler—in this case that we're going to declare a method. We're already seen the `object` and `val` keywords.

The `def` is immediately followed by the name of the method, in this case `boxes`, in the same way that `val` and `object` are immediately followed by the name they declare. Like a `val` declaration, a method declaration is not a top-level declaration and must be wrapped in an `object` declaration (or other top-level declaration) when written in a file.

Next we have the method parameters, defined in brackets (`()`). The method parameters are the parts that the caller can "plug-in" to the expression that the method evaluates. When declaring method parameters we must give them both a name and a type. A colon (`:`) separates the name and the type. We haven't had to declare types before. Most of the time Scala will work out the types for us, a process known as *type inference*. Type inference, however, cannot infer the type of method parameters so we must provide them.

After the method parameters comes the result type. The result type is the type of the value the method evaluates to when it is called. Unlike parameter types Scala can infer the result type, but it is good practice to include it and we will do so throughout Creative Scala.

Finally, the body expression of the method calculates the result of calling the method. A body can be a block expression, as in `boxes` above, or just a single expression.

Method Declaration Syntax

The syntax for a method declaration is

```
def methodName(param1: Param1Type, ...): ResultType =
  bodyExpression
```

where

- `methodName` is the name of the method;
- the optional `param1 : Param1Type, ...` are one or more pairs of parameter name and parameter type;
- the optional `ResultType` is the type of the result of calling the method; and
- `bodyExpression` is the expression that is evaluated to yield the result of calling the method.

Exercises

Let's practice declaring methods by writing some simple examples.

Square

Write a method `square` that accepts an `Int` argument and returns the `Int` square of it's argument. (Squaring a number is multiplying it by itself.)

See the solution

Halve

Write a method `halve` that accepts a `Double` argument and returns the `Double` that is half of it's argument.

See the solution

6.3 Method Semantics

Now that we know how to declare methods, let's turn to the semantics. How do we understand a method call in terms of our substitution model?

We already know we can substitute a method call with the value it evaluates to. However we need a more fine-grained model so we can work out what this value will be. Our extended model is as follows: when we see a method call we will create a new block and within this block: bind the parameters to the respective expressions given in the method call and substitute the method body.

We can then apply substitution as usual.

Let's see a simple example. Given the method

```
def square(x: Int): Int =
  x * x
```

we can expand the method call

```
square(2)
```

by introducing a block

```
{
  square(2)
}
```

binding the parameter x to the expression 2

```
{
  val x = 2
  square(2)
}
```

and substituting the method body

```
{
  val x = 2
  x * x
}
```

We can now perform substitution as usual giving

```
{
  2 * 2
}
```

and finally

```
{
  4
}
```

Once again we see that substitution is involved but no single step was particularly difficult.

Exercise

Last time we looked at substitution we spent a lot of time investigating order of evaluation. In the description above we have decided that a method's arguments are evaluated before the body is evaluated. This is not the only possibility. We could, for example, evaluate the method's arguments only at the point they are needed. This could save us some time if a method doesn't use one of its parameters. By using our old friend `println`, determine when method parameters are evaluated in Scala.

See the solution

6.4 Conclusions

In this chapter, we learned how to write our own simple methods and we saw how to use the substitution model of evaluation to understand method calls.

We saw that methods both abstract over expressions, in the same way as names, and also generalize over expressions, allowing us to represent a group of related expressions with one name.

We wrote some interesting methods, but we still have more repeated code than is desirable (think about the repeated calls to `box` and `circle` in the exercises.) In the next chapter, we will learn how we can generalize over this using structural recursion over the natural numbers.

Chapter 7

Structural Recursion

In this chapter we see our first major pattern for structuring computations: *structural recursion over the natural numbers*. That's quite a mouthful, so let's break it down:

- By a pattern, we mean a way of writing code that is useful in lots of different contexts. We'll encounter structural recursion in many different situations throughout this book.
- By the natural numbers we mean the whole numbers 0, 1, 2, and upwards.
- By recursion we mean something that refers to itself. Structural recursion means a recursion that follows the structure of the data it is processing. If the data is recursive (refers to itself) then the structural recursion will also refer to itself. We'll see in more detail what this means in a moment.

If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.

```
import doodle.core._
import doodle.image._
import doodle.image.syntax._
import doodle.image.syntax.core._
import doodle.java2d._
```

7.1 A Line of Boxes

Let's start with an example, drawing a line or row of boxes as in fig. 7.1.

Let's define a box to begin with.

```
val aBox = Image.square(20).fillColor(Color.royalBlue)
```

Then one box in a row is just

Figure 7.1: Five boxes filled with Royal Blue

```
val oneBox = aBox
```

If we want to have two boxes side by side, that is easy enough.

```
val twoBoxes = aBox.beside(oneBox)
```

Similarly for three.

```
val threeBoxes = aBox.beside(twoBoxes)
```

And so on for as many boxes as we care to create.

You might think this is an unusual way to create these images. Why not just write something like this, for example?

```
val threeBoxes = aBox.beside(aBox).beside(aBox)
```

These two definitions are equivalent. We've chosen to write later images in terms of earlier ones to emphasise the structure we're dealing with, which is building up to structural recursion.

Writing images in this way could get very tedious. What we'd really like is some way to tell the computer the number of boxes we'd like. More technically, we would like to abstract over the expressions above. We learned in the previous chapter that methods abstract over expressions, so let's try to write a method to solve this problem.

We'll start by writing a method skeleton that defines, as usual, what goes into the method and what it evaluates to. In this case we supply an `Int count`, which is the number of boxes we want, and we get back an `Image`.

```
def boxes(count: Int): Image =
  ???
```

Now comes the new part, the *structural recursion*. We noticed that `threeBoxes` above is defined in terms of `twoBoxes`, and `twoBoxes` is itself defined in terms of `box`. We could even define `box` in terms of *no* boxes, like so:

```
val oneBox = aBox.beside(Image.empty)
```

Here we used `Image.empty` to represent no boxes.

Imagine we had already implemented the `boxes` method. We can say the following properties of `boxes` always hold, if it is correctly implemented:

- `boxes(0) == Image.empty`
- `boxes(1) == aBox.beside(boxes(0))`
- `boxes(2) == aBox.beside(boxes(1))`
- `boxes(3) == aBox.beside(boxes(2))`

The last three properties all have the same general shape. We can describe all of them, and any case for `n > 0`, with the single property `boxes(n) == aBox.beside(boxes(n - 1))`. So we're left with two properties

- `boxes(0) == Image.empty`
- `boxes(n) == aBox.beside(boxes(n-1))`

These two properties completely define the behavior of `boxes`. In fact we can implement `boxes` by converting these properties into code.

A full implementation of `boxes` is

Figure 7.2: Three stacked boxes filled with Royal Blue

```
def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox.beside(boxes(n-1))
  }
```

Try it and see what results you get! This implementation is only tiny bit more verbose than the properties we wrote above, and is our first structural recursion over the natural numbers.

At this point we have two questions to answer. Firstly, how does this `match` expression work? More importantly, is there some general principle we can use to create methods like this on our own? Let's take each question in turn.

Exercise: Stacking Boxes

Even before we get into the details of `match` expressions you should be able to modify `boxes` to produce an image like fig. 7.2.

At this point we're trying to get used to the syntax of `match`, so rather than copying and pasting `boxes` write it all out by hand again to get some practice.

See the solution

7.2 Match Expressions

In the previous section we saw the `match` expression

```
count match {
  case 0 => Image.empty
  case n => aBox.beside(boxes(n-1))
}
```

How are we to understand this new kind of expression, and write our own? Let's break it down.

The very first thing to say is that `match` is indeed an expression, which means it evaluates to a value. If it didn't, the `boxes` method would not work!

To understand what it evaluates to we need more detail. A `match` expression in general has the shape

```
<anExpression> match {
  case <pattern1> => <expression1>
  case <pattern2> => <expression2>
  case <pattern3> => <expression3>
  ...
}
```

<anExpression>, concretely `count` in the case above, is the expression that evaluates to the value we're matching against. The patterns <pattern1> and so on are matched against this value. So far we've seen two kinds of patterns:

- a literal (as in `case 0`) which matches exactly the value that literal evaluates to; and
- a wildcard (as in `case n`) which matches *anything*, and introduces a binding within the right-hand side expression.

Finally, the right-hand side expressions, <expression1> and so on, are just expressions like any other we've written so far. The entire `match` expression evaluates to the value of the right-hand side expression of the *first* pattern that matches. So when we call `boxes(0)` both patterns will match (because the wildcard matches anything), but because the literal pattern comes first the expression `Image.empty` is the one that is evaluated.

A `match` expression that checks for all possible cases is called an exhaustive match. If we can assume that `count` is always greater or equal to zero, the `match` in `boxes` is exhaustive.

Once we're comfortable with `match` expressions we need to look at the structure of the natural numbers before we can explain structural recursion over them.

Exercises

Guess the Result

Let's check our understanding of match by guessing what each of the following expressions evaluates to, and why.

```
"abcd" match {
  case "bcde" => 0
  case "cdef" => 1
  case "abcd" => 2
}

1 match {
  case 0 => "zero"
  case 1 => "one"
  case 1 => "two"
}

1 match {
  case n => n + 1
  case 1 => 1000
}

1 match {
  case a => a
  case b => b + 1
  case c => c * 2
}
```

See the solution

No Match

What happens if no pattern matches in a `match` expression? Take a guess, then write a `match` expression that fails to match and see if you managed to guess correctly. (At this point we have no reason to expect any particular behavior so any reasonable guess will do.)

See the solution

7.3 The Natural Numbers

The natural numbers are the whole numbers, or integers, greater than or equal to zero. In other words the numbers 0, 1, 2, 3, ... (Some people define the natural numbers as starting at 1, not 0. It doesn't greatly matter for our purposes which definition you choose, but here we'll assume they start at 0.)

One interesting property of the natural numbers is that we can define them recursively. That is, we can define them in terms of themselves. This kind of circular definition seems like it would lead to nonsense. We avoid this by including in the definition a *base case* that ends the recursion. Concretely, the definition is:

A natural number n is

- 0; or
- 1 + `m`, where `m` is a natural number.

The case for 0 is the base case, whilst the other case is recursive as it defines a natural number n in terms of a natural number `m`. Because `m` is always smaller than n, and the base case is the smallest possible natural number, this definition defines all of the natural numbers.

Given a natural number, say, 3, we can break it down using the definition above as follows:

3 = 1 + 2 = 1 + (1 + 1) = 1 + (1 + (1 + 0))

We use the recursive rule to expand the equation until we cannot use it any more. We then use the base case to stop the recursion.

7.4 Structural Recursion

Now onto structural recursion. The structural recursion pattern for the natural numbers gives us two things:

- a reusable code skeleton for processing any natural number; and
- the guarantee that we can use this skeleton to implement *any* computation on natural numbers.

Remember we wrote `boxes` as

```
def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox.beside(boxes(n-1))
  }
```

When we developed `boxes` we just seemed to stumble upon this pattern. Here we see that this pattern follows directly from the definition of the natural numbers. Remember the recursive definition of the natural numbers: a natural number n is

- 0; or
- 1 + `m`, where `m` is a natural number.

The patterns in the `match` expression exactly match this definition. The expression

```
count match {
  case 0 => ???
  case n => ???
}
```

means we're checking `count` for two cases, the case when `count` is 0, and the case when `count` is any other natural number n (which is `1 + m`).

The right hand side of the `match` expression says what we do in each case. The case for zero is `Image.empty`. The case for n is `aBox.beside(boxes(n-1))`.

Now for the really important point. Notice that the structure of the right-hand side mirrors the structure of the natural number we match. When we match the base case 0, our result is the base case `Image.empty`. When we match the recursive case n the structure of the right hand side matches the structure of the recursive case in the definition of natural numbers. The definition states that n is `1 + m`. On the right-hand side we replace 1 with `aBox`, we replace + with `beside`, and we recursively call `boxes` with `m` (which is `n-1`) where the definition recurses.

```
def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox.beside(boxes(n-1))
  }
```

To reiterate, the left hand side of the `match` expression exactly matches the definition of natural numbers. The right-hand also matches the definition but we replace natural numbers with images. The image that is equivalent to zero is `Image.empty`. The image that is equivalent to `1 + m` is `aBox.beside(boxes(m))`.

This general pattern holds for anything we care to write that transforms the natural numbers into some other type. We always have a `match` expression. We always have the two patterns, corresponding to the base and recursive cases. The right hand side expressions always consist of the base case, and the recursive case which itself has a result specific substitute for 1 and +, and a recursive call for `n-1`.

Structural Recursion over Natural Numbers Pattern

The general pattern for structural recursion over the natural numbers is

```
def name(count: Int): Result =
  count match {
    case 0 => resultBase
    case n => resultUnit add name(n-1)
  }
```

where `Result`, `resultBase`, `resultUnit`, and `add` are specific to the problem we're solving. So to implement a structural recursion over the natural numbers we must

- recognise the method we're writing has a natural number as it's input;

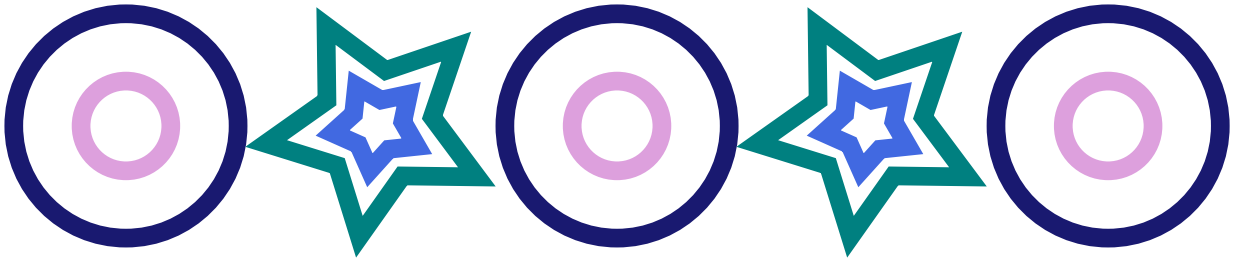

Figure 7.3: A row constructed by alternating between two different images.

- work out the result type; and
- decide what should be the base, unit, and addition for the result.

We're now ready to go explore the fun that can be had with this simple but powerful tool.

7.4.1 Proofs and Programs

If you've studied maths you have probably come across proof by induction. The general pattern of a proof by induction looks very much like the general pattern of a structural recursion over the natural numbers. This is no coincidence; there is a deep relationship between the two. We can view a structural recursion over the natural numbers as exactly a proof by induction. When we claim the ability to write any transformation on the natural numbers in terms of the structural recursion skeleton, this claim is backed up by the mathematical foundation we're implicitly using. We can also prove properties of our code by using the connection between the two: any structural recursion is implicitly defining a proof of some property.

This general connection between proofs and programs is known as the *Curry-Howard Isomorphism*.

Exercises

Three (or More) Stacks

We've seen how to create a horizontal row of boxes. Now write a method `stacks` that takes a natural number as input and creates a vertical stack of boxes.

See the solution

Alternating Images

We do more with the counter than simply using it in the recursive call. In this exercise we'll choose one `Image` when the counter is odd and a different `Image` when the counter is even. This will give us a row of alternating images as shown in fig. 7.3.

To do this we need to learn about a new method on `Int`. The *modulo* method, written `%`, returns the remainder of dividing one `Int` by another. Here are some examples.

```
4 % 2
// res1: Int = 0
3 % 2
// res2: Int = 1
2 % 2
// res3: Int = 0
1 % 2
// res4: Int = 1
```

We can see that when the first number is even the result is 0; otherwise it is 1. So we need to check is the result is 0 and act accordingly. There are a few ways to do this. Here's one example

```
(4 % 2 == 0) match {
  case true  => "It's even!"
  case false => "It's odd!"
}
// res5: String = "It's even!"
```

Here we match against the result of the comparison `(4 % 2 == 0)`. The type of this expression is `Boolean`, which has two possible values (`true` and `false`).

For `Booleans` there is special syntax that is more compact than `match`: an `if` expression. Here's the same code rewritten using `if`.

```
if(4 % 2 == 0) "It's even!"
else "It's odd!"
// res6: String = "It's even!"
```

Use whichever you are more comfortable with!

That's all the background we need. Now we can write the method we're interested in. Here's the skeleton:

```
def alternatingRow(count: Int): Image =
  ???
```

Implement the method. It's up to you what you choose for the two images used in the output.

See the solution

Getting Creative

We can use the counter to modify the image in other ways. For example we can make the color, size, or any othe property of an image depend on the counter. I have made an example in fig. 7.4, but come up with your own ideas. Implement a method

```
def funRow(count: Int): Image =
  ???
```

that generates such an image.

See the solution

Figure 7.4: A row constructed by making size and color depend on the counter.

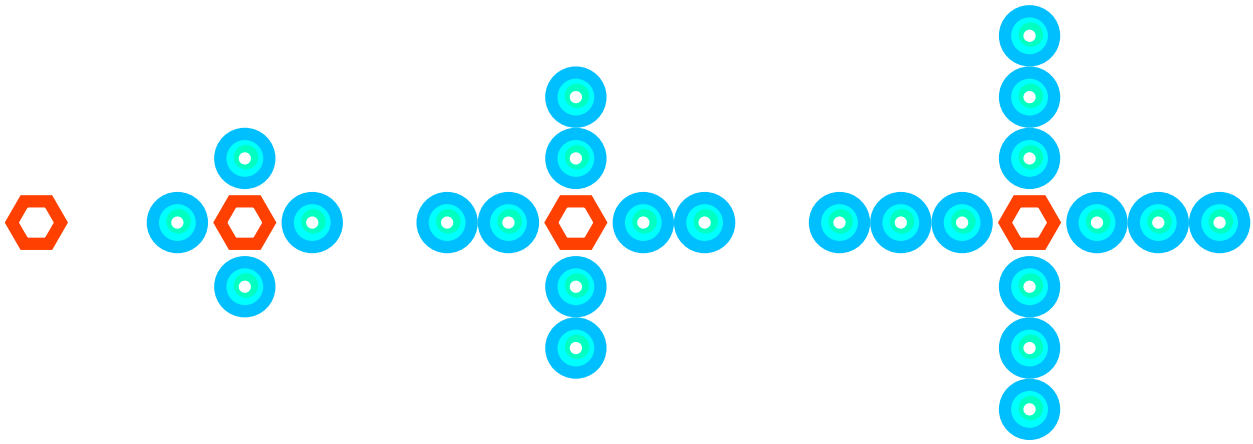

Figure 7.5: Crosses generated by `count` from 0 to 3.

Cross

Our final exercise is to create a method `cross` that will generate cross images. fig. 7.5 shows four cross images, which correspond to calling the method `cross` with `0` to 3.

The method skeleton is

```
def cross(count: Int): Image =
  ???
```

People often find this exercise harder than the preceding ones, so we'll make the process very explicit here.

Firstly, what pattern will we use to fill in the body of `cross`? Write out the pattern.

See the solution

Now we've identified the pattern we're using, we need to fill in the problem specific parts:

- the base case; and
- the unit and addition operators.

Hint: use fig. 7.5 to identify the elements above.

See the solution

Now finish the implementation of `cross`.

See the solution

7.5 Reasoning about Recursion

We're now experienced users of structural recursion over the natural numbers. Let's now return to our substitution model and see if it works with our new tool of recursion.

Recall that substitution says we can substitute the value of an expression wherever we see a value. In the case of a method call, we can substitute the body of the method with appropriate renaming of the parameters.

Our very first example of recursion was `boxes`, written like so:

```
val aBox = Image.square(20).fillColor(Color.royalBlue)

def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox.beside(boxes(n-1))
  }
```

Let's try using substitution on `boxes(3)` to see what we get.

Our first substitution is

```
boxes(3)
// Substitute body of `boxes`
3 match {
  case 0 => Image.empty
  case n => aBox.beside(boxes(n-1))
}
```

Knowing how to evaluate a `match` expression and using substitution again gives us

```
3 match {
  case 0 => Image.empty
  case n => aBox.beside(boxes(n-1))
}
// Substitute right-hand side expression of `case n`
aBox.beside(boxes(2))
```

We can substitute again on `boxes(2)` to obtain

```
aBox.beside(boxes(2))
// Substitute body of boxes
aBox.beside {
  2 match {
    case 0 => Image.empty
    case n => aBox.beside(boxes(n-1))
  }
}
// Substitute right-hand side expression of `case n`
aBox.beside {
  aBox.beside(boxes(1))
}
```

Repeating the process a few more times we get

```
aBox.beside {
  aBox.beside {
    1 match {
      case 0 => Image.empty
      case n => aBox.beside(boxes(n-1))
    }
  }
}
// Substitute right-hand side expression of `case n`
aBox.beside {
  aBox.beside {
      aBox.beside(boxes(0))
  }
}
// Substitute body of boxes
aBox.beside {
  aBox.beside {
    aBox.beside {
      0 match {
        case 0 => Image.empty
        case n => aBox.beside(boxes(n-1))
      }
    }
  }
}
// Substitute right-hand side expression of `case 0`
aBox.beside {
  aBox.beside {
    aBox.beside {
      Image.empty
    }
  }
}
```

Our final result, which simplifies to

```
aBox.beside(aBox).beside(aBox).beside(Image.empty)
```

is exactly what we expect. Therefore we can say that substitution works to reason about recursion. This is great! However the substitutions are quite complex and difficult to keep track of without writing them down.

7.5.1 Reasoning About Structural Recursion

There is a more practical way to reason about structural recursion. Structural recursion guarantees the overall recursion is correct if we get the individual components correct. There are two parts to the structural recursion; the base case and the recursive case. The base case we can check just by looking at it. The recursive case has the recursive call (the method calling itself) but *we don't have to consider this*. It is given to us by structural recursion so it will be correct so long as the other parts are correct. We can simply assume the recursive call it correct and then check that we are doing the right thing with the result of this call.

Let's apply this to reasoning about `boxes`.

```
def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
```

```
    case n => aBox.beside(boxes(n-1))
  }
```

We can tell the base case is correct by inspection. Looking at the recursive case we *assume* that `boxes(n-1)` will do the right thing. The question then becomes: is what we do with the result of the recursive call `boxes(n-1)`, correct? The answer is yes: if the recursion `boxes(n-1)` creates `n-1` boxes in a line, sticking a box in front of them is the right thing to do. Since the individual cases are correct the whole thing is guaranted correct by structural recursion.

This way of reasoning is much more compact than using substitution *and* guaranteed to work *if* we're using structural recursion.

Exercises

Below are some rather silly examples of structural recursion. Work out if the methods do what they claim to do *without* running them.

```
// Given a natural number, returns that number
// Examples:
//   identity(0) == 0
//   identity(3) == 3
def identity(n: Int): Int =
  n match {
    case 0 => 0
    case n => 1 + identity(n-1)
  }
```

See the solution

```
// Given a natural number, double that number
// Examples:
//   double(0) == 0
//   double(3) == 6
def double(n: Int): Int =
  n match {
    case 0 => 0
    case n => 2 * double(n-1)
  }
```

See the solution

7.6 Conclusions

In this chapter we've seen our first big pattern for structuring code, *structural recursion over the natural numbers*. There are a few key points.

First is the structural recursion pattern itself. We saw we can use this to write methods that produce a value with a varying size, and *the pattern is the same everytime.* We've seen a lot of examples that generate images, but we can use this pattern for anything that is transforming a natural number into anything else (including other natural numbers). We'll use this pattern, and other variants of structural recursion, throughout the book.

The second big idea is how we reason about structural recursion. We can use substitution but it is easier to take a shortcut. For structural recursion we are guaranteed to get the correct result if we get the base case

and the recursive case correct. In particular we don't need to reason through the recursive call; we assume it returns the correct result and only check that we correctly implement the next step adding to the result.

The third key point is that we can use the value of the counter to do other things beyond the recursion. We looked at using it to adjust the images we created at each step, which gives us new creative possibilities.

In the next section we'll look at creating more complex images using structural recursion, and see a few new techniques we can use with it.

Chapter 8

Fractals

A fractal is an image that is *self-similar*, meaning that it contains copies of itself. Fractals are an intriguing type of image as they build complex output from simple rules. In this chapter we will build some simple fractals, get more experience with structural recursion over natural numbers, and finally learn more programming techniques.

> If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.
>
> ```
> import doodle.core._
> import doodle.image._
> import doodle.image.syntax._
> import doodle.image.syntax.core._
> import doodle.java2d._
> ```

8.1 The Chessboard

In this exercise and the next we're trying to sharpen your eye for recursive structure.

Our first image is the chessboard. Though arguably not a fractal, the chessboard does contain itself: a 4x4 chessboard can be constructed from 4 2x2 chessboards, an 8x8 from 4 4x4s, and so on. The picture in fig. 8.1 shows this.

Your mission in this exercise is to identify the recursive structure in a chessboard, and implement a method to draw chessboards. The method skeleton is

```
def chessboard(count: Int): Image =
  ???
```

Implement `chessboard`. Remember we can use the structural recursion skeleton and reasoning technique to guide our implementation.

See the solution

If you have prior programming experience you might have immediately thought of creating a chessboard via two nested loops. Here we're taking a different approach by defining a larger chessboard as a composition of smaller chessboards. Grasping this different approach to decomposing problems is a key step in becoming proficient in functional programming.

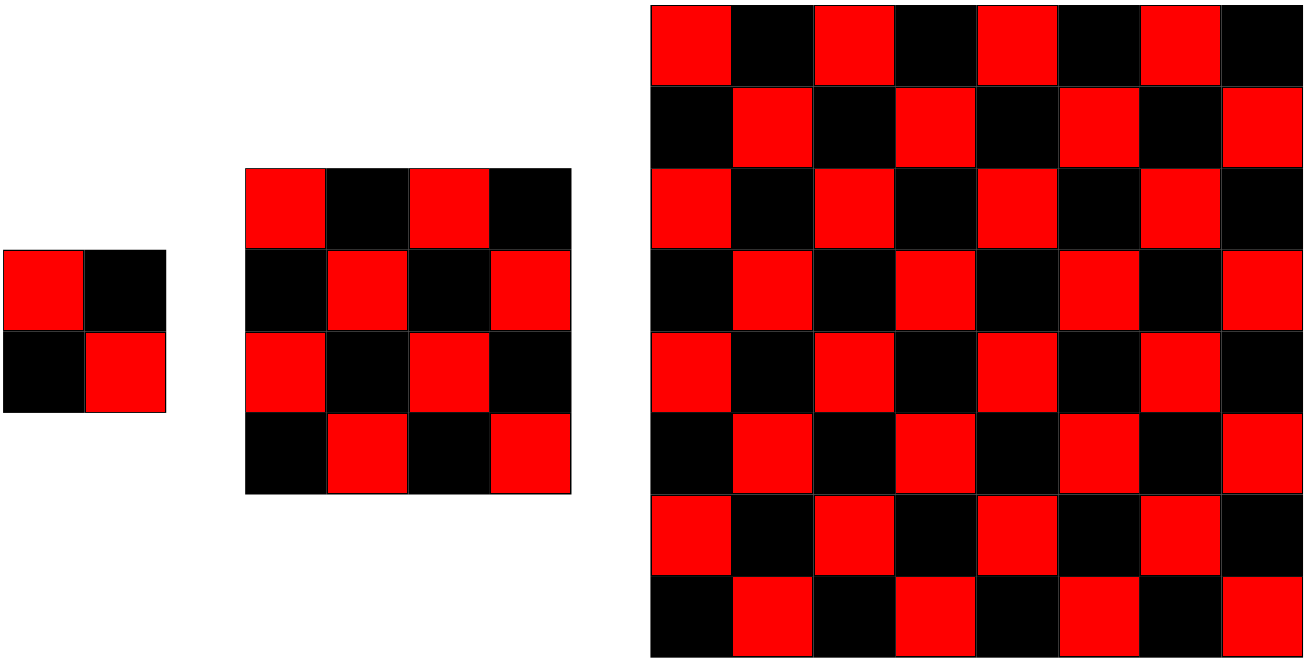

Figure 8.1: Chessboards generated by `count` from 0 to 2.

8.2 Sierpinkski Triangle

The Sierpinski triangle, show in fig. **??**, is a famous fractal. (Technically, fig. **??** shows a Sier*pink*ski triangle.)

Although it looks complicated we can break the structure down into a form that we can generate with structural recursion over the natural numbers. Implement a method with skeleton

```
def sierpinski(count: Int): Image =
  ???
```

No hints this time. We've already seen everything we need to know.

See the solution

8.3 Auxiliary Parameters

We've seen how to use structural recursion over the natural numbers to write a number of interesting programs. In this section we're going to learn how *auxillary parameters* allow us to write more complex programs. An auxiliary parameter is just an additional parameter to our method that allows us to pass extra information down the recursive call.

For example, imagine creating the picture in fig. 8.3, which shows a line of boxes that grow in size as we move along the line.

How can we create this image?

We know it has to be a structural recursion over the natural numbers, so we can immediately write down the skeleton

```
def growingBoxes(count: Int): Image =
  count match {
    case 0 => base
```

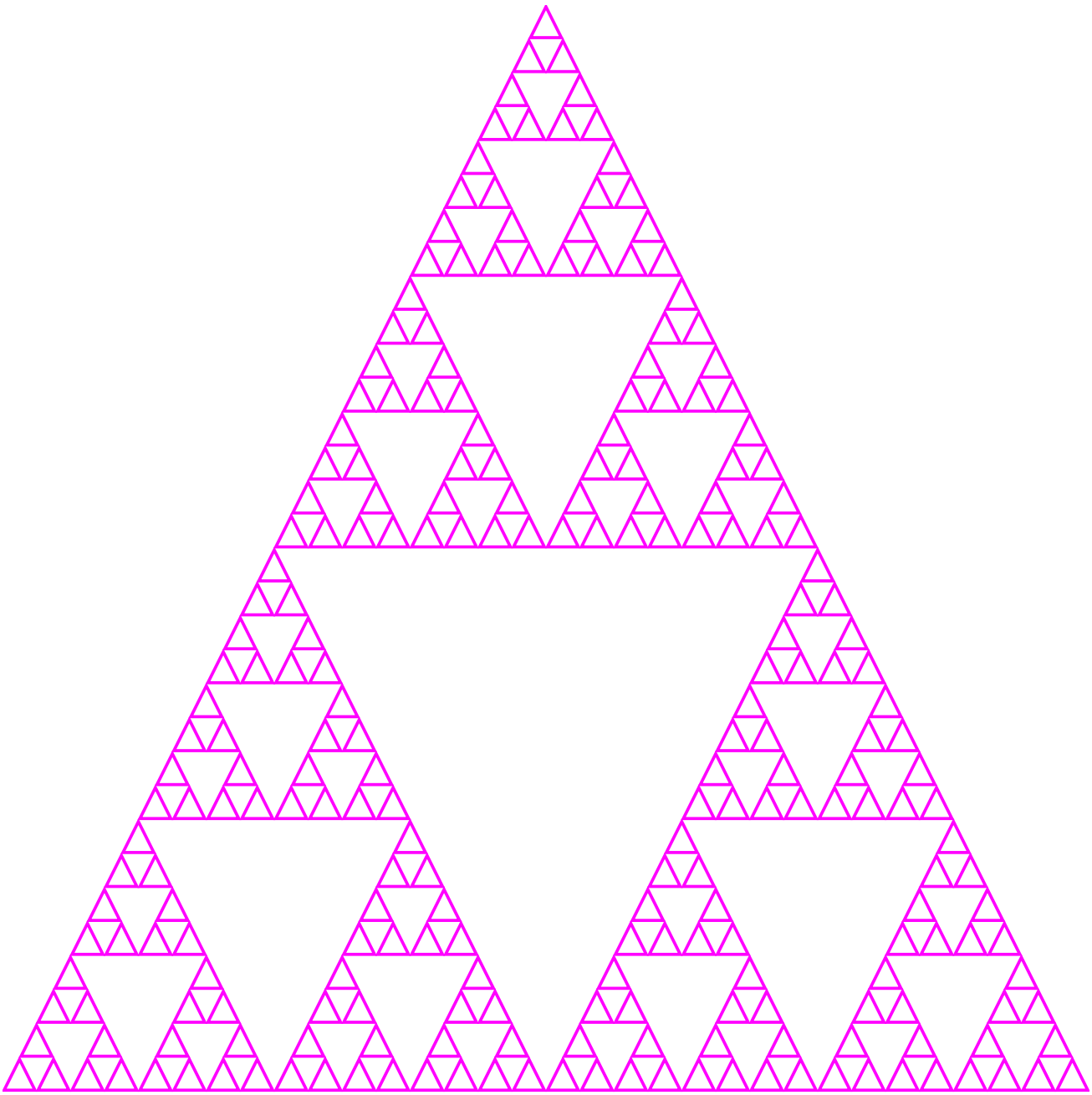

Figure 8.2: The Sierpinski triangle.

Figure 8.3: Boxes that grow in size with each recursion.

```
    case n => unit add growingBoxes(n-1)
  }
```

Using what we learned working with `boxes` earlier we can go a bit further and write down

```
def growingBoxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => Image.square(???).beside(growingBoxes(n-1))
  }
```

The challenge becomes how to make the box grow in size as we move to the right.

There are two ways to do this. The tricky way is to switch the order in the recursive case and make the size of the box a function of `n`. Here's the code.

```
def growingBoxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => growingBoxes(n-1).beside(Image.square(n*10))
  }
```

Spend some time figuring out why this works before moving on to the solution using an auxiliary parameter.

Alternatively we can simply add another parameter to `growingBoxes` that tells us how big the current box should be. When we recurse we change this size. Here's the code.

```
def growingBoxes(count: Int, size: Int): Image =
  count match {
    case 0 => Image.empty
    case n =>
      Image
        .square(size)
        .beside(growingBoxes(n-1, size + 10))
  }
```

The auxiliary parameter method has two advantages: we only have to think about what changes from one recursion to the next (in this case, the box gets larger), and it allows the caller to change this parameter (for example, making the starting box larger or smaller).

Now we've seen the auxiliary parameter method let's practice using it.

Gradient Boxes

In this exercise we're going to draw a picture like that in fig. 8.4. We already know how to draw a line of boxes. The challenge in this exercise is to make the color change at each step.

Hint: you can `spin` the fill color at each recursion.

See the solution

Concentric Circles

Now let's try a variation on the theme, drawing concentric circles as shown in fig. 8.5. Here we are changing the size rather than the color of the image at each step. Otherwise the pattern stays the same. Have a go at implementing it.

See the solution

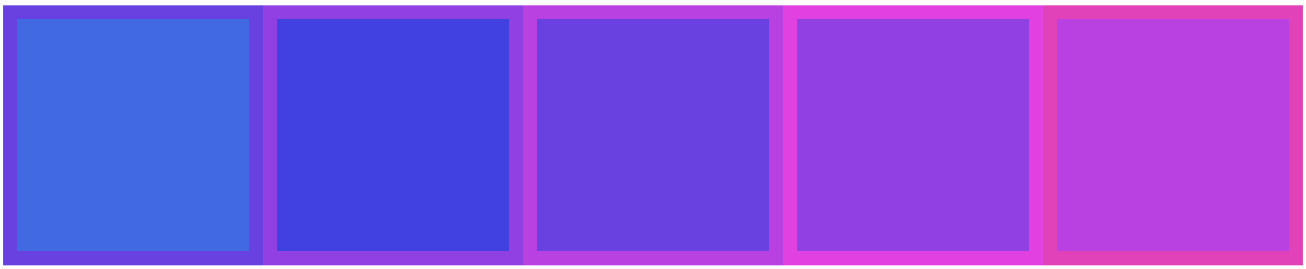

Figure 8.4: Five boxes filled with changing colors starting from Royal Blue

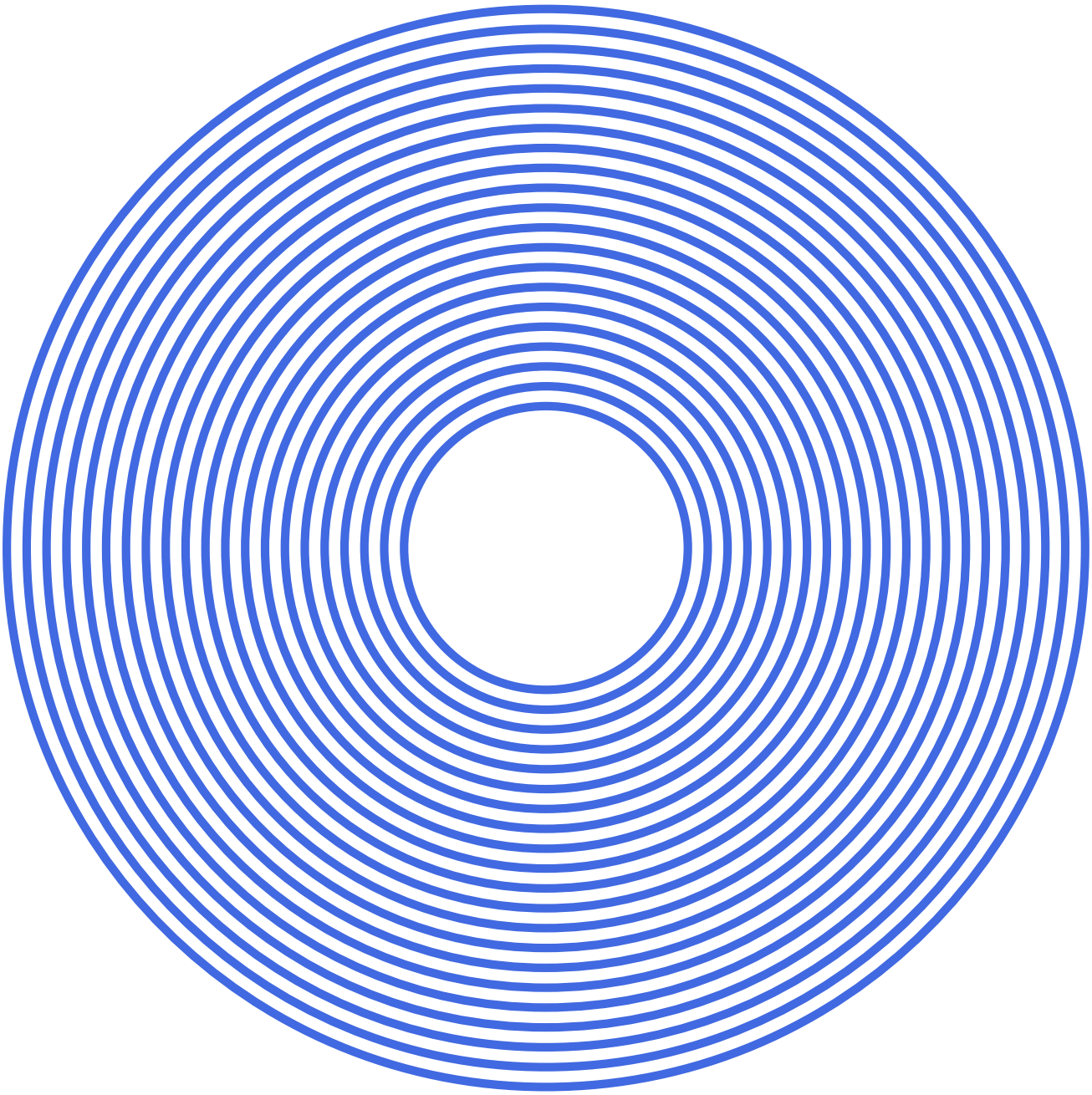

Figure 8.5: Concentric circles, colored Royal Blue

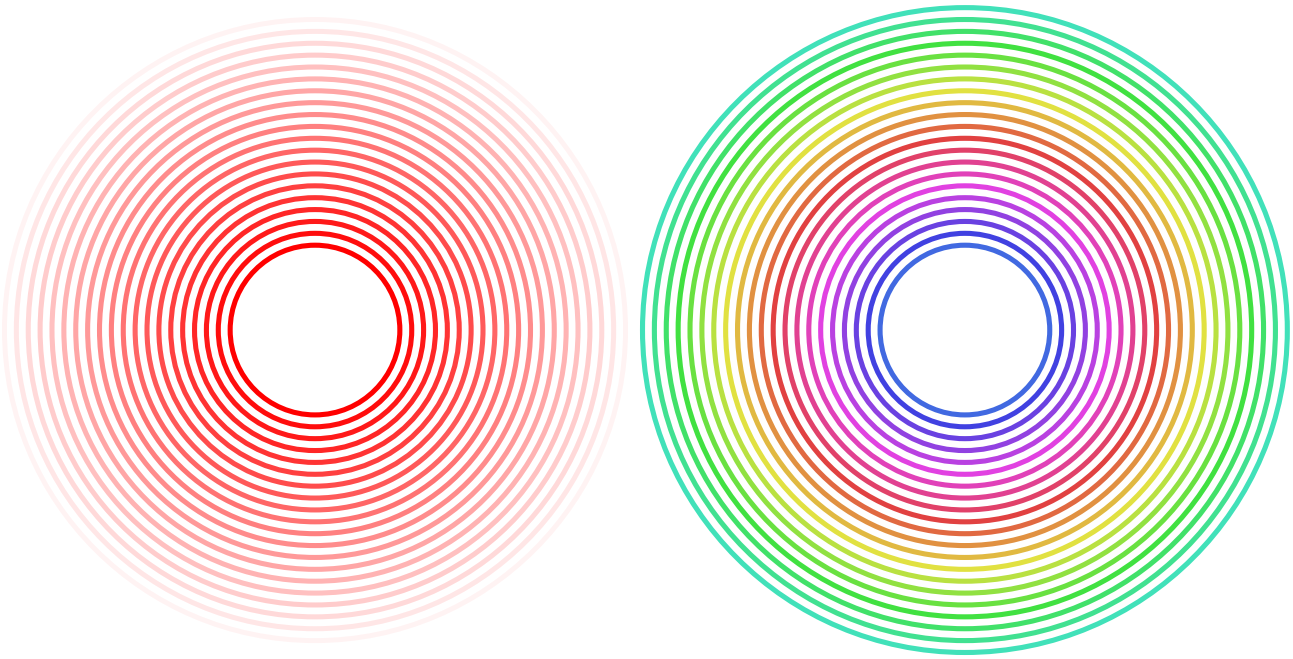

Figure 8.6: Concentric circles with interesting color variations

Once More, With Feeling

Now let's combine both techniques to change size and color on each step, giving results like those shown in fig. 8.6. Experiment until you find something you like.

See the solution

8.4 Nested Methods

A method is a declaration. The body of a method can contain declarations and expressions. Therefore, a method declaration can contain other method declarations.

To see why this is useful, lets look at a method we wrote earlier:

```
def cross(count: Int): Image = {
  val unit = Image.circle(20)
  count match {
    case 0 => unit
    case n => unit.beside(unit.above(cross(n-1)).above(unit)).beside(unit)
  }
}
```

We have declared `unit` inside the method `cross`. This means the declaration of `unit` is only in scope within the body of `cross`. It is good practice to limit the scope of declarations to the minimum needed, to avoid accidentally shadowing other declarations. However, let's consider the runtime behavior of `cross` and we'll see that is has some undesirable characteristics.

We'll use our substitution model to expand `cross(1)`.

```
cross(1)
// Expands to
{
  val unit = Image.circle(20)
```

```
  1 match {
    case 0 => unit
    case n => unit.beside(unit.above(cross(n-1)).above(unit)).beside(unit)
  }
}
// Expands to
{
  val unit = Image.circle(20)
  unit.beside(unit.above(cross(0)).above(unit)).beside(unit)
}
// Expands to
{
  val unit = Image.circle(20)
  unit.beside(unit.above
  {
    val unit = Image.circle(20)
    0 match {
      case 0 => unit
      case n => unit.beside(unit.above(cross(n-1)).above(unit)).beside(unit)
    }
  }
  .above(unit)).beside(unit)
}
// Expands to
{
  val unit = Image.circle(20)
  unit.beside(unit.above
  {
    val unit = Image.circle(20)
    unit
  }
  .above(unit)).beside(unit)
}
```

The point of this enormous expansion is to demonstrate that we're recreating `unit` every time we recurse within `cross`. We can prove this is true by printing something every time `unit` is created.

```
def cross(count: Int): Image = {
  val unit = {
    println("Creating unit")
    Image.circle(20)
  }
  count match {
    case 0 => unit
    case n => unit.beside(unit.above(cross(n-1)).above(unit)).beside(unit)
  }
}

cross(1)
// Creating unit
// Creating unit
// res1: Image = Beside(
//   Beside(Circle(20.0), Above(Above(Circle(20.0), Circle(20.0)), Circle(20.0))),
//   Circle(20.0)
// )
```

This doesn't matter greatly for `unit` because it's very small, but we could be doing something that takes up a lot of memory or time, and it's undesirable to repeat it when we don't have to.

We could solve this by shifting `unit` outside of `cross`.

```
val unit = {
  println("Creating unit")
  Image.circle(20)
}
// Creating unit
// unit: Image = Circle(20.0)

def cross(count: Int): Image = {
  count match {
    case 0 => unit
    case n => unit beside (unit above cross(n-1) above unit) beside unit
  }
}

cross(1)
// res3: Image = Beside(
//   Beside(Circle(20.0), Above(Above(Circle(20.0), Circle(20.0)), Circle(20.0))),
//   Circle(20.0)
// )
```

This is undesirable because `unit` now has a larger scope than needed. A better solution it to use a nested or internal method.

```
def cross(count: Int): Image = {
  val unit = {
    println("Creating unit")
    Image.circle(20)
  }
  def loop(count: Int): Image = {
    count match {
      case 0 => unit
      case n => unit beside (unit above loop(n-1) above unit) beside unit
    }
  }

  loop(count)
}

cross(1)
// Creating unit
// res5: Image = Beside(
//   Beside(Circle(20.0), Above(Above(Circle(20.0), Circle(20.0)), Circle(20.0))),
//   Circle(20.0)
// )
```

This has the behavior we're after, creating `unit` only once while minimising its scope. The internal method `loop` is using structural recursion exactly as before. We just need to ensure that we call it in `cross`. I usually name this sort of internal method `loop` or `iter` (short for iterate) to indicate that they're performing a loop.

This technique is just a small variation of what we've done already, but let's do a few exercises to make sure we've got the pattern.

Exercises

Chessboard

Rewrite `chessboard` using a nested method so that `blackSquare`, `redSquare`, and `base` are only created once when `chessboard` is called.

```
def chessboard(count: Int): Image = {
  val blackSquare = Image.square(30) fillColor Color.black
  val redSquare   = Image.square(30) fillColor Color.red

  val base =
    (redSquare   beside blackSquare) above (blackSquare beside redSquare)
  count match {
    case 0 => base
    case n =>
      val unit = cross(n-1)
      (unit beside unit) above (unit beside unit)
  }
}
```

See the solution

Boxing Clever

Rewrite `boxes`, shown below, so that `aBox` is only in scope within `boxes` and only created once when `boxes` is called.

```
val aBox = Image.square(20).fillColor(Color.royalBlue)

def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox.beside(boxes(n-1))
  }
```

See the solution

8.5 Exercises

We now have a number of new tools in our toolbox. It's time to get some practice putting them all together.

Here's an example of the familiar chessboard pattern. We have used an auxillary parameter to pass along a color that we change at each recursion. By changing the hue by a prime number we end up a complex pattern with infrequently repeating colors. See fig. 8.7 for an example.

```
def chessboard(count: Int, color: Color): Image =
  count match {
    case 0 =>
      val contrast = color.spin(180.degrees)
      val box = Image.square(20)
      box
        .fillColor(color)
        .beside(box.fillColor(contrast))
        .above(box.fillColor(contrast).beside(box.fillColor(color)))

    case n =>
      chessboard(n - 1, color.spin(17.degrees))
        .beside(chessboard(n - 1, color.spin(-7.degrees)))
        .above(
          chessboard(n - 1, color.spin(-19.degrees))
            .beside(chessboard(n - 1, color.spin(3.degrees)))
        )
```

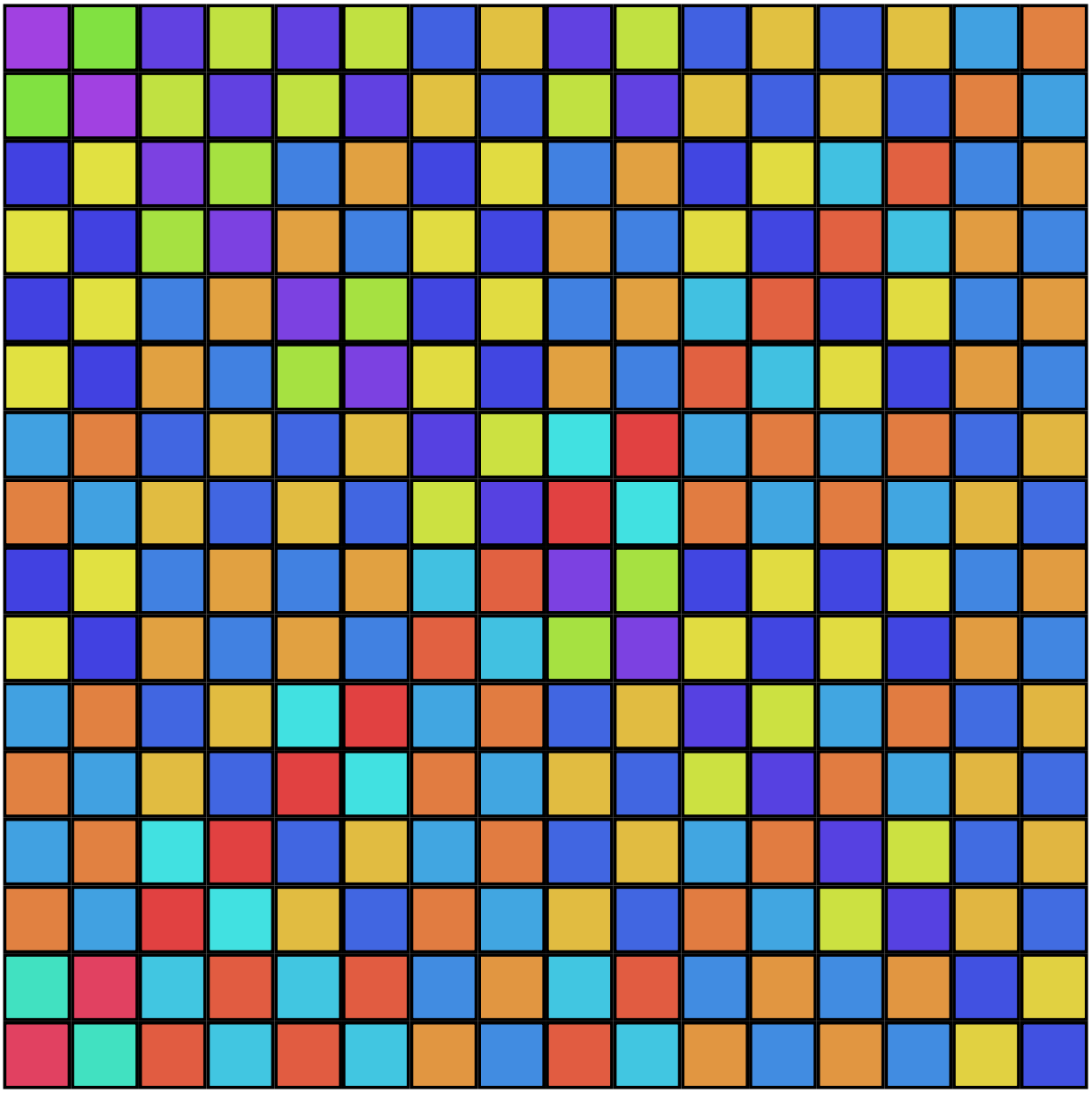

Figure 8.7: Chessboard with colors evolving at each recursive step.

```
}
```

Your mission is to take the ideas we've seen in this chapter, perhaps using the chessboard example for inspiration, and create your own artwork. No other guidelines this time; it's up to you and your imagination.

Chapter 9

Horticulture and Higher-order Functions

In this chapter we're going to learn how to draw flowers and to use functions as first-class values.

We know that programs work with values, but not all values are *first-class*. A first-class value is something we can pass as a parameter to a method, or return as a result from a method call, or give a name using `val`.

If we pass a function as an argument to another function then:

- the function that is passed is being used as a first-class value; and
- the function that is receiving the function parameter is called a *higher-order function*.

This terminology is not especially important, but you'll encounter it in other writing so it's useful to know (at least vaguely) what it means. It will soon become clearer when we see some examples.

So far we have used the terms *function* and *method* interchangeably. We'll soon see that in Scala these two terms have distinct, though related, meanings.

Enough background. Let's dive in to see:

- how we create functions in Scala; and
- how we use first-class functions to structure programs.

Our motivating example for this will be drawing flowers as in fig. 9.1.

> If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.
>
> ```
> import doodle.core._
> import doodle.image._
> import doodle.image.syntax._
> import doodle.image.syntax.core._
> import doodle.java2d._
> ```

9.1 Functions

A function is basically a method, but we can use a function as a first-class value:

- we can pass it as an argument or parameter to a method or function;

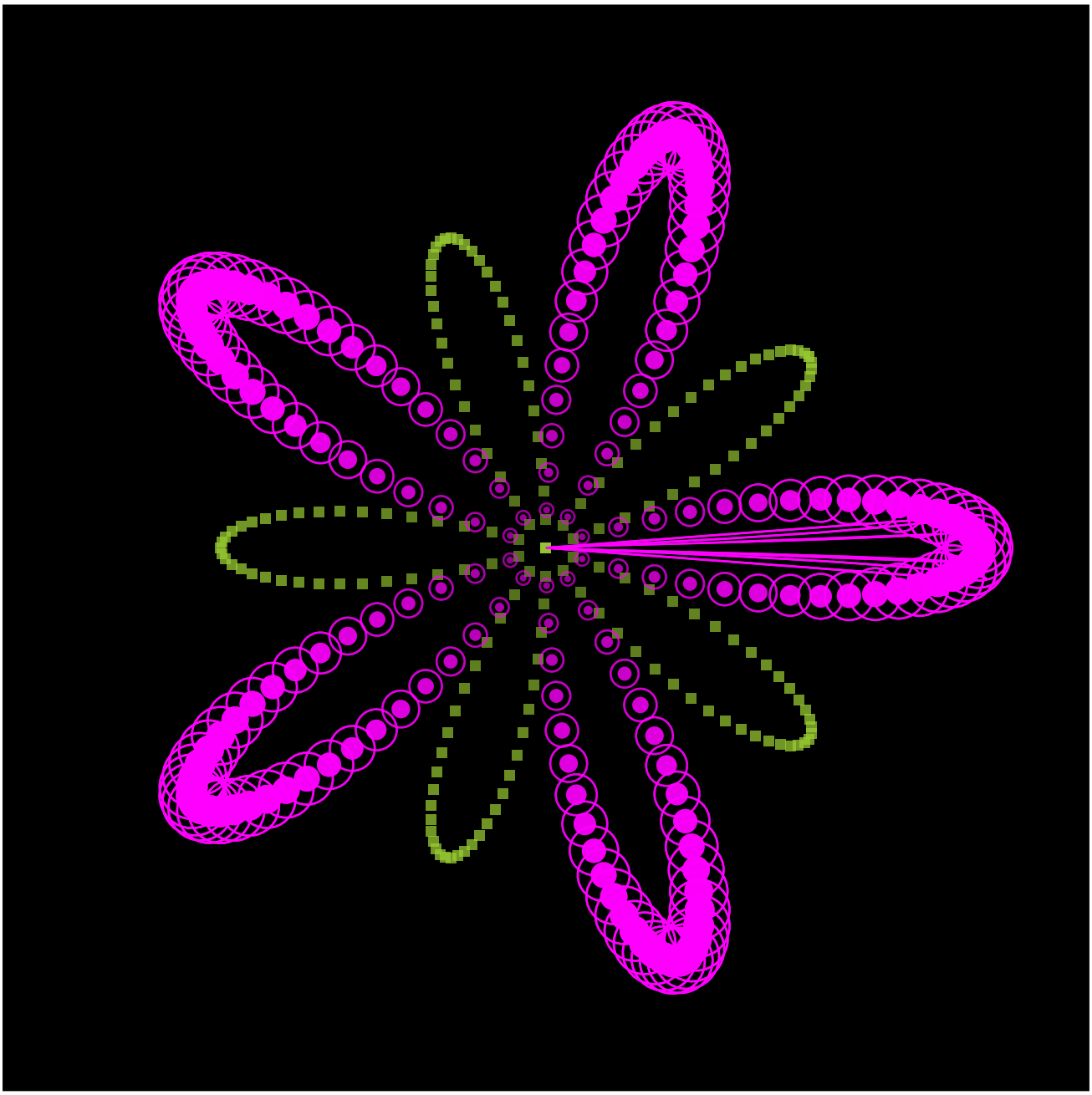

Figure 9.1: A flower created using the techniques in this chapter

- we can return it from a method or function; and
- we can give it a name using `val`.

Here's an example where we give the name `add42` to a function that adds 42 to its input.

```
val add42 = (x: Int) => x + 42
// add42: Int => Int = <function1>
```

We can call it just like we'd call a method.

```
add42(0)
// res0: Int = 42
```

This is an example of a function literal. Let's learn about them now.

9.1.1 Function Literals

We've just seen an example of a function literal, which was

```
(x: Int) => x + 42
// res1: Int => Int = <function1>
```

The general syntax is an extension of this.

> **Function Literal Syntax**
>
> The syntax for declaring a function literal is
>
> ```
> (parameter: type, ...) => expression
> ```
>
> where - the optional `parameters` are the names given to the function parameters; - the `types` are the types of the function parameters; and - the `expression` determines the result of the function.
>
> The parentheses around the parameters are optional if the function has just a single parameter.

9.1.2 Function Types

To pass functions to methods we need to know how to write down their types (because when we declare a parameter we have to declare its type).

We write a function type like `(A, B) => C` where `A` and `B` are the types of the parameters and `C` is the result type. The same pattern generalises from functions of no arguments to an arbitrary number of arguments.

Here's an example. We create a method that accepts a function, and that function is from `Int` to `Int`. We write this type as `Int => Int` or `(Int) => Int`.

```
def squareF(x: Int, f: Int => Int): Int =
  f(x) * f(x)
```

We can pass `add42` to this method

```
squareF(0, add42)
// res2: Int = 1764
```

We could also pass a function literal

```
squareF(0, x => x + 42)
// res3: Int = 1764
```

Note that we didn't have to put the parameter type on the function literal in this case because Scala has enough information to infer the type.

Function Type Declaration Syntax

To declare a function type, write

```
(A, B, ...) => C
```

where

- `A, B, ...` are the types of the input parameters; and
- `C` is the type of the result.

If a function only has one parameter the parentheses may be dropped:

```
A => B
```

9.1.3 Functions as Objects

All first class values are objects in Scala, including functions. This means functions can have methods, including some useful means for composition.

```
val addTen = (a: Int) => a + 10
// addTen: Int => Int = <function1>
val double = (a: Int) => a * 2
// double: Int => Int = <function1>
val combined = addTen.andThen(double) // this composes the two functions
// combined: Int => Int = scala.Function1$$Lambda$9669/1403896095@119ecd3a // this composes the two
    functions
combined(5)
// res4: Int = 30
```

Calling a function is actually calling the method called `apply` on the function. Scala allows a shortcut for any object that has a method called `apply`, where can drop the method name `apply` and write the call like a function call. This means the following are equivalent.

```
val halve = (a: Int) => a / 2
// halve: Int => Int = <function1>
halve(4)
// res5: Int = 2
halve.apply(4)
```

```
// res6: Int = 2
```

9.1.4 Converting Methods to Functions

Methods are very similar to functions, so Scala provides a way to convert functions to methods. If we follow a method name with a _ it will be converted to a function.

```
def times42(x: Int): Int =
  x * 42

val times42Function = times42 _
// times42Function: Int => Int = <function1>
```

We can also write a method call but replace all parameters with _ and Scala will convert the method to a function.

```
val times42Function2 = times42(_)
// times42Function2: Int => Int = <function1>
```

Exercises

9.1.4.0.1 Function Literals Let's get some practice writing function literals. Write a function literal that:

- squares it's `Int` input;
- has a `Color` parameter and spins the hue of that `Color` by 15 degrees; and
- takes an `Image` input and creates four copies in a row, where each copy is rotated by 90 degrees relative to the previous image (use the `rotate` method on `Image` to achieve this.)

See the solution

9.1.4.0.2 Function Types Here's an interesting function we'll do more with in later sections. We don't need to understand what it does right now, though you might want to experiment with it.

```
val roseFn = (angle: Angle) =>
  Point.cartesian((angle * 7).cos * angle.cos, (angle * 7).cos * angle.sin)
```

What is the type of the function `roseFn` defined above? What does this type mean?

See the solution

9.2 Parametric Curves

Right now we only know how to create basic shapes like circles and rectangles. We'll need more control to create the flower shapes that are our goal. We're going to use a tool from mathematics known as a *parametric equation* or *parametric curve* to do so.

A parametric equation is a function from some input (the parameter in "parametric") to a point, a location in space. The input tells us how far along the curve we are. For example, a parametric equation for a circle might have as its input an angle and it would give us the point on the circle at that angle. In Scala we could write

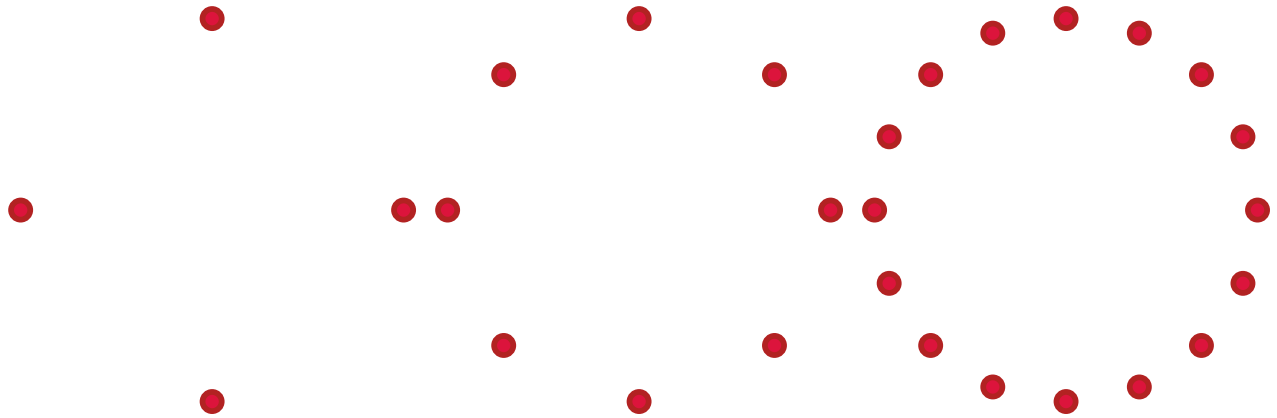

Figure 9.2: Parametric circle with points drawn, from left to right, every 90, 45, and 22.5 degrees.

```
def parametricCircle(angle: Angle): Point =
  ???
```

If we choose lots of different values for the input, and then draw a shape at each point we get back from the parametric equation, we can suggest the shape of the curve.

In fig. 9.2 we give an example of drawing small circles at the points generated by the parametric equation for a circle. Going from left to right we draw points every 90, 45, and 22.5 degrees. You can see how the outline of the shape, the large circle, becomes clearer as we draw more points.

To create parametric curves we need to learn 1) how to represent points in Doodle, 2) how to position an image at a particular point in space, and 3) revise a bit of geometry you might not have touched since high school. Let's look at each item in turn.

9.3 Points

In Doodle we have a `Point` type to represent a position in two dimensions. We have two equivalent representations in terms of:

- x and y coordinates, called a cartesian representation; and
- in terms of an angle and distance (the radius) at that angle from the origin, called a polar representation.

This difference is shown in fig. 9.3.

We can create points in the cartesian representation using `Point(Double, Double)` where the two parameters are the x and y coordinates, and in the polar representation using `Point(Double, Angle)` where we specify the radius and the angle. The table below shows the main methods on `Point`.

Constructor	Type	Description	Example
`Point(Double, Double)`	`Point`	Constructs a `Point` using the cartesian representation.	`Point(1.0, 1.0)`
`Point(Double, Angle)‘‘`	`Point`	Constructs a `Point` using the polar representation.	`Point(1.0, 90.degrees)`
`Point.zero`	`Point`	Constructs a `Point` at the origin (x and y are zero)	`Point.zero`

Constructor	Type	Description	Example
`Point.x`	`Double`	Gets the x coordinate of the `Point`.	`Point.zero.x`
`Point.y`	`Double`	Gets the y coordinate of the `Point`.	`Point.zero.y`
`Point.r`	`Double`	Gets the radius of the `Point`.	`Point.zero.r`
`Point.angle`	`Angle`	Gets the angle of the `Point`.	`Point.zero.angle`

9.4 Flexible Layout

Can we position an `Image` at a point? So far we only know how to layout images with `on`, `beside`, and `above`. We need an additional tool, the `at` method, to achieve more flexible layout. Here's an example using `at` that draws a dot at the corners of a square.

```
val dot = Image.circle(5).strokeWidth(3).strokeColor(Color.crimson)
val squareDots =
  dot.at(0, 0)
    .on(dot.at(0, 100))
    .on(dot.at(100, 100))
    .on(dot.at(100, 0))
```

This produces the image shown in fig. 9.4.

To understand how `at` layout works, and why we have to place the dots `on` each other, we need to know a bit more about how Doodle does layout.

Every `Image` in Doodle has a point called its *origin*, and a *bounding box* which determines the limits of the image. By convention the origin is in the center of the bounding box but this is not required. We can see the origin and bounding box of an `Image` by calling the `debug` method. In fig. 9.5 we show the output of the code

```
val c = Image.circle(40)
val c1 = c.beside(c.at(10, 10)).beside(c.at(10, -10)).debug
val c2 = c.debug.beside(c.at(10, 10).debug).beside(c.at(10, -10).debug)
val c3 = c.debug.beside(c.debug.at(10, 10)).beside(c.debug.at(10, -10))
c1.above(c2).above(c3)
```

This shows how the origin and bounding box change as we combines `Images`.

When we layout `Images` using `above`, `beside`, or `on` it is the bounding boxes and origins that determine how the individual components are positioned relative to one another. For `on` the rule is that the origins are placed on top of one another. For `beside` the rule is that origins are horizontally aligned and placed so that the bounding boxes just touch. The origin of the compound image is placed equidistant from the left and right edges of the compound bounding box on the horizontal line that connects the origins of the component images. The rule for `above` is the same as `beside`, but we use vertical alignment instead of horizontal alignment.

Using `at` we can move an `Image` relative to its origin. In the examples we're using here we want all the elements to share the same origin, so we use `on` to combine `Images` that we have moved using `at`.

There are four ways we can call `at`:

- by passing the x- and y-offset, as in `dot.at(100, 100)`;
- by passing the radius and angle, as in `dot.at(100, 90.degrees)`;

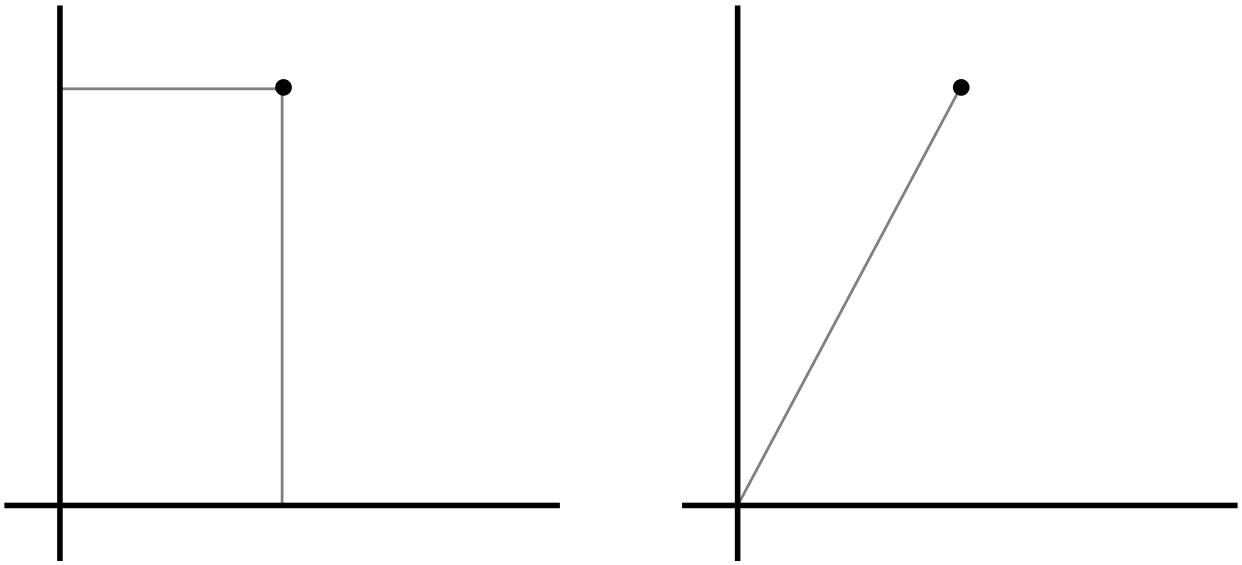

Figure 9.3: A point represented in cartesian (x and y) coordinates and polar (radius and angle) coordinates

Figure 9.4: Using `at` layout to position four dots at the corners of a square.

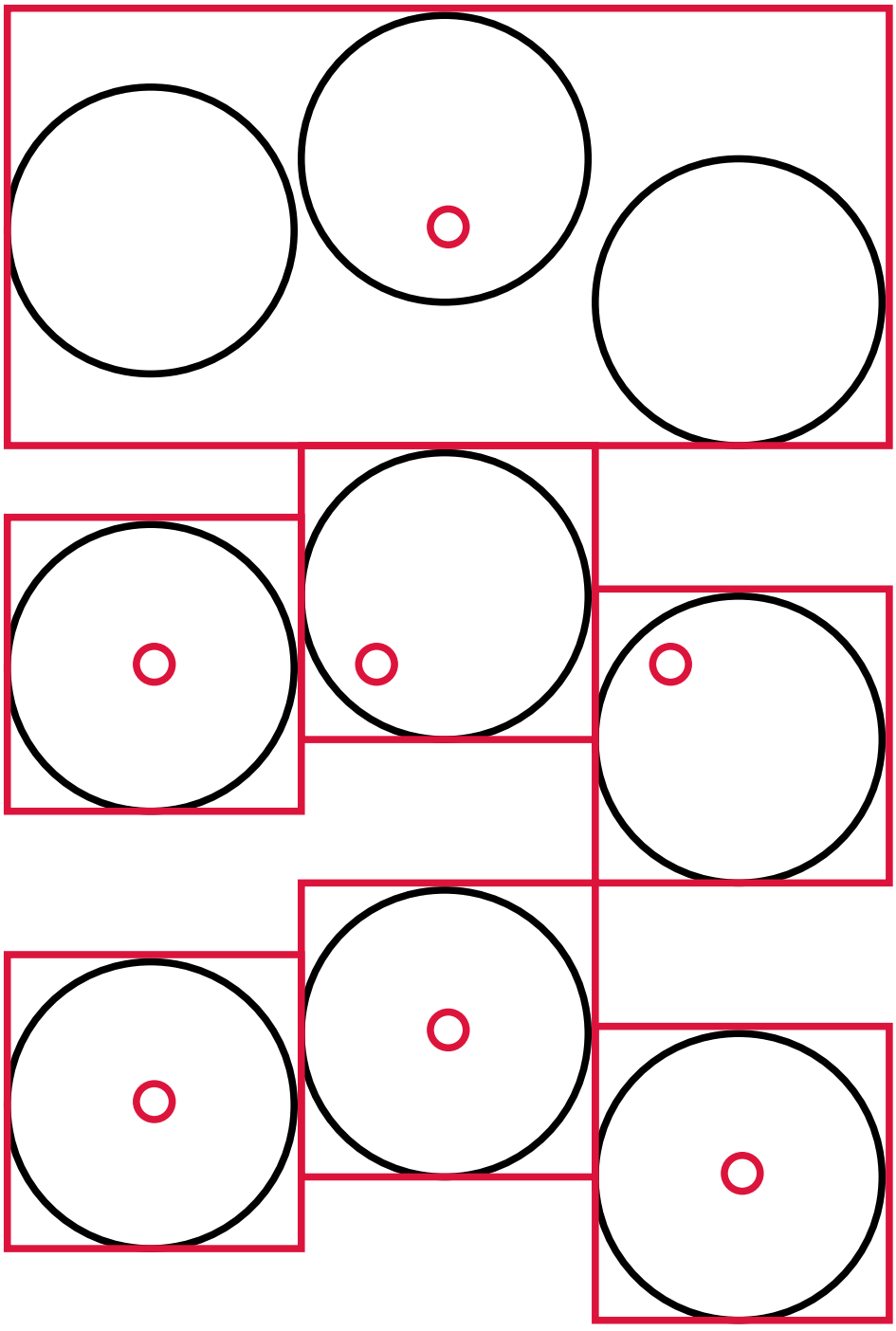

Figure 9.5: Using the debug method to inspect the origin and bounding box of an `Image`

- by passing a `Point`, as in `dot.at(Point(100, 100))`; or
- by passing a `Vec` (a vector) giving the offset, as in `dot.at(Vec(100, 100))`.

We can convert a `Point` to a `Vec` using the `toVec` method.

```
Point.cartesian(1.0, 1.0).toVec
// res1: Vec = Vec(1.0, 1.0)
```

9.5 Geometry

The final building block is the geometry to position points. If a point is positioned at a distance `r` from the origin at an angle `a`, the x- and y-coordinates are `(a.cos) * r` and `(a.sin) * r` respectively. Alternatively we can just use polar form! For example, here's how we would position a point at a distance of 1 and an angle of 45 degrees.

```
val polar = Point(1.0, 45.degrees)
// polar: Point = Polar(1.0, Angle(0.7853981633974483))
val cartesian = Point((45.degrees.cos) * 1.0, (45.degrees.sin) * 1.0)
// cartesian: Point = Cartesian(0.7071067811865476, 0.7071067811865475)

// They are the same
polar.toCartesian == cartesian
// res2: Boolean = true
cartesian.toPolar == polar
// res3: Boolean = true
```

9.6 Putting It All Together

We can put this all together to create a parametric circle. In cartesian coordinates the code for a parametric circle with radius 200 is

```
def parametricCircle(angle: Angle): Point =
  Point.cartesian(angle.cos * 200, angle.sin * 200)
```

In polar form it is simply

```
def parametricCircle(angle: Angle): Point =
  Point.polar(200, angle)
```

Now we could sample a number of points evenly spaced around the circle. To create an image we can draw something at each point (say, a triangle).

```
def sample(samples: Int): Image = {
  // Angle.one is one complete turn. I.e. 360 degrees
  val step = Angle.one / samples
  val dot = Image
              .triangle(10, 10)
              .fillColor(Color.limeGreen)
              .strokeColor(Color.lawngreen)
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => Image.empty
      case n =>
```

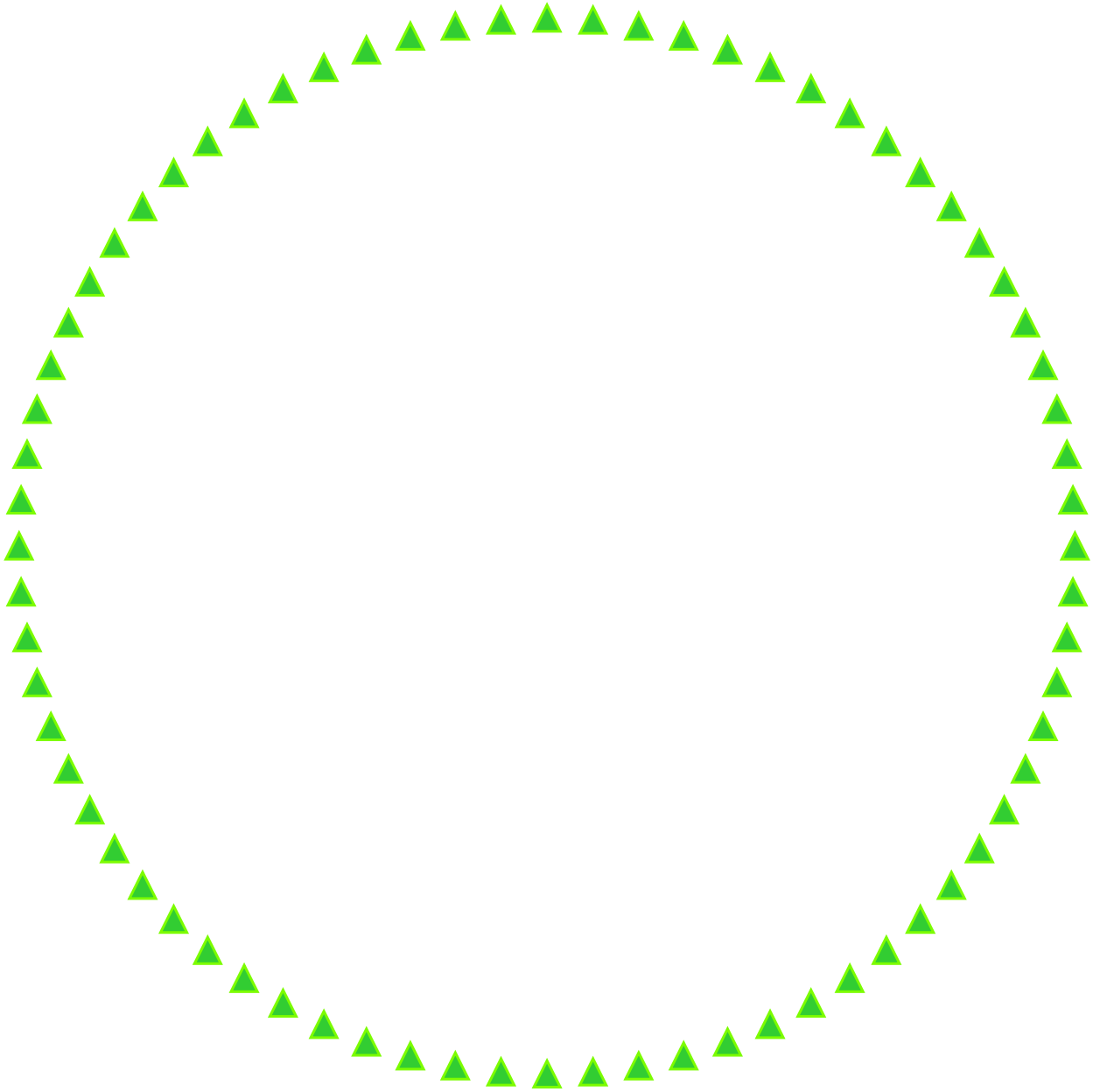

Figure 9.6: Triangles arranged in a circle, using the code from `sample` above.

```
        dot.at(parametricCircle(angle)).on(loop(n - 1))
    }
  }

  loop(samples)
}
```

This is a structural recursion, which is hopefully a familiar pattern by now.

If we draw this we'll see the outline of a circle suggested by the triangles. See fig. 9.6, which shows the result of `sample(72)`.

9.6.1 Parametric Curves as First-class Functions

So far we haven't seen anything that requires we use our parametric curves as functions instead of methods (and, indeed, we have defined them as method though we know we can easily convert methods to functions.) It's time we got something useful from functions. Remember that functions are first-class values, which means: we can pass them to a method, we can return them from a method, and we can give them a name using `val`. We're going to see an example where the first property—the ability to pass them as parameters—is useful.

We've just defined a method called `sample` that samples from our parametric curve. Right now it is restricted to sampling from the method `parametricCircle`. It would make a lot of sense to reuse this method with different parametric curves, which means we need to be able to pass a parametric curve to sample from to the `sample` method. We can do this with a function parameter. Here is what the code might look like.

```
def sample(samples: Int, dot: Image, curve: Angle => Point): Image = {
  val step = Angle.one / samples
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => Image.empty
      case n =>
        dot.at(curve(angle)).on(loop(n - 1))
    }
  }

  loop(samples)
}
```

In this implementation of sample I have added *two* new parameters, the parametric curve to sample from and the `Image` to use to draw the samples. This gives us more flexibility in the output. Now we just need to define some more parametric curves, which is what the next exercise involves.

Exercises

We have some new tools in our toolbox. It's time to have some fun exploring what we can do with them.

9.6.1.0.1 Spirals To create a circle we keep the radius constant as the angle increases. If, instead, the radius increases as the angle increases we'll get a spiral. (How quickly should the radius increase? It's up to you! Different choices will give you different spirals.)

Implement a function or method `parametricSpiral` that creates a spiral.

See the solution

9.6.1.0.2 Samples Use the parametric curves we have defined so far to create something interesting. There is an example in fig. 9.7

9.7 Flowers and Other Curves

In the previous section we saw that is was useful for methods to accept functions as parameters. In this section we'll see that it is useful for methods to return functions.

We've seen all the basic steps we need to make our flowers. Now we just need to know the curve that makes the flower shape! The shape I used is known as the rose curve. One example is shown in fig. 9.8.

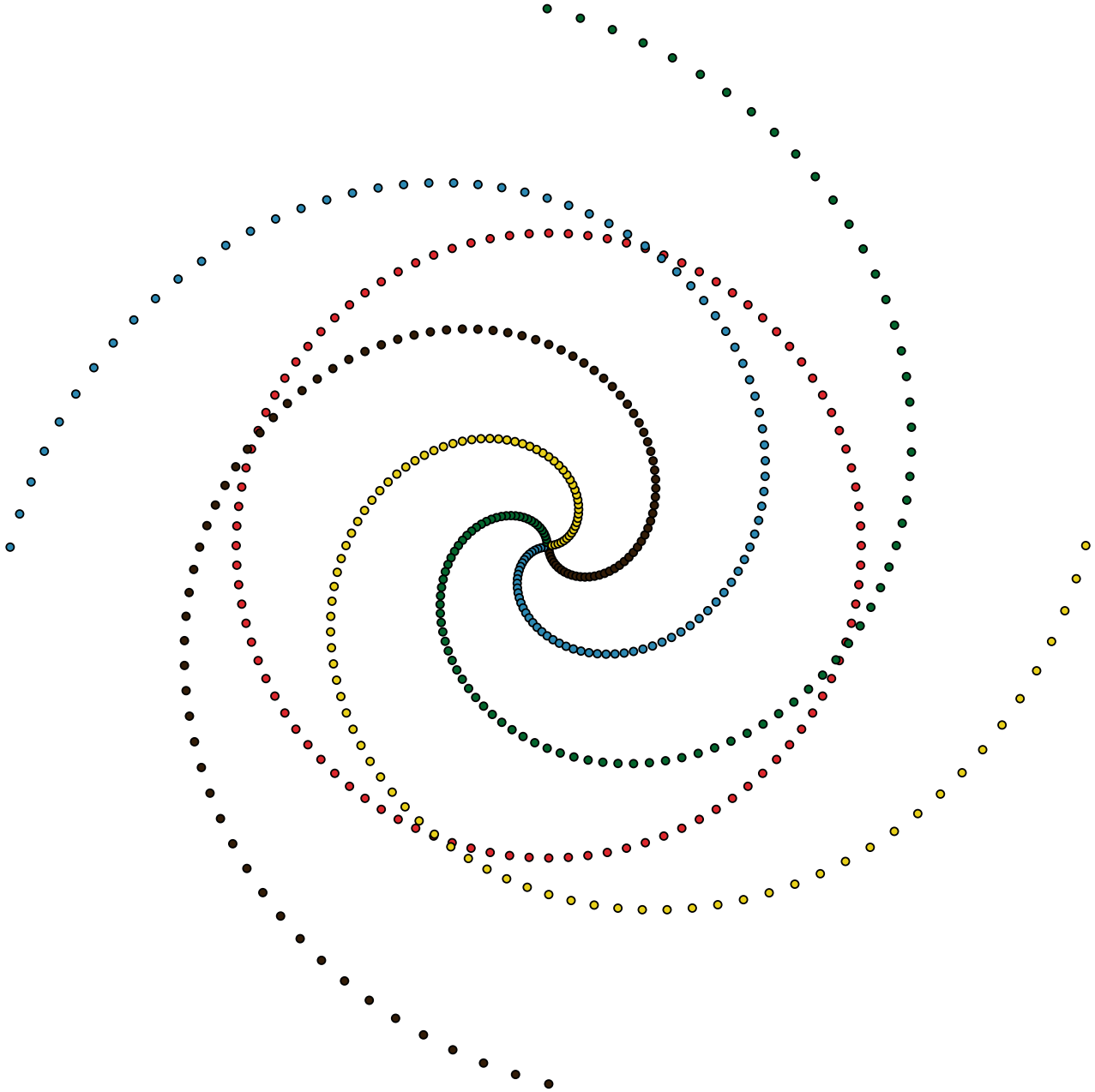

Figure 9.7: A picture created using the parametric curves we have seen so far.

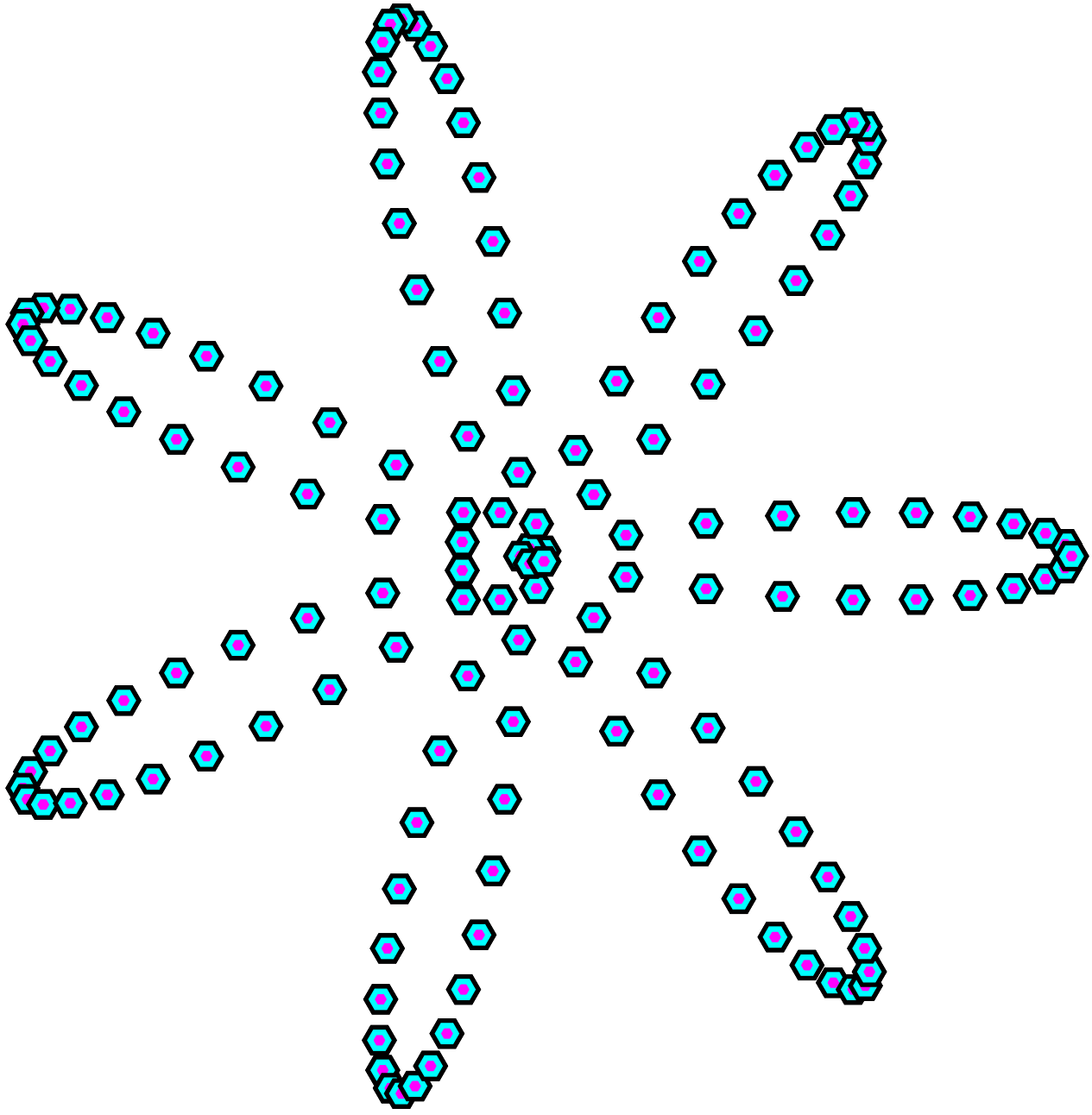

Figure 9.8: An example of the rose curve.

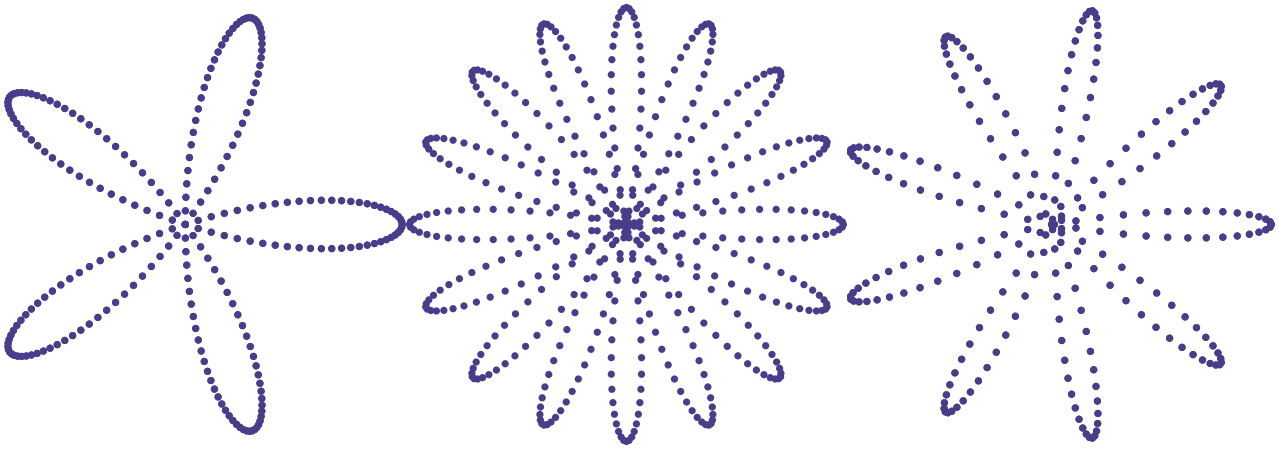

Figure 9.9: Examples of rose curves, with the parameter k chosen as 5, 8, or 9.

The code for the parametric curve that gives this shape is below.

```
// Parametric equation for rose with k = 7
val rose7 = (angle: Angle) =>
  Point((angle * 7).cos * 200, angle)
```

You may wonder why I called this function `rose7`. It's because we can vary the shape by changing the value 7 to something else. We could make a method or function to which we pass the value of this parameter and this function would return a particular rose curve. Here's that idea in code.

```
def rose(k: Int): Angle => Point =
  (angle: Angle) => Point((angle * k).cos * 200, angle)
```

The `rose` method describes a family of curves. They all look similar, and we create individuals by choosing a particular value for the parameter k. In fig. 9.9 we show more rose curves, this time with k as 5, 8, and 9 respectively.

Let's look at some other interesting curves. In fig. 9.10 we show examples of a family of curves called Lissajous curves.

The code for this is

```
def lissajous(a: Int, b: Int, offset: Angle): Angle => Point =
  (angle: Angle) =>
    Point(100 * ((angle * a) + offset).sin, 100 * (angle * b).sin)
```

The examples in fig. 9.10 use values of a and b of 1, 2, or 3, and the `offset` set to 90 degrees.

There are an unlimited number of functions we could use to create interesting curves. Let's see one more example, this time of what known as an epicycloid. An epicycloid is produced when we trace a point on a circle rotating around another circle. We can stack circles on top of circles, and change the speed at which they rotate, to produce many different curves. For our example we are going to fix the number and radius of the circles and allow their speed of rotation to vary. Here is the code:

```
def epicycloid(a: Int, b: Int, c: Int): Angle => Point =
  (angle: Angle) =>
    (Point(75, angle * a).toVec + Point(32, angle * b).toVec + Point(15, angle * c).toVec).toPoint
```

You might notice this code converts points to vectors and back again. This is a little technical detail (we cannot add points but we can add vectors) that isn't important if you aren't familiar with vectors.

In fig. 9.11 we see three examples created by choosing the parameters a, b, c, as (1, 6, 14), (7, 13, 25), and (1, 7, -21) respectively.

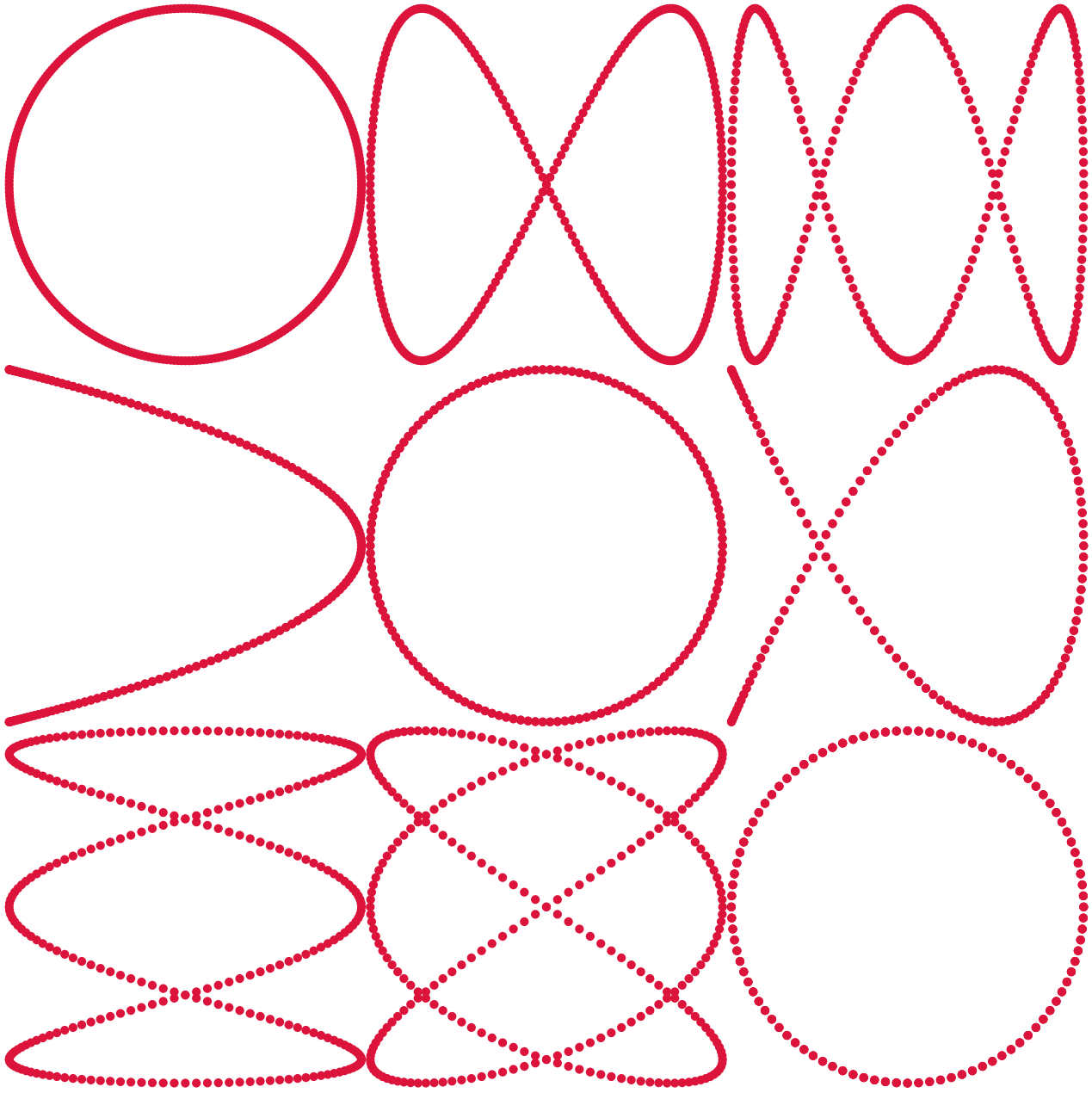

Figure 9.10: Examples of Lissajous curves, with the parameters a and b chosen as 1, 2, or 3.

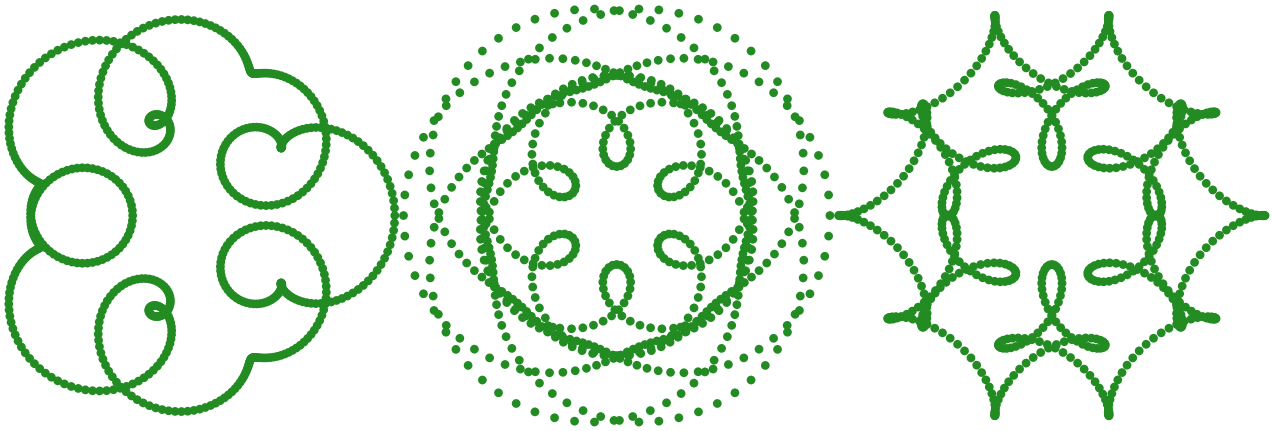

Figure 9.11: Examples of epicycloid curves, with the parameters chosen as (1, 6, 14), (7, 13, 25), and (1, 7 -21).

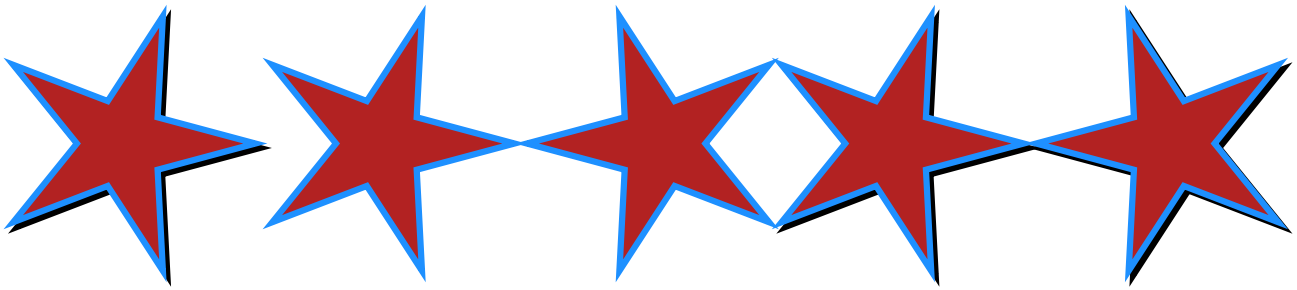

Figure 9.12: Illustrating function composition by showing the output of the individual components and the composition.

Exercise

We now have a lot of tools to play with. Your task here is simply to use some of the ideas we've just seen to make an image that you like. For inspiration you could use the image that we started the chapter with, but don't let it constrain you if you think of something else to explore.

9.8 Higher Order Methods and Functions

In previous sections we have seen the utility of passing functions to methods and returning functions from methods. In this section we'll see the usefulness of *function composition**. Composition, in the mathematical rather than artistic sense, means creating something more complex by combining simpler parts. We could say we compose the numbers 1 and 1, using addition, to produce 2. By composing functions we mean to create a function that connects the output of one component function to the input of another component function.

Here's an example. We use the `andThen` method to create a function that connects the output of the first function to the input of the second function.

```
val dropShadow = (image: Image) =>
  image.on(image.strokeColor(Color.black).fillColor(Color.black).at(5, -5))

val mirrored = (image: Image) =>
  image.beside(image.transform(Transform.horizontalReflection))

val composed = mirrored.andThen(dropShadow)
```

In fig. 9.12 we see the output of the program

```
val star = Image
  .star(100, 50, 5, 0.degrees)
  .fillColor(Color.fireBrick)
  .strokeColor(Color.dodgerBlue)
  .strokeWidth(7.0)
dropShadow(star)
  .beside(mirrored(star))
  .beside(composed(star))
```

This shows how the composed function applies the output of the first function to the second function: we first mirror the function then add a drop shadow.

Let's see how we can apply function composition to our examples of parametric curves. One limitation of the parametric cures we've created so far is that their size is fixed. For example when we defined `parametricCircle` we fixed the radius at 200.

```
def parametricCircle(angle: Angle): Point =
  Point.polar(200, angle)
```

What if we want to create circles of different radius? We could use a method that returns a function like so.

```
def parametricCircle(radius: Double): Angle => Point =
  (angle: Angle) => Point.polar(radius, angle)
```

This would be a reasonable solution but we're going to explore a different approach using our new tool of function composition. Our approach will be this:

- each parametric curve will be of some default size that we'll loosely define as "usually between 0 and 1"; and
- we'll define a function `scale` that will change the size as appropriate.

A quick example will make this more concrete. Let's redefine `parametricCircle` so the radius is 1.

```
val parametricCircle: Angle => Point =
  (angle: Angle) => Point(1.0, angle)
```

Now we can define `scale`.

```
def scale(factor: Double): Point => Point =
  (point: Point) => Point(point.r * factor, point.angle)
```

We can use function composition to create circles of different sizes as follows.

```
val circle100 = parametricCircle.andThen(scale(100))
val circle200 = parametricCircle.andThen(scale(200))
val circle300 = parametricCircle.andThen(scale(300))
```

We can use the same approach for our spiral, adjusting the function slightly so the radius of the spiral varies from about 0.36 at 0 degrees to 1 at 360 degrees.

```
val parametricSpiral: Angle => Point =
  (angle: Angle) => Point(Math.exp(angle.toTurns - 1), angle)
```

Then we can compose with `scale` to produce spirals of different size.

```
val spiral100 = parametricSpiral.andThen(scale(100))
val spiral200 = parametricSpiral.andThen(scale(200))
val spiral300 = parametricSpiral.andThen(scale(300))
```

What else can we do with function composition?

Our parametric functions have type `Angle => Point`. We can compose these with functions of type `Point => Image` and with this setup we can make the "dots" from which we build our images depend on the point.

Here's an example when the dots get bigger as the angle increases.

```
val growingDot: Point => Image =
  (pt: Point) => Image.circle(pt.angle.toTurns * 20).at(pt)

val growingCircle = parametricCircle
  .andThen(scale(100))
  .andThen(growingDot)
```

Exercise: Sample

If we want to draw this function we'll need to change `sample` so the parametric has type `Angle => Image` instead of `Angle => Point`. In other words we want the following skeleton.

```
def sample(samples: Int, curve: Angle => Image): Image =
  ???
```

Implement `sample`.

See the solution

Once we've implemented `sample` we can start drawing pictures. For example, in fig. 9.13 we have the output of `growingCircle` above.

9.8.1 More Uses of Composition

At this point we can do a lot. Let's see another example. Remember the concentric circles exercise we used as an example:

```
def concentricCircles(count: Int, size: Int): Image =
  count match {
    case 0 => Image.empty
    case n => Image.circle(size).on(concentricCircles(n-1, size + 5))
  }
```

This pattern allows us to create many different images by changing the use of `Image.circle` to another shape. However, each time we provide a new replacement for `Image.circle`, we also need a new definition of `concentricCircles` to go with it.

We can make `concentricCircles` completely general by supplying the replacement for `Image.circle` as a parameter. Here we've renamed the method to `concentricShapes`, as we're no longer restricted to drawing circles, and made `singleShape` responsible for drawing an appropriately sized shape.

```
def concentricShapes(count: Int, singleShape: Int => Image): Image =
  count match {
    case 0 => Image.empty
    case n => singleShape(n).on(concentricShapes(n-1, singleShape))
  }
```

Now we can re-use the same definition of `concentricShapes` to produce plain circles, squares of different hue, circles with different opacity, and so on. All we have to do is pass in a suitable definition of `singleShape`:

```
// Passing a function literal directly:
val blackCircles: Image =
  concentricShapes(10, (n: Int) => Image.circle(50 + 5*n))

// Converting a method to a function:
def redCircle(n: Int): Image =
```

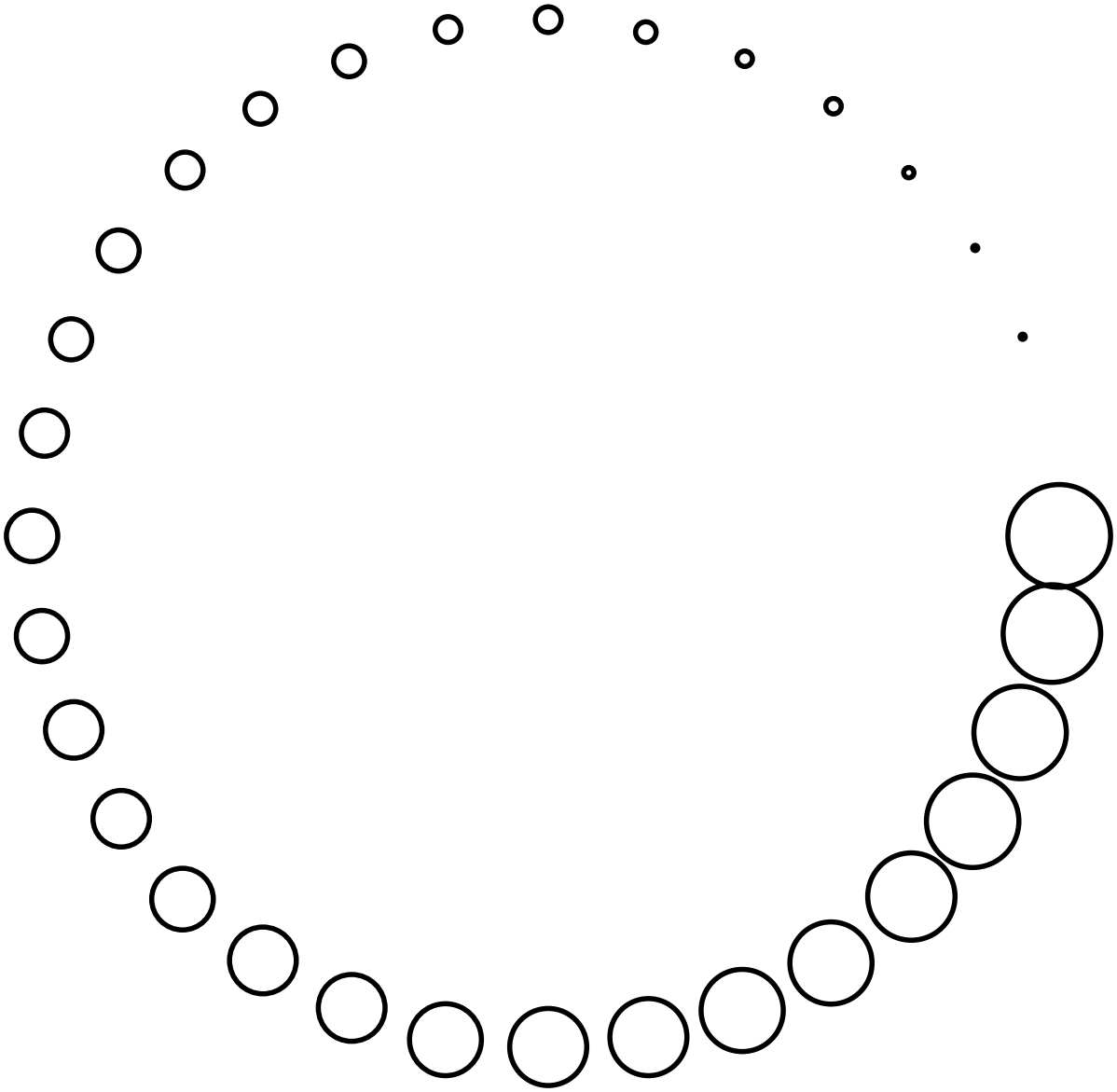

Figure 9.13: A circle created by composing smaller components.

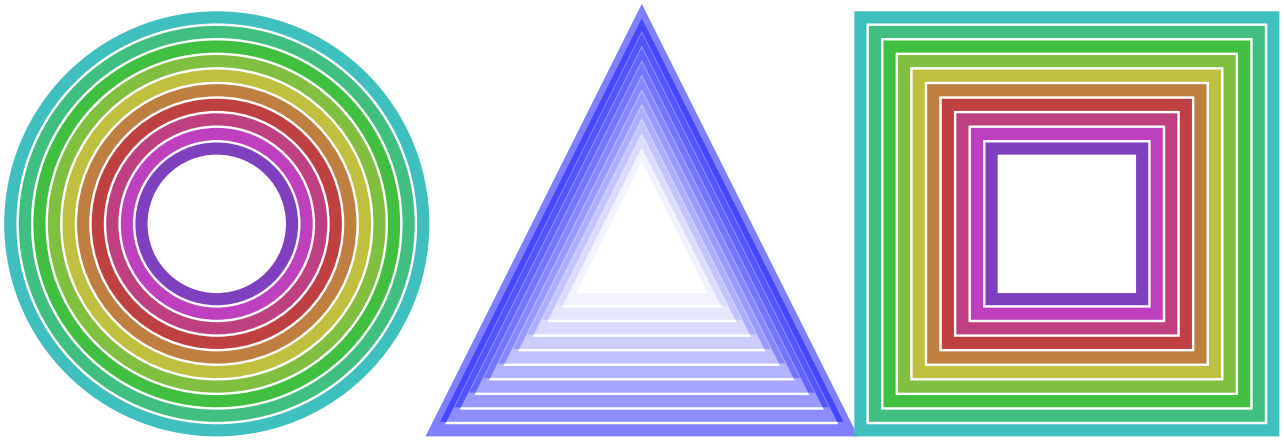

Figure 9.14: Colors and Shapes

```
  Image.circle(50 + 5*n).strokeColor(Color.red)

val redCircles: Image =
  concentricShapes(10, redCircle _)
```

Exercises

The Colour and the Shape

Starting with the code below we are going to write color and shape functions to produce the image shown in fig. 9.14.

```
def concentricShapes(count: Int, singleShape: Int => Image): Image =
  count match {
    case 0 => Image.empty
    case n => singleShape(n).on(concentricShapes(n-1, singleShape))
  }
```

The `concentricShapes` method is equivalent to the `concentricCircles` method from previous exercises. The main difference is that we pass in the definition of `singleShape` as a parameter.

Let's think about the problem a little. We need to do two things:

1. write an appropriate definition of `singleShape` for each of the three shapes in the target image; and
2. call `concentricShapes` three times, passing in the appropriate definition of `singleShape` each time and putting the results `beside` one another.

Let's look at the definition of the `singleShape` parameter in more detail. The type of the parameter is `Int => Image`, which means a function that accepts an `Int` parameter and returns an `Image`. We can declare a method of this type as follows:

```
def outlinedCircle(n: Int): Image =
  Image.circle(n * 10)
```

We can convert this method to a function, and pass it to `concentricShapes` to create an image of concentric black outlined circles:

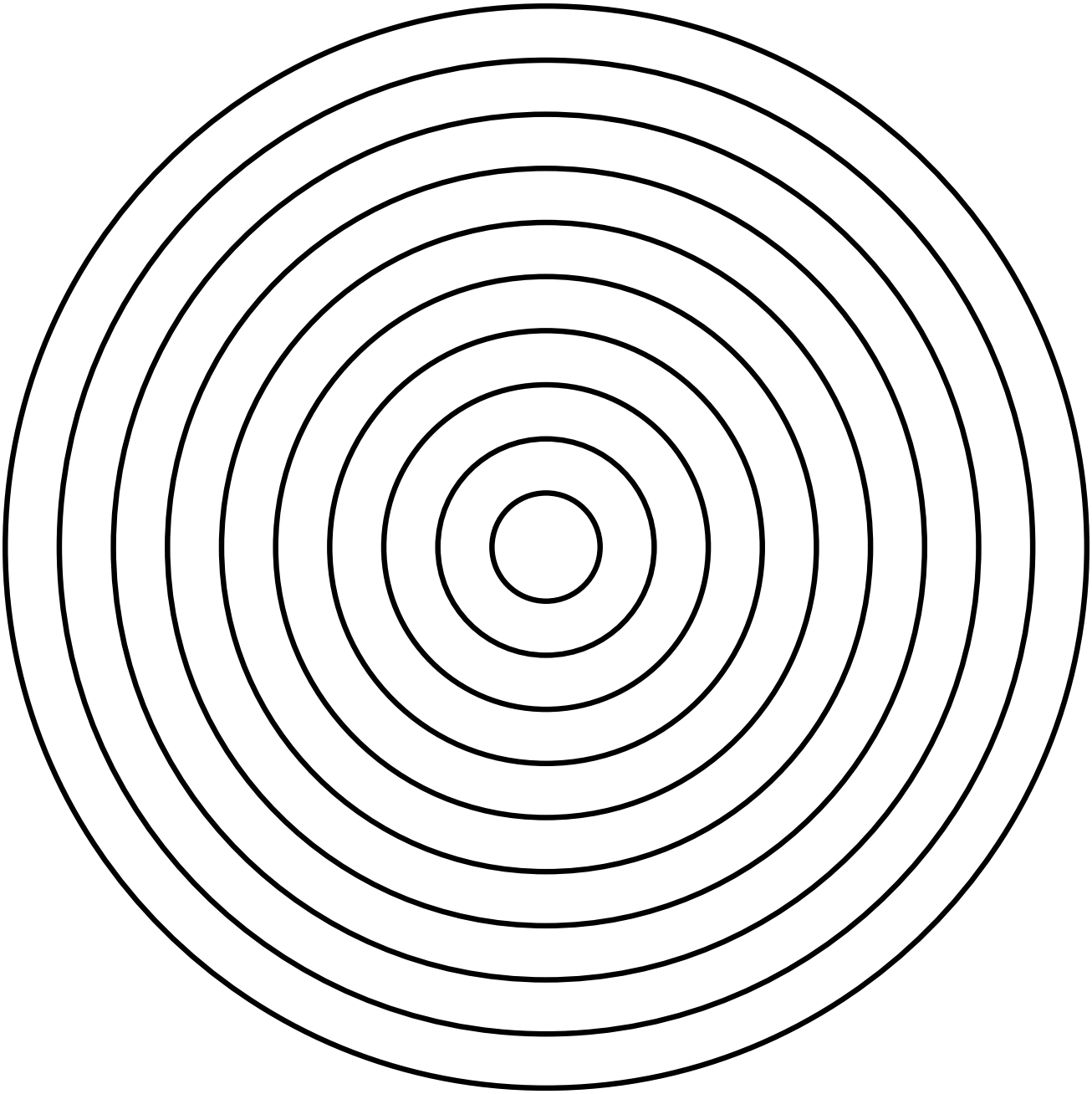

Figure 9.15: Many outlined circles

```
concentricShapes(10, outlinedCircle _)
```

This produces the output shown in fig. 9.15.

The rest of the exercise is just a matter of copying, renaming, and customising this function to produce the desired combinations of colours and shapes:

```
def circleOrSquare(n: Int) =
  if(n % 2 == 0) Image.rectangle(n*20, n*20) else Image.circle(n*10)

concentricShapes(10, outlinedCircle).beside(concentricShapes(10, circleOrSquare))
```

See fig. 9.16 for the output.

For extra credit, when you've written your code to create the sample shapes above, refactor it so you have two sets of base functions—one to produce colours and one to produce shapes. Combine these functions using a

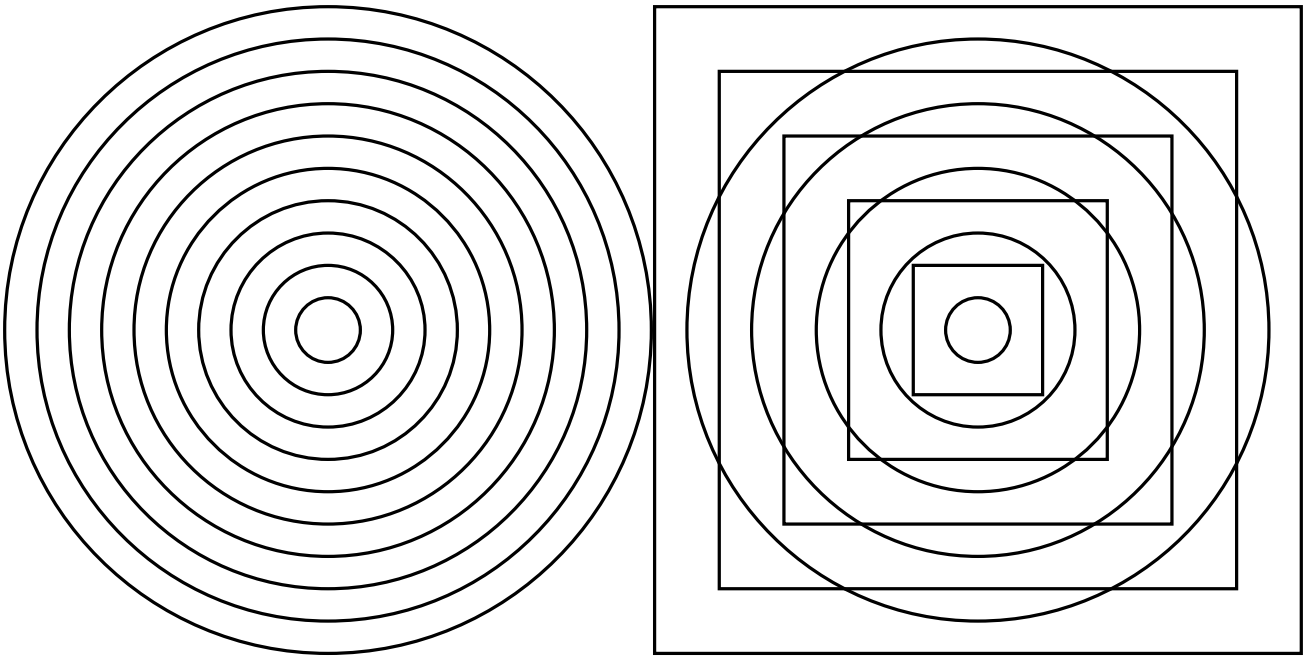

Figure 9.16: Many outlined circles beside many circles and squares

combinator as follows, and use the result of the combinator as an argument to `concentricShapes`

```
def colored(shape: Int => Image, color: Int => Color): Int => Image =
  (n: Int) => ???
```

See the solution

More Shapes

The `concentricShapes` methods takes an `Int => Image` function, and we can construct such as function using `sample`, the parametric curves we created earlier, and the various utilities we have created along the way. There is an example is fig. 9.17.

The code to create this is below.

```
def dottyCircle(n: Int): Image =
  sample(
    72,
    parametricCircle.andThen(scale(100 + n * 24)).andThen(growingDot)
  )

concentricShapes(10, colored(dottyCircle, spinning))
```

Use the techniques we've seen so far to create a picture of your choosing (perhaps similar to the flower with which we started the chapter). No solution here—there is no right or wrong answer.

9.9 Exercises

Now we are chock full of knowledge about functions, we're going to return to the problem of drawing flowers. We'll be asking you to do more design work here than we have done previously.

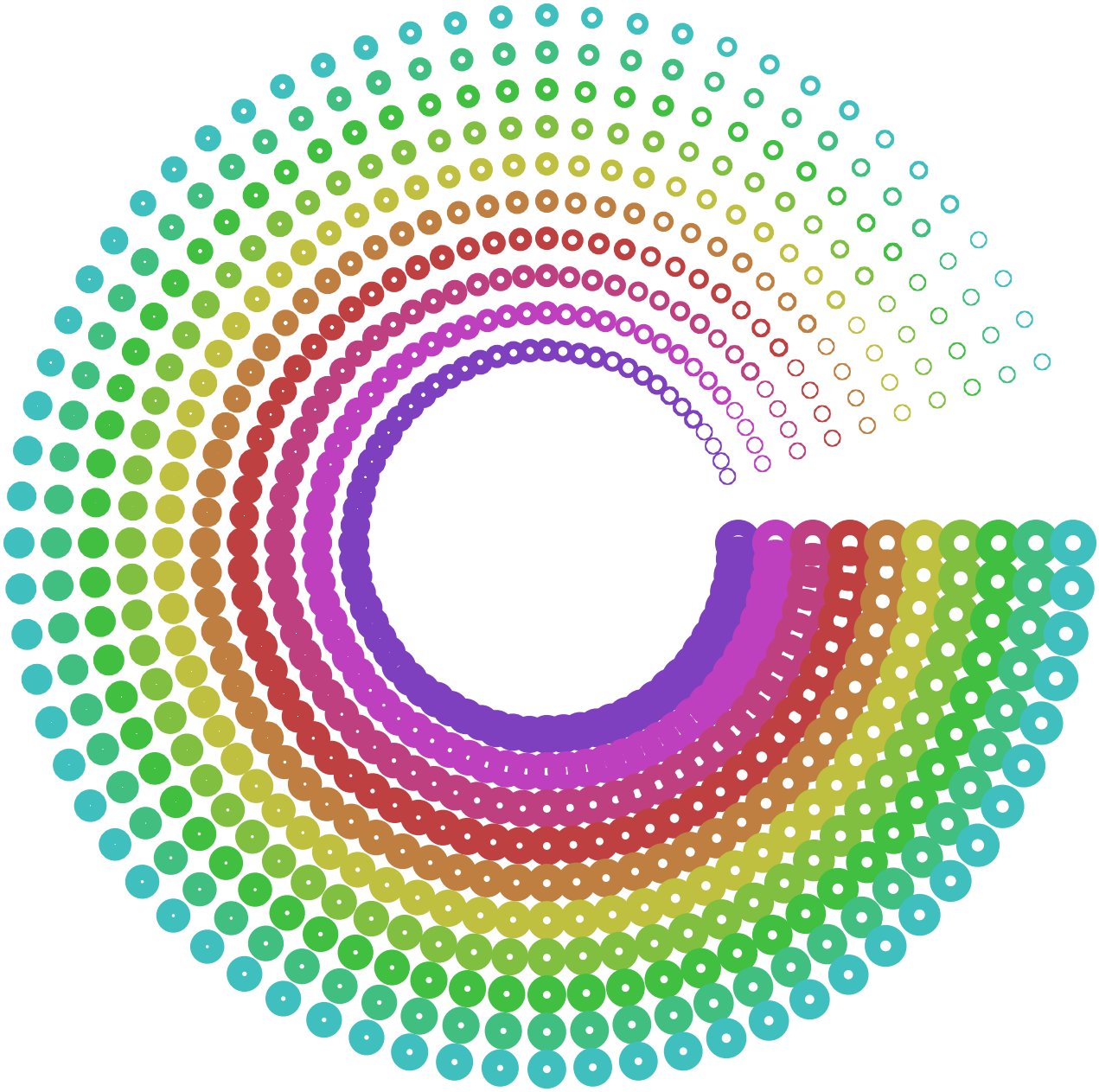

Figure 9.17: Concentric dotty circles

Your task is to break down the task of drawing a flower into small functions that work together. Try to give yourself as much creative freedom as possible, by breaking out each individual component of the task into a function.

Try to do this now. If you run into problems look at our decomposition below.

9.9.1 Components

We've identified two components of drawing flowers:

- the parametric equation; and
- the structural recursion over angles.

What other components might we abstract into functions? What are their types? (This is a deliberately open ended question. Explore!)

See the solution

9.9.2 Combine

Now we've broken out the components we can combine them to create interesting results. Do this now.

See the solution

9.9.3 Experiment

Now experiment with the creative possibilities open to you!

See the solution

9.10 Conclusions

In this chapter we looked at functions. We saw that functions are, unlike methods *first-class values*. This means we can pass functions to methods (or other functions), return them from methods and functions, and give them a name using `val`.

We saw that, because functions are values, we can *compose* them. This means to create new functions by combining existing functions. We saw a few different ways of doing this. Function composition allowed us to build a toolbox of useful functions that we could then combine to create interesting results. Composition is one of the key ideas in functional programming. Doodle is another example of composition: we combine `Images` using methods like `on` and `above` to create new `Images`.

Chapter 10

Shapes, Sequences, and Stars

In this chapter we will learn how to build our own shapes out of the primitve lines and curves that make up the triangles, rectangles, and circles we've used so far. In doing so we'll learn how to represent sequences of data, and manipulate such sequences using higher-order functions that abstract over structural recursion. That's quite a lot of jargon, but we hope you'll see it's not as difficult as it sounds!

> If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.
>
> ```
> import doodle.core._
> import doodle.image._
> import doodle.image.syntax._
> import doodle.image.syntax.core._
> import doodle.java2d._
> ```

10.1 Paths

All shapes in Doodle are ultimately represented as paths. You can think of a path as giving a sequence of movements for an imaginary pen, starting from the local origin. Pen movements come in three varieties:

- moving the pen to a point without drawing a line;
- drawing a straight line from the current position to a point; and
- drawing a Bezier curve from the current position to a point, with the shape of the curve determined by two *control points*.

Paths themselves come in two varieties:

- open paths, where the end of the path is not necessarily the starting point; and
- closed paths, that end where they begin—and if the path doesn't end where it started a line will be inserted to make it so.

The picture in fig. 10.1 illustrates the components that can make up a path, and shows the difference between open and closed paths.

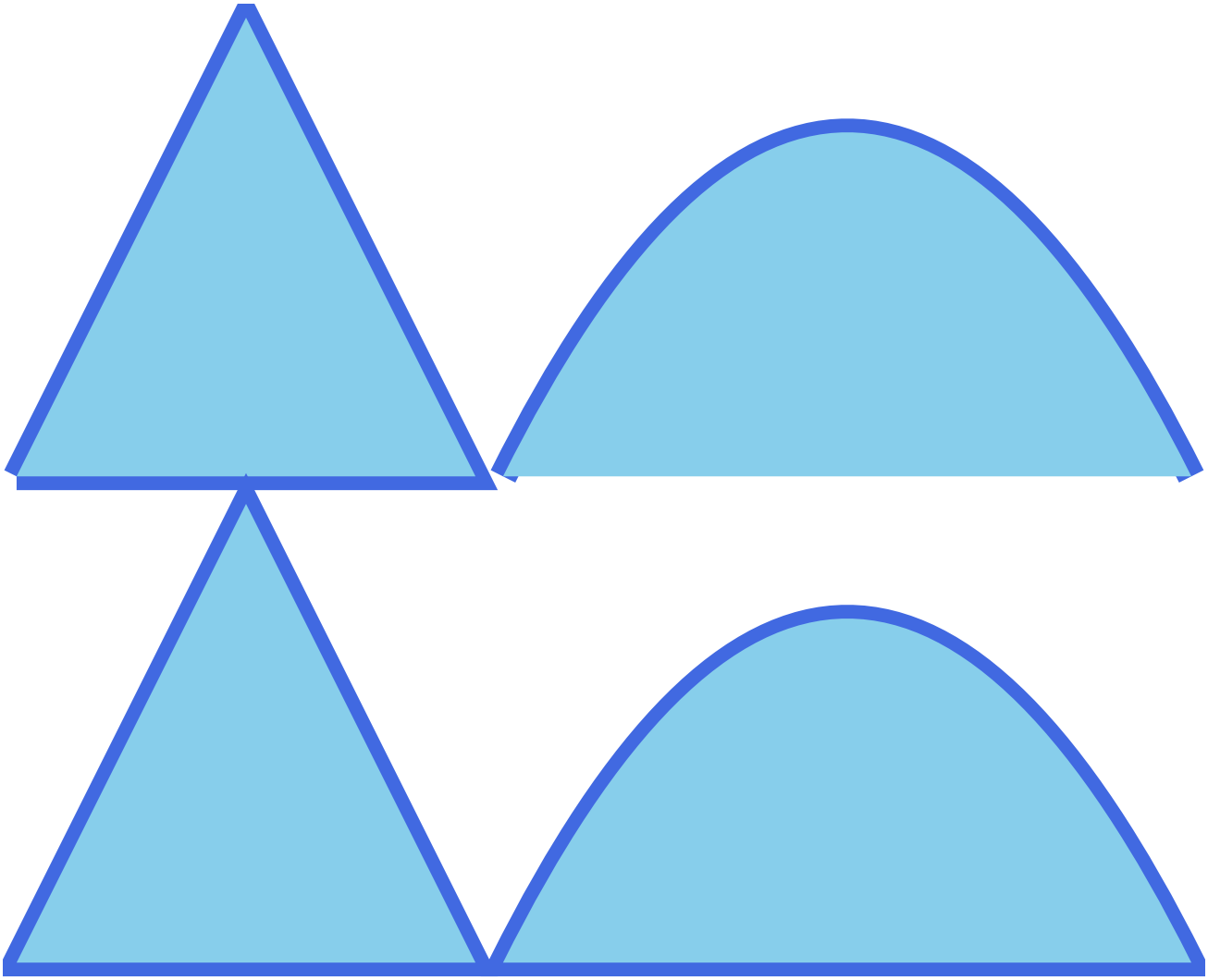

Figure 10.1: The same paths draw as open (top) and closed (bottom) paths. Notice how the open triangle is not properly joined at the bottom left, and the closed curve inserts a straight line to close the shape.

10.1.1 Creating Paths

Now we know about paths, how do we create them in Doodle? Here's the code that created fig. **??**.

```
import doodle.core.PathElement._

val triangle =
  List(
    lineTo(Point(50, 100)),
    lineTo(Point(100, 0)),
    lineTo(Point(0, 0))
  )

val curve =
  List(curveTo(Point(50, 100), Point(100, 100), Point(150, 0)))

def style(image: Image): Image =
  image.strokeWidth(6.0)
    .strokeColor(Color.royalBlue)
    .fillColor(Color.skyBlue)

val openPaths =
  style(Image.openPath(triangle).beside(Image.openPath(curve)))

val closedPaths =
  style(Image.closedPath(triangle).beside(Image.closedPath(curve)))

val paths = openPaths.above(closedPaths)
```

From this code we can see we create paths using the `openPath` and `closePath` methods on `Image`, just as we create other shapes. A path is created from a `List` of `PathElement`. The different kinds of `PathElement` are created by calling methods on the `PathElement` object, as described in tbl. 10.1. In the code above we used the declaration `import doodle.core.PathElement._` to make all the methods on `PathElement` available in the local scope.

Table 10.1: How to create the three different types of `PathElement`.

Method	Description	Example
`PathElement.moveTo(Point)`	Move the pen to `Point` without drawing.	`PathElement.moveTo(Point(1, 1))`
`PathElement.lineTo(Point)` ‘	Draw a straight line to Point‘	`PathElement.lineTo(Point(2, 2))`
`PathElement.curveTo(Point, Point, Point)`	Draw a curve. The first two points specify control points and the last point is where the curve ends.	`PathElement.curveTo(Point(1,0), Point(0,1), Point(1,1))`

Constructing a `List` is straight-forward: we just call `List` with the elements we want the list to contain. Here are some examples.

```
// List of Int
List(1, 2, 3)
// res0: List[Int] = List(1, 2, 3)

// List of Image
```

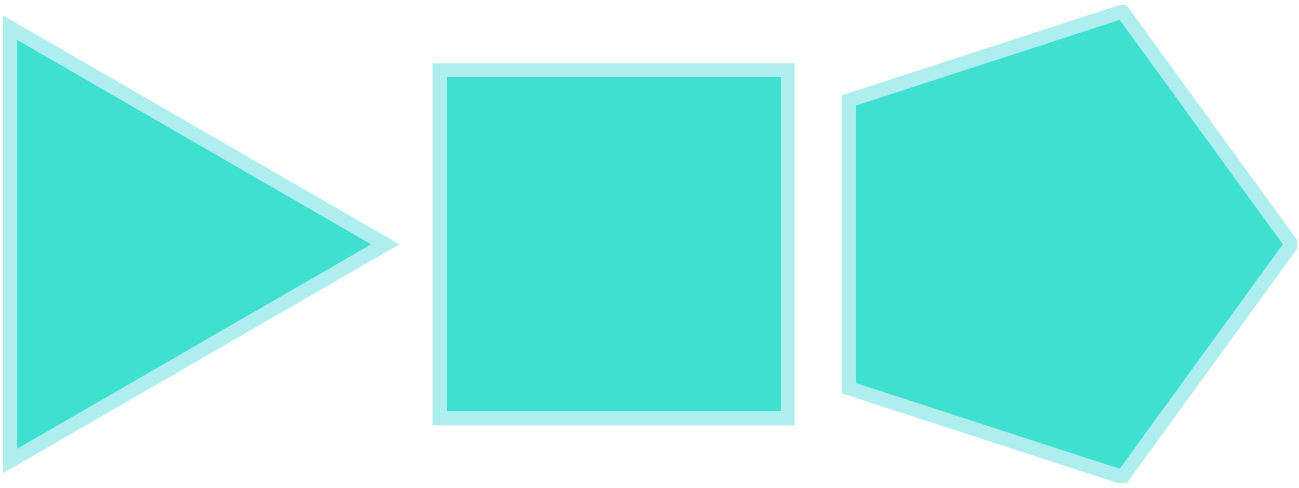

Figure 10.2: A triangle, square, and pentagon, defined using paths.

```
List(Image.circle(10), Image.circle(20), Image.circle(30))
// res1: List[Image] = List(Circle(10.0), Circle(20.0), Circle(30.0))

// List of Color
List(Color.paleGoldenrod, Color.paleGreen, Color.paleTurquoise)
// res2: List[Color] = List(
//   RGBA(UnsignedByte(110), UnsignedByte(104), UnsignedByte(42), Normalized(1.0)),
//   RGBA(UnsignedByte(24), UnsignedByte(123), UnsignedByte(24), Normalized(1.0)),
//   RGBA(UnsignedByte(47), UnsignedByte(110), UnsignedByte(110), Normalized(1.0))
// )
```

Notice the type of a `List` includes the type of the elements, written in square brackets. So the type of a list of integers is written `List[Int]` and a list of `PathElement` is written `List[PathElement]`.

Exercises

Polygons

Create paths to define a triangle, square, and pentagon. Your image might look like fig. 10.2. *Hint:* you might find it easier to use polar coordinates to define the polygons.

See the solution

Curves

Repeat the exercise above, but this time use curves instead of straight lines to create some interesting shapes. Our curvy polygons are shown in fig. 10.3. *Hint:* you'll have an easier time if you generalise into a method your code for creating a curve.

See the solution

10.2 Working with Lists

At this point you might be thinking it would be nice to create a method to draw polygons rather than constructing each one by hand. There is clearly a repeating pattern to their construction and we would be able to generalise

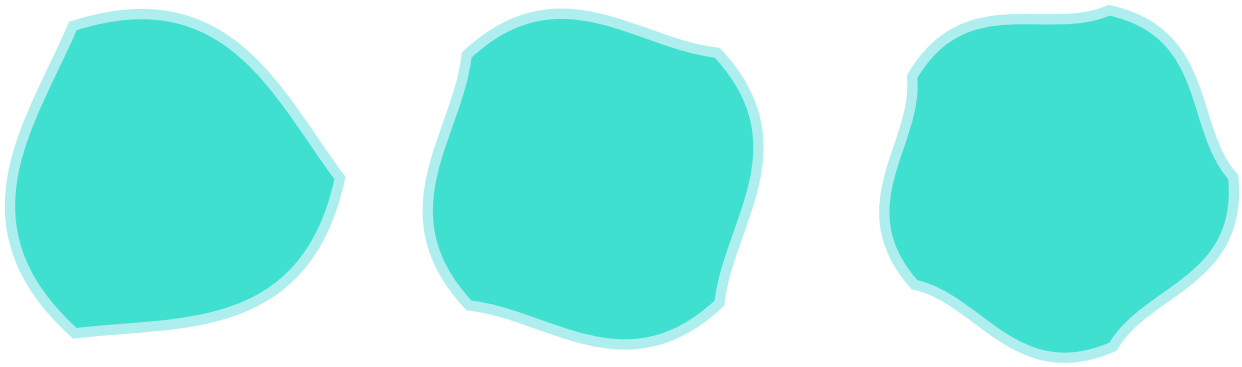

Figure 10.3: A curvy triangle, square, and polygon, defined using paths.

this pattern if we knew how to create a list of arbitrary size. It's time we learned more about building and manipulating lists.

10.2.1 The Recursive Structure of Lists

You'll recall when we introduced structural recursion over the natural numbers we said we could transform their recursive structure into any other recursive structure. We demonstrated this for concentric circles and a variety of other patterns.

Lists have a recursive structure, and one that is very similar to the structure of the natural numbers. A `List` is

- the empty list `Nil`; or
- a pair of an element `a` and a `List`, written `a :: tail`, where `tail` is the rest of the list.

For example, we can write out the list `List(1, 2, 3, 4)` in its "long" form as

```
1 :: 2 :: 3 :: 4 :: Nil
// res0: List[Int] = List(1, 2, 3, 4)
```

Notice the similarity to the natural numbers. Earlier we noted we can expand the structure of a natural number so we could write, say, 3 as `1 + 1 + 1 + 0`. If we replace + with `::` and `0` with `Nil` we get the `List 1 :: 1 :: 1 :: Nil`.

What does this mean? It means we can easily transform a natural number into a `List` using our familiar tool of structural recursion[1]. Here's a very simple example, which given a number builds a list of that length containing the `String` "Hi".

```
def sayHi(length: Int): List[String] =
  length match {
    case 0 => Nil
    case n => "Hi" :: sayHi(n - 1)
  }

sayHi(5)
```

[1]This connection goes deeper. We can abstract the idea of things that can be "added" into a concept called a monoid, and a list represents a particular type of monoid called the free monoid. We aren't going to work with this abstraction in Creative Scala but you're encouraged to explore on your own!

```
// res1: List[String] = List("Hi", "Hi", "Hi", "Hi", "Hi")
```

The code here is transforming:

- `0` to `Nil`, for the base case; and
- `n` (which, remember, we think of as `1 + m`) to `"Hi" :: sayHi(n - 1)`, transforming `1` to `"Hi"`, `+` to `::`, and recursing as usual on `m` (which is `n - 1`).

This recursive structure also means we can transform lists into other recursive structures, such a natural number, different lists, chessboards, and so on. Here we increment every element in a list—that is, transform a list to a list—using structural recursion.

```
def increment(list: List[Int]): List[Int] =
  list match {
    case Nil => Nil
    case hd :: tl => (hd + 1) :: increment(tl)
  }

increment(List(1, 2, 3))
// res2: List[Int] = List(2, 3, 4)
```

Here we sum the elements of a list of integers—that is, transform a list to a natural number—using structural recursion.

```
def sum(list: List[Int]): Int =
  list match {
    case Nil => 0
    case hd :: tl => hd + sum(tl)
  }

sum(List(1, 2, 3))
// res3: Int = 6
```

Notice when we take a `List` apart with pattern matching we use the same `hd :: tl` syntax we use when we construct it. This is an intentional symmetry.

10.2.2 Type Variables

What about finding the length of a list? We know we can use our standard tool of structural recursion to write the method. Here's the code to calculate the length of a `List[Int]`.

```
def length(list: List[Int]): Int =
  list match {
    case Nil => 0
    case hd :: tl => 1 + length(tl)
  }
```

Note that we don't do anything with the elements of the list—we don't really care about their type. Using the same code skeleton can just as easily calculate the length of a `List[Int]` as a `List[HairyYak]` but we don't currently know how to write down the type of a list where we don't care about the type of the elements.

Scala lets us write methods that can work with any type, by using what is called a *type variable*. A type variable is written in square brackets like `[A]` and comes after the method name and before the parameter list. A type variable can stand in for any specific type, and we can use it in the parameter list or result type to indicate some type that we don't know up front. For example, here's how we can write `length` so it works with lists of any type.

```
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0
    case hd :: tl => 1 + length(tl)
  }
```

Structural Recursion over a List

A `List` of elements of type `A` is:

- the empty list `Nil`; or
- an element `a` of type `A` and a `tail` of type `List[A]`: `a :: tail`

The structural recursion skeleton for transforming `list` of type `List[A]` to some type `B` has shape

```
def doSomething[A,B](list: List[A]): B =
  list match {
    case Nil => ??? // Base case of type B here
    case hd :: tl => f(hd, doSomething(tl))
  }
```

where `f` is a problem specific method combining `hd` and the result of the recursive call to produce something of type `B`.

Exercises

Building Lists

In these exercises we get some experience constructing lists using structural recursion on the natural numbers.

Write a method called `ones` that accepts an `Int` `n` and returns a `List[Int]` with length `n` and every element `1`. For example

```
ones(3)
// res5: List[Int] = List(1, 1, 1)
```

See the solution

Write a method `descending` that accepts an natural number `n` and returns a `List[Int]` containing the natural numbers from `n` to `1` or the empty list if `n` is zero. For example

```
descending(0)
// res8: List[Int] = List()
descending(3)
// res9: List[Int] = List(3, 2, 1)
```

See the solution

Write a method `ascending` that accepts a natural number `n` and returns a `List[Int]` containing the natural numbers from `1` to `n` or the empty list if `n` is zero.

```
ascending(0)
// res13: List[Int] = List()
ascending(3)
// res14: List[Int] = List(1, 2, 3)
```

See the solution

Create a method `fill` that accepts a natural number `n` and an element `a` of type `A` and constructs a list of length `n` where all elements are `a`.

```
fill(3, "Hi")
// res18: List[String] = List("Hi", "Hi", "Hi")
fill(3, Color.blue)
// res19: List[Color] = List(
//   RGBA(
//     UnsignedByte(-128),
//     UnsignedByte(-128),
//     UnsignedByte(127),
//     Normalized(1.0)
//   ),
//   RGBA(
//     UnsignedByte(-128),
//     UnsignedByte(-128),
//     UnsignedByte(127),
//     Normalized(1.0)
//   ),
//   RGBA(
//     UnsignedByte(-128),
//     UnsignedByte(-128),
//     UnsignedByte(127),
//     Normalized(1.0)
//   )
// )
```

See the solution

Transforming Lists

In these exercises we practice the other side of list manipulation—transforming lists into elements of other types (and sometimes, into different lists).

Write a method `double` that accepts a `List[Int]` and returns a list with each element doubled.

```
double(List(1, 2, 3))
// res23: List[Int] = List(2, 4, 6)
double(List(4, 9, 16))
// res24: List[Int] = List(8, 18, 32)
```

See the solution

Write a method `product` that accepts a `List[Int]` and calculates the product of all the elements.

```
product(Nil)
// res28: Int = 1
product(List(1,2,3))
// res29: Int = 6
```

See the solution

Write a method `contains` that accepts a `List[A]` and an element of type `A` and returns true if the list contains the element and false otherwise.

```
contains(List(1,2,3), 3)
// res33: Boolean = true
contains(List("one", "two", "three"), "four")
// res34: Boolean = false
```

See the solution

Write a method `first` that accepts a `List[A]` and an element of type `A` and returns the first element of the list if it is not empty and otherwise returns the element of type `A` passed as a aprameter.

```
first(Nil, 4)
// res38: Int = 4
first(List(1,2,3), 4)
// res39: Int = 1
```

See the solution

Challenge Exercise: Reverse

Write a method `reverse` that accepts a `List[A]` and reverses the list.

```
reverse(List(1, 2, 3))
// res43: List[Int] = List(3, 2, 1)
reverse(List("a", "b", "c"))
// res44: List[String] = List("c", "b", "a")
```

See the solution

Polygons!

At last, let's return to our example of drawing polygons. Write a method `polygon` that accepts the number of sides of the polygon and the starting rotation and produces a `Image` representing the specified regular polygon. *Hint:* use an internal accumulator.

Use this utility to create an interesting picture combining polygons. Our rather unimaginative example is in fig. 10.4. We're sure you can do better.

See the solution

10.3 Transforming Sequences

We've seen that using structural recursion we can create and transform lists. This pattern is simple to use and to understand, but it requires we write the same skeleton time and again. In this section we'll learn that we can replace structural recursion in some cases by using a method on `List` called `map`. We'll also see that other useful datatypes provide this method and we can use it as a common interface for manipulating sequences.

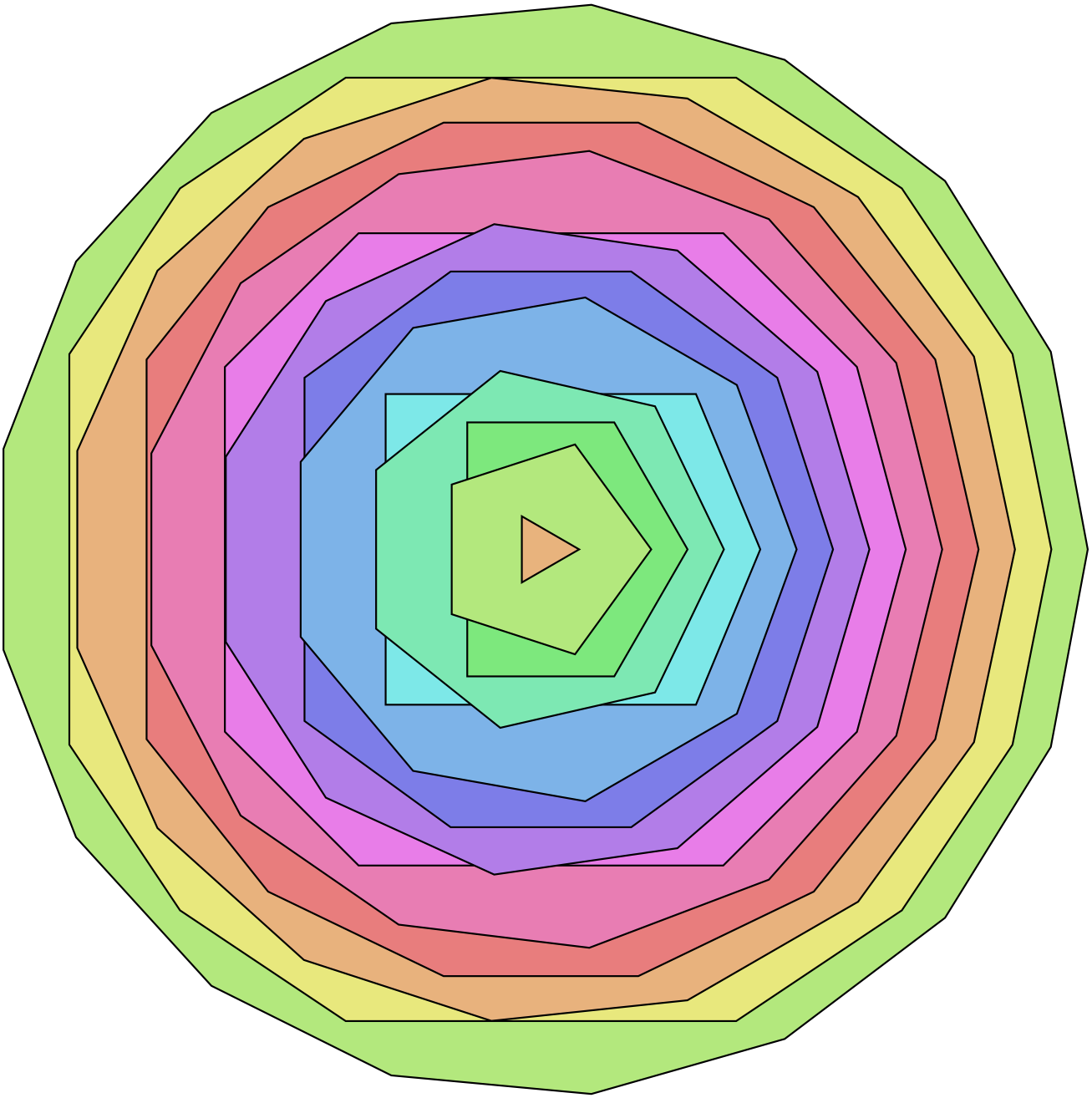

Figure 10.4: Concentric polygons with pastel gradient fill.

10.3.1 Transforming the Elements of a List

In the previous section we saw several examples where we transformed one list to another. For example, we incremented the elements of a list with the following code.

```
def increment(list: List[Int]): List[Int] =
  list match {
    case Nil => Nil
    case hd :: tl => (hd + 1) :: tl
  }

increment(List(1, 2, 3))
// res0: List[Int] = List(2, 2, 3)
```

In this example the *structure* of the list doesn't change. If we start with three elements we end with three elements. We can abstract this pattern in a method called `map`. If we have a list with elements of type `A`, we pass `map` a function of type `A => B` and we get back a list with elements of type `B`. For example, we can implement `increment` using `map` with the function `x => x + 1`.

```
def increment(list: List[Int]): List[Int] =
  list.map(x => x + 1)

increment(List(1, 2, 3))
// res2: List[Int] = List(2, 3, 4)
```

Each element is transformed by the function we pass to `map`, in this case `x => x + 1`. With `map` we can transform the elements, but we cannot change the number of elements in the list.

We find a graphical notation helps with understanding `map`. If we had some type `Circle` we can draw a `List[Circle]` as a box containing a circle, as shown in fig. 10.5.

Now we can draw an equation for `map` as in fig. 10.6. If you prefer symbols instead of pictures, the picture is showing that `List[Circle].map(Circle => Triangle) = List[Triangle]`. One feature of the graphical representation is it nicely illustrates that `map` cannot create a new "box", which represents the structure of the list—or more concretely the number of elements and their order.

The graphical drawing of `map` exactly illustrates what holds at the type level for `map`. If we prefer we can write it down symbolically:

```
List[A].map(A => B) = List[B]
```

The left hand side of the equation has the type of the list we map and the function we use to do the mapping. The right hand is the type of the result. We can perform algebra with this representation, substituting in concrete types from our program.

10.3.2 Transforming Sequences of Numbers

We have also written a lot of methods that transform a natural number to a list. We briefly discussed how we can represent a natural number as a list. `3` is equivalent to `1 + 1 + 1 + 0`, which in turn we could represent as `List(1, 1, 1)`. So what? Well, it means we could write a lot of the methods that accepts natural numbers as methods that worked on lists.

For example, instead of

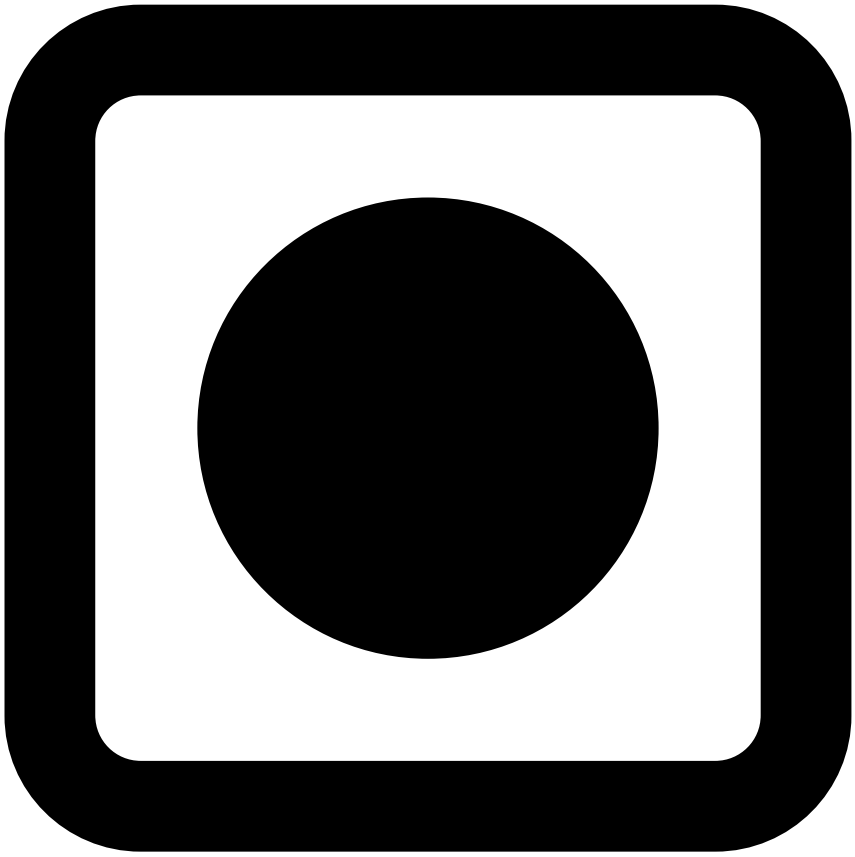

Figure 10.5: A `List[Circle]` representing by a circle within a box

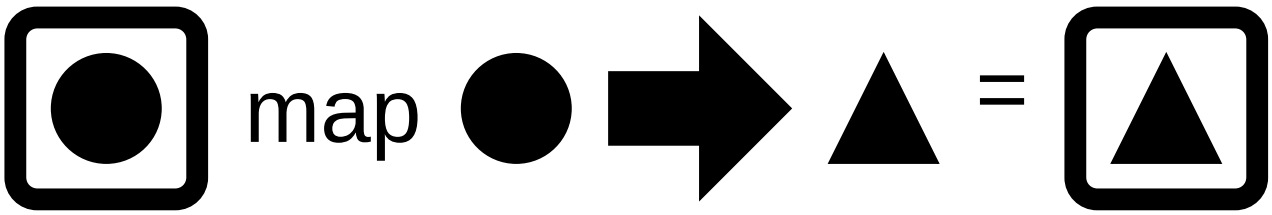

Figure 10.6: `map` shown graphically

```
def fill[A](n: Int, a: A): List[A] =
  n match {
    case 0 => Nil
    case n => a :: fill(n-1, a)
  }

fill(3, "Hi")
// res3: List[String] = List("Hi", "Hi", "Hi")
```

we could write

```
def fill[A](n: List[Int], a: A): List[A] =
  n.map(x => a)

fill(List(1, 1, 1), "Hi")
// res5: List[String] = List("Hi", "Hi", "Hi")
```

The implementation of this version of `fill` is more convenient to write, but it is much less convenient for the user to call it with `List(1, 1, ,1)` than just writing 3.

If we want to work with sequences of numbers we are better off using `Ranges`. We can create these using the `until` method of `Int`.

```
0 until 10
// res6: Range = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

`Ranges` have a `by` method that allows us to change the step between consecutive elements of the range:

```
0 until 10 by 2
// res7: Range = Range(0, 2, 4, 6, 8)
```

`Ranges` also have a `map` method just like `List`

```
(0 until 3) map (x => x + 1)
// res8: collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3)
```

You'll notice the result has type `IndexedSeq` and is implemented as a `Vector`—two types we haven't seen yet. We can treat an `IndexedSeq` much like a `List`, but for simplicity we can convert a `Range` or an `IndexedSeq` to a `List` using the `toList` method.

```
(0 until 7).toList
// res9: List[Int] = List(0, 1, 2, 3, 4, 5, 6)
(0 until 3).map(x => x + 1).toList
// res10: List[Int] = List(1, 2, 3)
```

With `Ranges` in our toolbox we can write `fill` as

```
def fill[A](n: Int, a: A): List[A] =
  (0 until n).toList.map(x => a)

fill(3, "Hi")
// res12: List[String] = List("Hi", "Hi", "Hi")
```

10.3.3 Ranges over Doubles

If we try to create a `Range` over `Double` we get an error.

```
0.0 to 10.0 by 1.0
// error: No warnings can be incurred under -Xfatal-warnings.
```

There are two ways around this. We can use an equivalent `Range` over `Int`. In this case we could just write

```
0 to 10 by 1
```

We can use the `.toInt` method to convert a `Double` to an `Int` if needed.

Alternatively we can write a `Range` using `BigDecimal`.

```
import scala.math.BigDecimal
BigDecimal(0.0) to 10.0 by 1.0
```

`BigDecimal` has methods `doubleValue` and `intValue` to get `Double` and `Int` values respectively.

```
BigDecimal(10.0).doubleValue()
// res16: Double = 10.0
BigDecimal(10.0).intValue()
// res17: Int = 10
```

Exercises

10.3.3.0.1 Ranges, Lists, and map Using our new tools, reimplement the following methods.

Write a method called `ones` that accepts an `Int` n and returns a `List[Int]` with length n and every element is 1. For example

```
ones(3)
// res18: List[Int] = List(1, 1, 1)
```

See the solution

Write a method `descending` that accepts an natural number n and returns a `List[Int]` containing the natural numbers from n to 1 or the empty list if n is zero. For example

```
descending(0)
// res21: List[Int] = List()
descending(3)
// res22: List[Int] = List(3, 2, 1)
```

See the solution

Write a method `ascending` that accepts a natural number n and returns a `List[Int]` containing the natural numbers from 1 to n or the empty list if n is zero.

```
ascending(0)
// res26: List[Int] = List()
ascending(3)
// res27: List[Int] = List(1, 2, 3)
```

See the solution

Write a method `double` that accepts a `List[Int]` and returns a list with each element doubled.

```
double(List(1, 2, 3))
// res31: List[Int] = List(2, 4, 6)
double(List(4, 9, 16))
// res32: List[Int] = List(8, 18, 32)
```

See the solution

10.3.3.0.2 Polygons, Again! Using our new tools, rewrite the `polygon` method from the previous section.

See the solution

10.3.3.0.3 Challenge Exercise: Beyond Map Can we use `map` to replace all uses of structural recursion? If not, can you characterise the problems that we can't implement with `map` but can implement with general structural recursion over lists?

See the solution

10.3.4 Tools with Ranges

We've seen the `until` method to construct `Ranges`, and the `by` method to change the increment in a `Range`. There is one more method that will be useful to know about: `to`. This constructs a `Range` like `until` but the `Range` includes the endpoint. Compare

```
1 until 5
// res37: Range = Range(1, 2, 3, 4)
1 to 5
// res38: Range.Inclusive = Range.Inclusive(1, 2, 3, 4, 5)
```

In technical terms, the `Range` constructed with `until` is a *half-open interval*, while the range constructed with `to` is an *open interval*.

Exercises

10.3.4.0.1 Using Open Intervals Write a method `ascending` that accepts a natural number `n` and returns a `List[Int]` containing the natural numbers from 1 to `n` or the empty list if `n` is zero. *Hint:* use `to`

```
ascending(0)
// res39: List[Int] = List()
ascending(3)
// res40: List[Int] = List(1, 2, 3)
```

See the solution

10.4 My God, It's Full of Stars!

Let's use our new tools to draw some stars. For the purpose of this exercise let's assume that a star is a polygon with `p` points. However, instead of connecting each point to its neighbours, we'll connect them to the `nth` point around the circumference.

For example, fig. 10.7 shows stars with `p=11` and `n=1 to 5`. `n=1` produces a regular polygon while values of `n` from 2 upwards produce stars with increasingly sharp points:

Write code to draw the diagram above. Start by writing a method to draw a `star` given `p` and `n`:

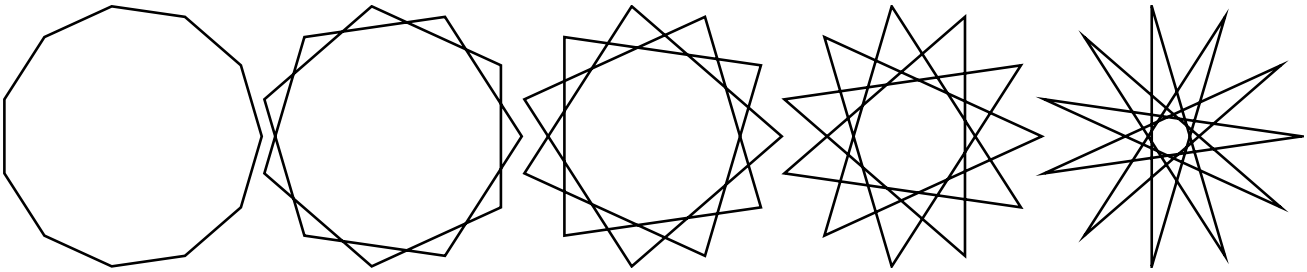

Figure 10.7: Stars with p=11 and n=1 `to` 5

```
def star(p: Int, n: Int, radius: Double): Image =
  ???
```

Hint: use the same technique we used for `polygon` previously.

See the solution

Using structural recursion and `beside` write a method `allBeside` with the signature

```
def allBeside(images: List[Image]): Image =
  ???
```

We'll use `allBeside` to create the row of stars. To create the picture we only need to use values of `skip` from 1 to `sides/2` rounded down. For example:

```
allBeside(
  (1 to 5).toList map { skip =>
    star(11, skip, 100)
  }
)
```

See the solution

When you've finished your row of stars, try constructing a larger image from different values of p and n. There is an example in fig. 10.8. *Hint:* You will need to create a method `allAbove` similar to `allBeside`.

See the solution

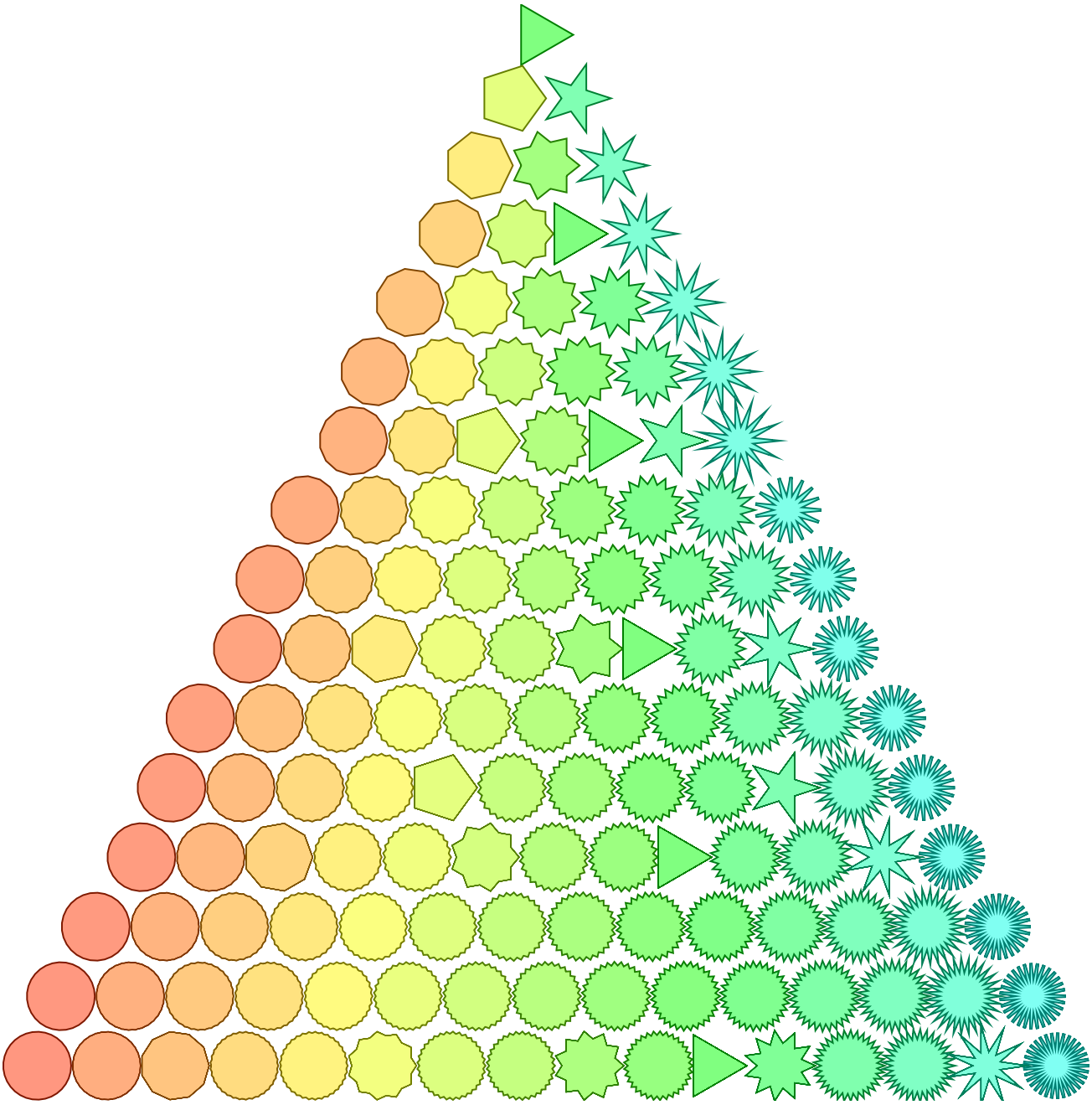

Figure 10.8: Stars with `p=3 to 33 by 2` and `n=1 to p/2`

Chapter 11

Animation and Fireworks

This chapter consolidates some of the topics we've just learned, namely functions and lists, and introduces animations as a new way of working with them. This will expand how we think about these programming concepts, and give us a new creative outlet with which to explore them.

> If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.
>
> ```
> import doodle.core._
> import doodle.image._
> import doodle.image.syntax._
> import doodle.image.syntax.core._
> import doodle.java2d._
> import doodle.reactor._
> import doodle.interact.syntax._
> ```

11.1 Reactors

We will create animations using a tool called a *reactor*. A reactor allows us to write an animation in terms of three things, and one optiona thing:

- some initial value, such as a `Point`;
- an update function that transforms the value into its next value every clock tick, such as moving the `Point`;
- a rendering function that turns the value into an `Image`; and
- an optional condition that determines when the animation stop.

Here's an example that moves a circle from left to right, stopping when the the circle gets to the point (300, 0).

```
val travellingCircle =
  Reactor.init(Point(-300, 0))
    .onTick(pt => Point(pt.x + 1, pt.y))
    .render(pt => Image.circle(10).at(pt))
    .stop(pt => pt.x >= 300)
```

Let's break this down:

- the value we pass to the `init` method is the initial value of animation, which is a point at (-300, 0);

- the function passed to `onTick` is called every clock tick (which is multiple times a second though we can change this if we wish) to move the point along the x-axis;
- the `render` method is passed a function that tells us how to convert the data—the `Point`—into an `Image`; and
- the function passed to `stop` tells the reactor when to stop.

(We could write the `onTick` function as `pt => pt + Vec(1,0)` if we're comfortable with vector arithmetic.)

This constructs a reactor but it does not draw it. To do this we must call the `run` method, passing a `Frame` that tells the reactor how big to make the window it draws on. Here's an example:

```
travellingCircle.run(Frame.size(600, 600))
```

This generates the animation shown in *TODO: render animation*

Here's an another example that moves a circle in a circular orbit. This time the animation has no stopping condition, so it continues forever.

```
val orbitingCircle =
  Reactor.init(Point(0, 300))
    .onTick(pt => pt.rotate(2.degrees))
    .render(pt => Image.circle(10).at(pt))
```

We run this reactor in the same way.

```
orbitingCircle.run(Frame.size(600, 600))
```

This generates the animation shown in *TODO: render animation*

11.1.0.1 Exercise: Rose Curve

Make an animation where an image moves in a rose curve (we saw the rose curve in an earlier chapter). Be as creative as you wish. You might find it fun to change the background of the `Frame` on which you draw the animation; a dark background is often more effective than a light one. You can do this by calling the `background` method on `Frame`. For example, here is how you'd create a 600 by 600 frame with a dark blue background.

```
Frame.size(600, 600).background(Color.midnightBlue)
```

Remember you will need to import `doodle.reactor._` to make the reactor library available.

11.2 Easing Functions

Take a look at the following animation.

```
val bubble =
  Reactor.linearRamp(0, 200)
    .render(r => Image.circle(r))
// bubble: Reactor[Double] = Reactor(
//   0.0,
//   doodle.reactor.Reactor$$$Lambda$9550/2044289989@6ee42a4f,
//   doodle.reactor.Reactor$$$Lambda$9578/901681921@4cc70267,
//   100 milliseconds,
//   <function1>,
```

```
//   doodle.reactor.Reactor$$$Lambda$9579/820706882@19e61b4c
// )
```

Chapter 12

Turtle Algebra and Algebraic Data Types

In this chapter we explore a new way of creating paths—turtle graphics—and learn some new ways of manipulating lists and functions.

> If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.
>
> ```
> import doodle.core._
> import doodle.image._
> import doodle.image.syntax._
> import doodle.image.syntax.core._
> import doodle.java2d._
> ```

12.1 Turtle Graphics

So far our paths have used an absolute coordinate system. For example, if we wanted to draw a square we'd use code like

```
import doodle.core.PathElement._

val path =
  Image.openPath(
    List(moveTo(10,10), lineTo(-10,10), lineTo(-10,-10), lineTo(10, -10), lineTo(10, 10))
  )
```

It's often easier to define paths in terms of relative coordinates, specifying how far we move forward or turn relative to our current location. This is how a turtle graphics system works. Here's an example.

```
import doodle.turtle._
import doodle.turtle.Instruction._

// Create a list of instructions for the turtle
val instructions: List[Instruction] =
  List(forward(10), turn(90.degrees),
       forward(10), turn(90.degrees),
       forward(10), turn(90.degrees),
       forward(10))
```

Figure 12.1: A plant generated using turtle graphics and an L-system.

```
// Ask the turtle to draw these instructions, creating an Image
val image: Image = Turtle.draw(instructions)
```

So where's the turtle in all this? This model was developed in the 60s by Seymour Papert in the programming language Logo. The original Logo could control a robot that drew on paper with a pen. This robot was called a turtle, due to its rounded shape, and way of programming this robot became known as turtle graphics.

Using turtle graphics and another concept, known as an L-system, we can create images that mimic nature such as the plant in fig. 12.1.

12.2 Controlling the Turtle

Let's look over the turtle graphics API, and use it to draw a few different images.

12.2.1 Instructions

We control the turtle by giving it instructions. These instructions are defined as methods on the object `doodle.turtle.Instruction` (similarly to the methods on `doodle.core.Image` that create images).

We can import the methods and then create instructions.

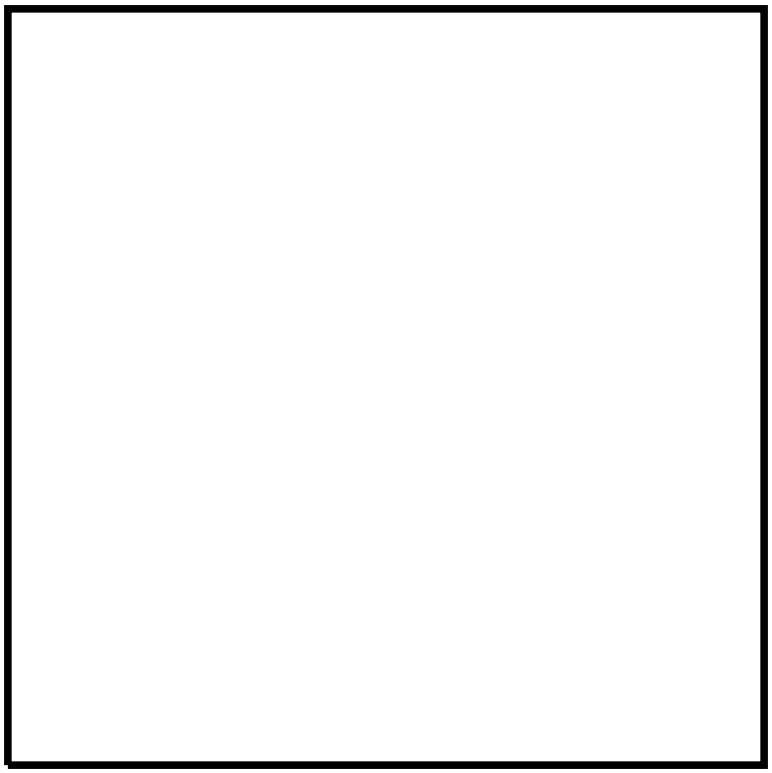

Figure 12.2: A square created via the turtle graphics system.

```
import doodle.turtle._
import doodle.turtle.Instruction._

forward(10)
// res0: Instruction = Forward(10.0)
turn(5.degrees)
// res1: Instruction = Turn(Angle(0.08726646259971647))
```

This doesn't do anything useful unless we assemble these commands into an image. To do so, we create a list of instructions and then ask the turtle (`doodle.turtle.Turtle` to be exact) to draw them to an `Image`.

```
val instructions =
  List(forward(10), turn(90.degrees),
       forward(10), turn(90.degrees),
       forward(10), turn(90.degrees),
       forward(10))

val path = Turtle.draw(instructions)
```

This creates a path—an `Image`—which we can then `draw` in the usual way. This gives the output shown in fig. 12.2. This is not a very exciting image, but we can change color, line width, and so on to create more interesting results.

The complete list of turtle instructions in given in tbl. 12.1

Table 12.1: The instructions understood by the turtle.

Instruction	Description	Example
`forward(distance)`	Move forward the given `distance`, specified as a `Double`.	`forward(100.0)`

Instruction	Description	Example
`turn(angle)`	Turn the given `angle` (an `Angle`) from the current heading.	`turn(10.degrees)`
`branch(instruction, ...)`	Save the current position and heading, draw the given `instructions` , and then return to the saved position to draw the rest of the instructions.	`branch(turn(10.degrees), forward(10))`
`noop`	Do nothing!	`noop`

Exercises

Polygons

In the previous chapter we wrote a method to create a polygon. Reimplement this method using turtle graphics instead. The method header should be something like

```
def polygon(sides: Int, sideLength: Double): Image =
 ???
```

You'll have to do a bit of geometry to work out the correct turn angle, but as that's half the fun we won't spoil it for you.

See the solution

12.2.1.1 The Square Spiral

The square spiral is shown in fig. 12.3. Write a method to create square spirals using turtle graphics.

This task requires a bit more design work than we usually ask of you. You'll have to work out how the square spiral is constructed (hint: it starts at the center) and then create a method to draw one.

See the solution

Turtles vs Polar Coordinates

We can create polygons in polar coordinates using a `Range` and `map` as shown below.

```
import doodle.core.Point._

def polygon(sides: Int, size: Int): Image = {
  val rotation = Angle.one / sides
  val elts =
    (1 to sides).toList.map { i =>
      PathElement.lineTo(polar(size, rotation * i))
    }
  Image.closedPath(PathElement.moveTo(polar(size, Angle.zero)) :: elts)
}
```

We cannot so easily write the same method to generate turtle instructions using a `Range` and `map`. Why is this? What abstraction are we missing?

See the solution

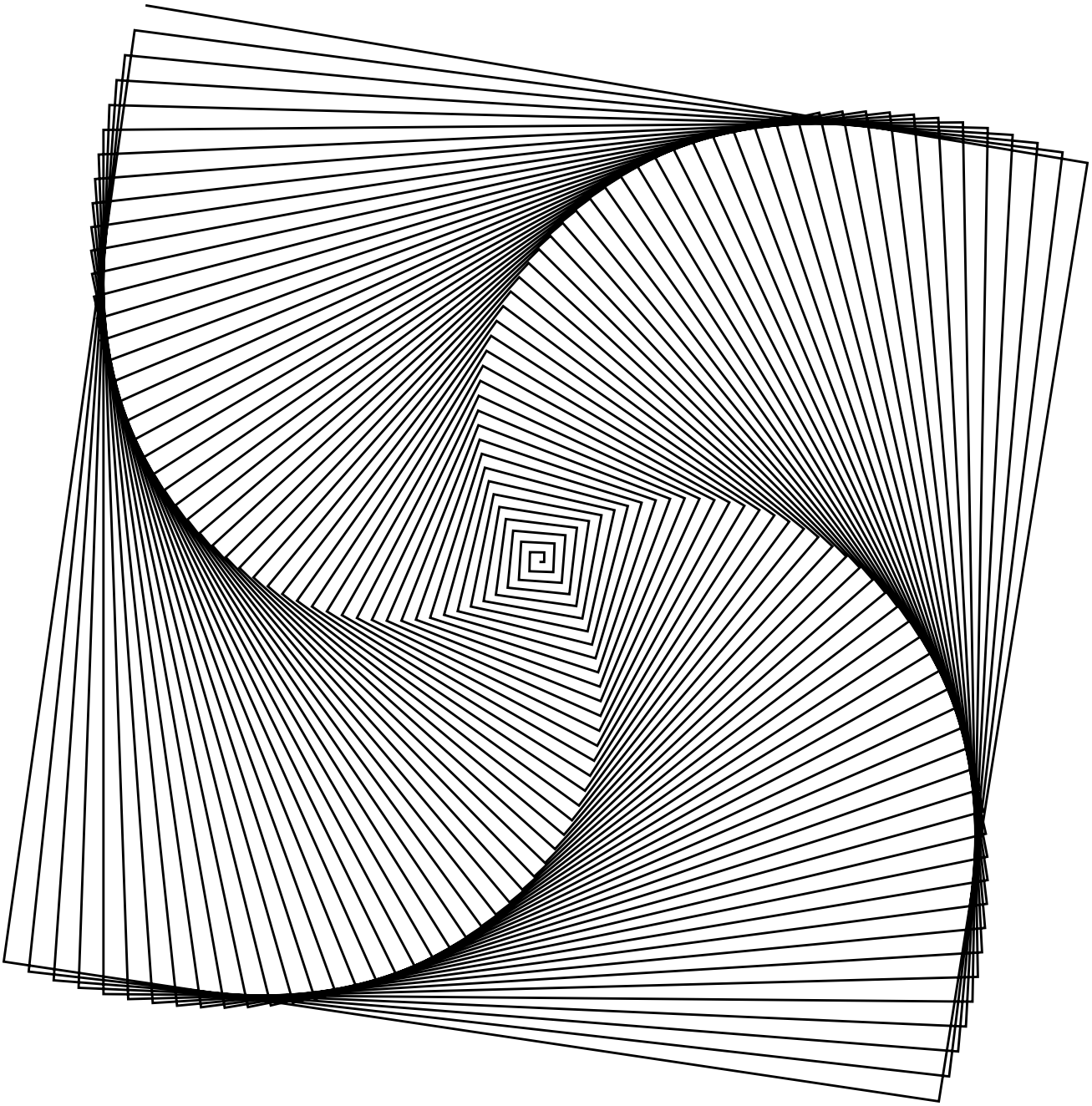

Figure 12.3: The square spiral!

12.3 Branching Structures

In addition to the standard imports given at the start of the chapter, in this section we're assuming the following:

```
import doodle.turtle._
import doodle.turtle.Instruction._
```

Using the `branch` turtle instruction we can explore some shapes that are difficult to create without it. The `branch` instruction takes a `List[Instruction]`. It saves the current state of the turtle (it's location and heading), draws the given instructions, and returns the turtle to the saved state.

Consider the code below, which creates the image in fig. 12.4. This is easy to draw with a branching turtle, but quite involved to create with just a path.

```
val y = Turtle.draw(List(
          forward(100),
          branch(turn(45.degrees), forward(100)),
          branch(turn(-45.degrees), forward(100)))
        )
// y: Image = OpenPath(
//   List(
//     MoveTo(Cartesian(0.0, 0.0)),
//     MoveTo(Cartesian(0.0, 0.0)),
//     LineTo(Cartesian(100.0, 0.0)),
//     LineTo(Cartesian(170.71067811865476, 70.71067811865474)),
//     MoveTo(Cartesian(100.0, 0.0)),
//     LineTo(Cartesian(170.71067811865476, -70.71067811865474)),
//     MoveTo(Cartesian(100.0, 0.0))
//   )
// )
```

Using branching we can model some forms of biological growth, producing, for example, images of plants as in fig. 12.1. One particular model is known as an *L-system*. An L-system has consists of two parts:

- an initial *seed* to start the growth; and
- *rewrite rules*, which specify how the growth occurs.

A specific example of this process is shown in fig. 12.5. The figure on the left hand side is the seed. The rewrite rules are:

- each straight line doubles in size; and
- a bud (the diamond at the end of a line) grows into two branches that end with buds.

Concretely, we can write these rules as a transformation on `Instruction` assuming that we use `NoOp` to represent a bud.

```
val stepSize = 10
// stepSize: Int = 10

def rule(i: Instruction): List[Instruction] =
  i match {
    case Forward(_) => List(forward(stepSize), forward(stepSize))
    case NoOp =>
      List(branch(turn(45.degrees), forward(stepSize), noop),
           branch(turn(-45.degrees), forward(stepSize), noop))
```

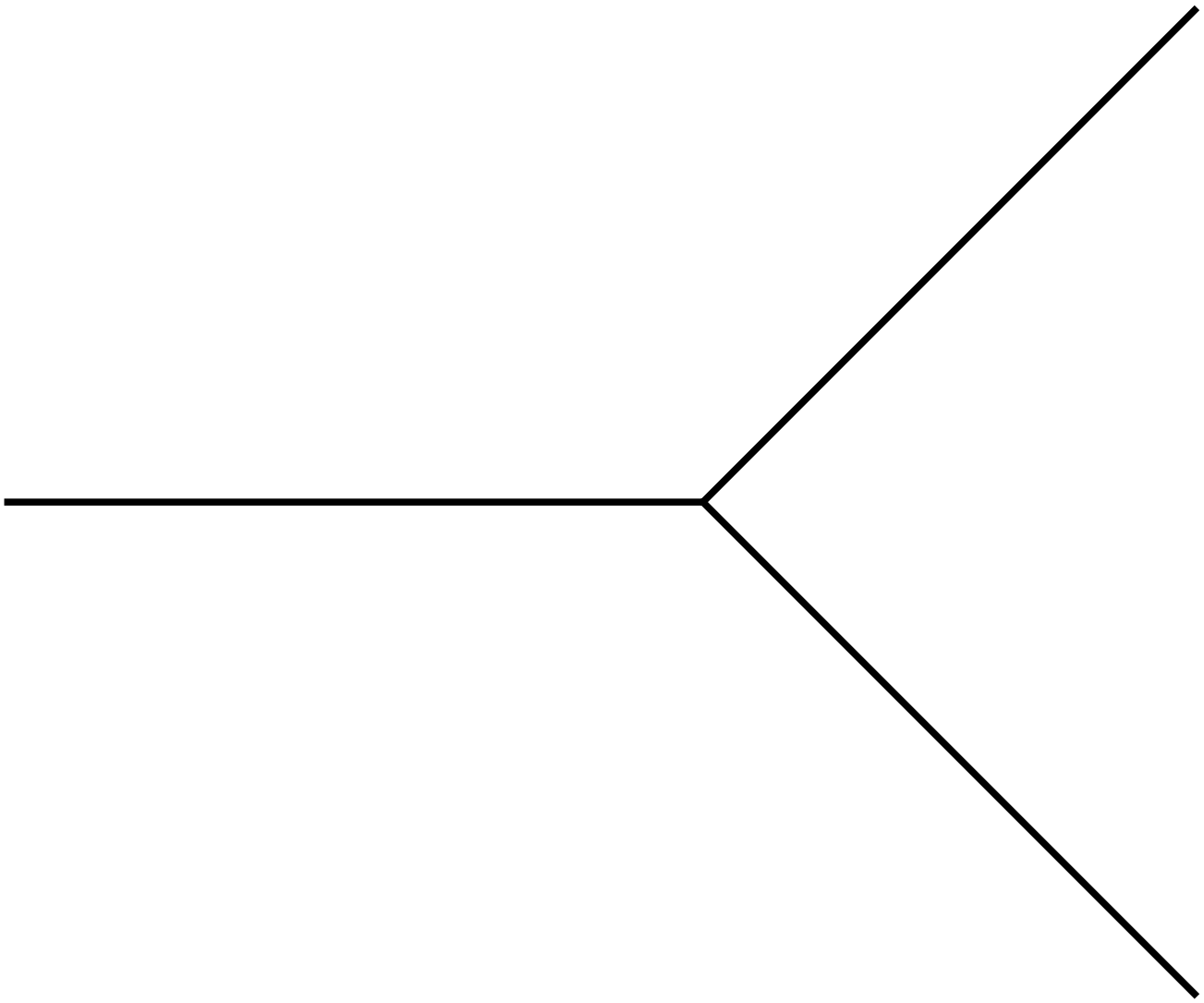

Figure 12.4: An image that is easy to create with a branching turtle, and comparatively difficult to create without.

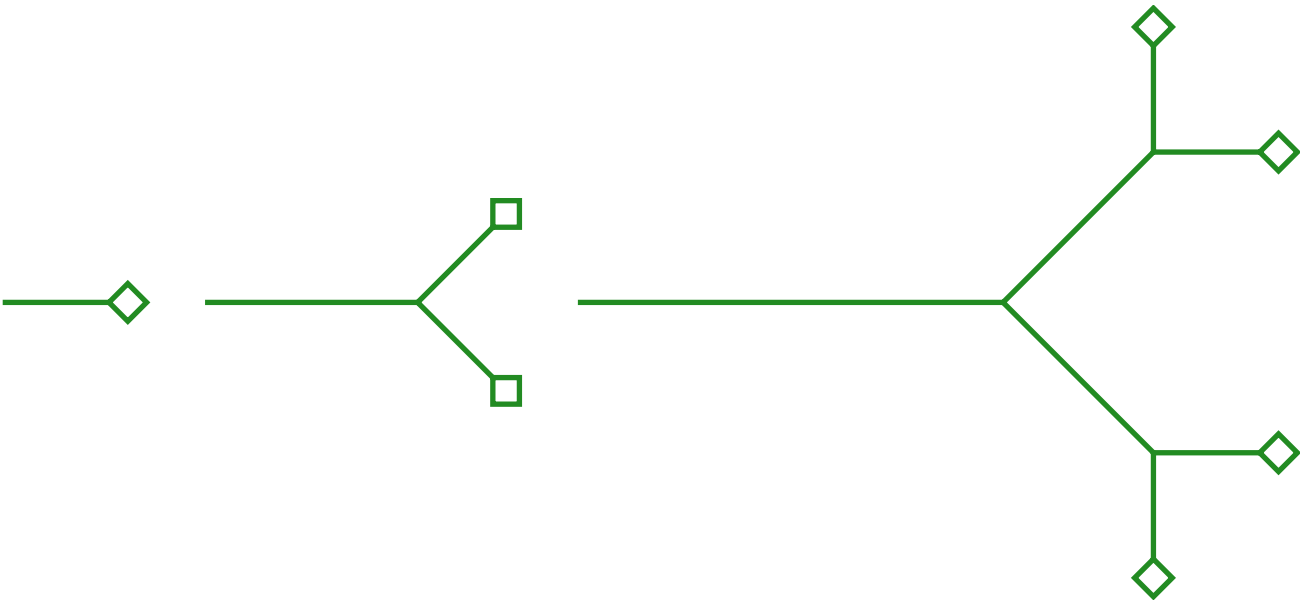

Figure 12.5: Modelling the growth of a plant using rewrite rules.

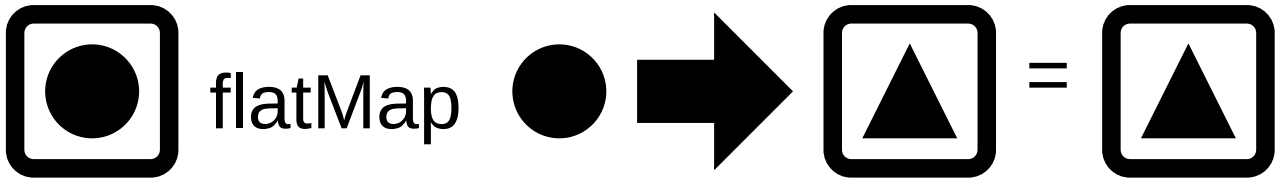

Figure 12.6: The type equation for flatMap illustrated graphically.

```
    case other => List(other)
  }
```

Note how we used pattern matching on `Instruction`, like we have on the other algebraic data types—natural numbers and `List`—we've seen so far. By importing `doodle.turtle.Instruction._` we can access all the patterns for `Instruction`, which are

- `Forward(distance)`, where `distance` is a `Double`;
- `Turn(angle)`, where `angle` is an `Angle`;
- `NoOp`; and
- `Branch(instructions)`, where `instructions` is a `List[Instruction]`.

As a function, `rule` has type `Instruction => List[Instruction]`, as we're potentially transforming each instruction into several instructions (as we do in the case of `Forward`). Now how can we actually apply this rule to a `List[Instruction]` to create a `List[Instruction]` (for example, applying it to `List[noop]`)? Can we use `map`?

See the solution

There is a method `flatten` on `List`, which will convert a `List[List[A]]` to `List[A]`. We *could* use a combination of `map` and `flatten` but we have a better solution. This pattern comes up enough—and in different contexts which we'll see later—that there is a method just to handle it. The method is called `flatMap`.

The type equation for `flatMap` is

```
List[A] flatMap (A => List[B]) = List[B]
```

and this is illustrated graphically in fig. 12.6. We can see that `flatMap` has the right type to combine `rule` with `List[Instruction]` to create a rewritten `List[Instruction]`.

When discussing `map` we said that it doesn't allow us to change the number of elements in the `List`. Graphically, we can't create a new "box" using `map`. With `flatMap` we can change the box, in the case lists meaning we can change the number of elements in the list.

Exercises

Double

Using `flatMap`, write a method `double` that transforms a `List` to a `List` where every element appears twice. For example

```
double(List(1, 2, 3))
// res0: List[Int] = List(1, 1, 2, 2, 3, 3)
double(List("do", "ray", "me"))
// res1: List[String] = List("do", "do", "ray", "ray", "me", "me")
```

See the solution

Or Nothing

Using `flatMap`, write a method `nothing` that transforms a `List` to the empty `List`. For example

```
nothing(List(1, 2, 3))
// res3: List[Int] = List()
nothing(List("do", "ray", "me"))
// res4: List[String] = List()
```

See the solution

Rewriting the Rules

Write a method `rewrite` with signature

```
def rewrite(instructions: List[Instruction],
            rule: Instruction => List[Instruction]): List[Instruction] =
  ???
```

This method should apply `rule` to rewrite every instruction in `instructions`, except for branches which you'll need to handle specially. If you encounter a branch you should rewrite all the instructions inside the branch but leave the branch alone.

Note: You'll need to pass a `List[Instruction]` to `branch`, while `branch` itself accepts zero or more instructions (so-called *varargs*). To convert the `List[Instruction]` into a form that `branch` will accept, follow the parameter with `:_*` like so

```
val instructions = List(turn(45.degrees), forward(10))
// instructions: List[Instruction] = List(
//   Turn(Angle(0.7853981633974483)),
//   Forward(10.0)
// )
branch(instructions:_*)
// res8: Branch = Branch(List(Turn(Angle(0.7853981633974483)), Forward(10.0)))
```

See the solution

Your Own L-System

We're now ready to create a complete L-system. Using `rewrite` from above, create a method `iterate` with signature

```
def iterate(steps: Int,
            seed: List[Instruction],
            rule: Instruction => List[Instruction]): List[Instruction] =
  ???
```

This should recursively apply `rule` to `seed` for `steps` iterations.

See the solution

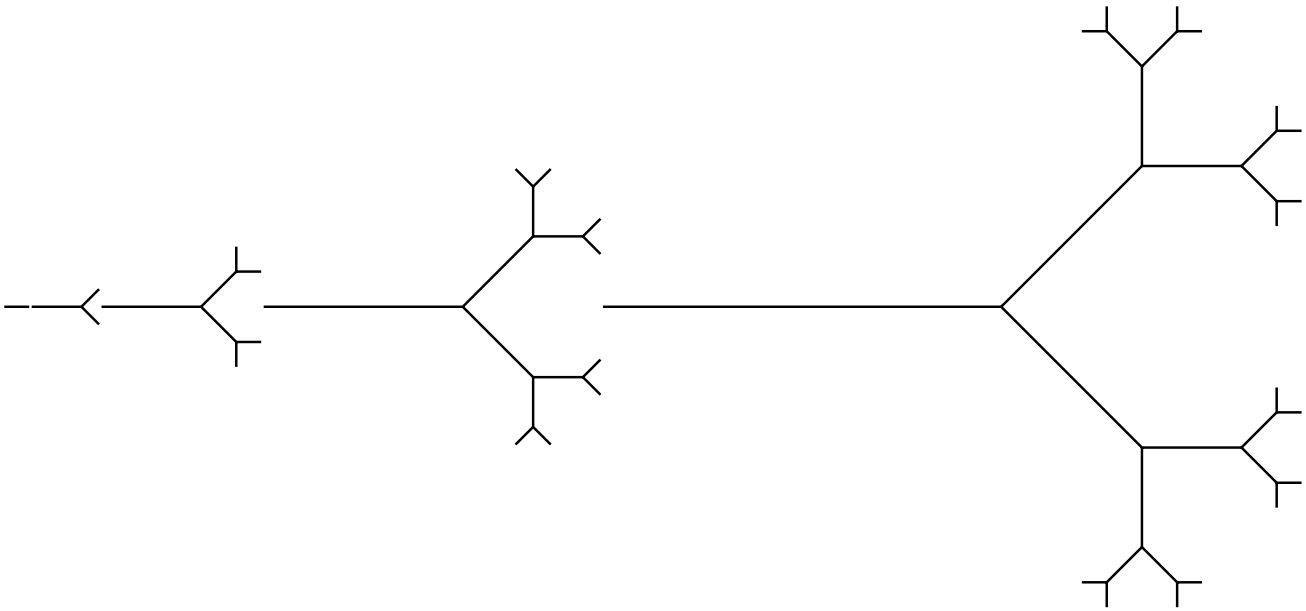

Figure 12.7: Five iterations of the simple branching L-system.

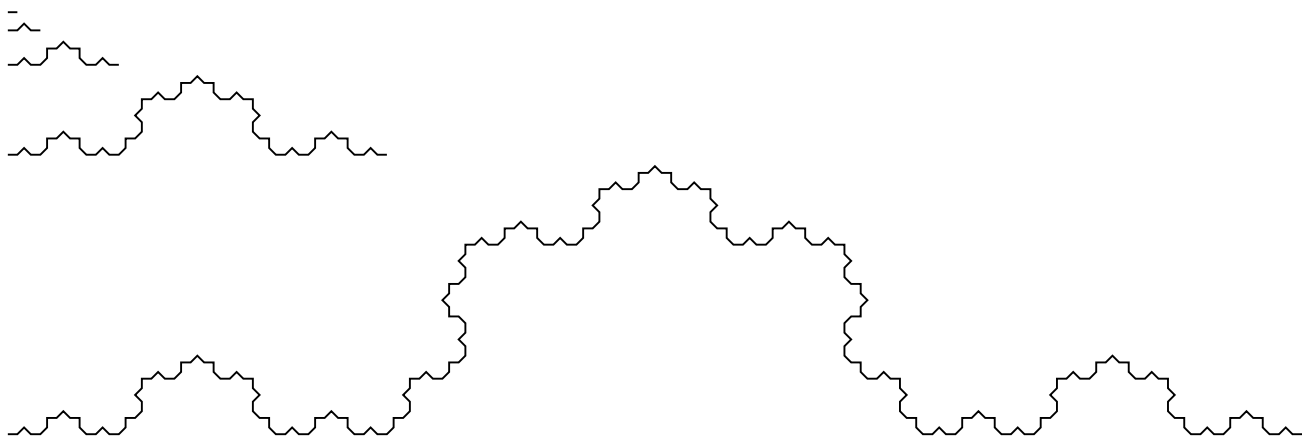

Figure 12.8: Five iterations of the Koch curve, a fractal that is simple to create with an L-System.

Plants and Other Creations

Create the pictures shown in fig. 12.7 and fig. **??** using your L-system implementation.

12.4 Exercises

12.4.1 Flat Polygon

Using the Turtle methods, `Range`, and `flatMap`, rewrite your method to create a polygon. The signature of `polygon` is

```
def polygon(sides: Int, sideLength: Double): Image =
  ???
```

See the solution

12.4.2 Flat Spiral

Using the Turtle methods, Range, and `flatMap`, rewrite your method to create the square spiral. The signature of `squareSpiral` is

```
def squareSpiral(steps: Int, distance: Double, angle: Angle, increment: Double): Image =
  ???
```

See the solution

12.4.3 L-System Art

In this exercise we want you to use your creativity to construct a picture of a natural object using your L-system implementation. You've seen many examples already that you can use an inspriation.

Chapter 13

Composition of Generative Art

In this chapter we'll explore techniques from generative art, which will in turn allow us to explore key concepts for functional programming. We'll see:

- uses for `map` and `flatMap` that go beyond manipulating collections of data that we've seen in the previous chapters;
- how we can handle side effects without breaking substitution; and
- some interesting, and possibly beautiful, images that combine elements of structure and randomness.

If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.

```
import doodle.core._
import doodle.image._
import doodle.image.syntax._
import doodle.image.syntax.core._
import doodle.java2d._
```

13.1 Generative Art

Generative art means art where some part of the composition is determined by an autonomous process. Concretely, for us this will mean adding an element of randomness.

Let's take a really simple example. We've learned previously how to create concentric circles.

```
def concentricCircles(n: Int): Image =
  n match {
    case 0 => Image.circle(10)
    case n => concentricCircles(n-1).on(Image.circle(n * 10))
  }
```

(We now know we could write this using a `Range` and a method like `allOn`.)

We also learned how we could make coloured circles, using a second parameter.

```
def concentricCircles(n: Int, color: Color): Image =
  n match {
    case 0 => Image.circle(10).fillColor(color)
    case n => concentricCircles(n-1, color.spin(15.degrees)).on(Image.circle(n * 10).fillColor(color
```

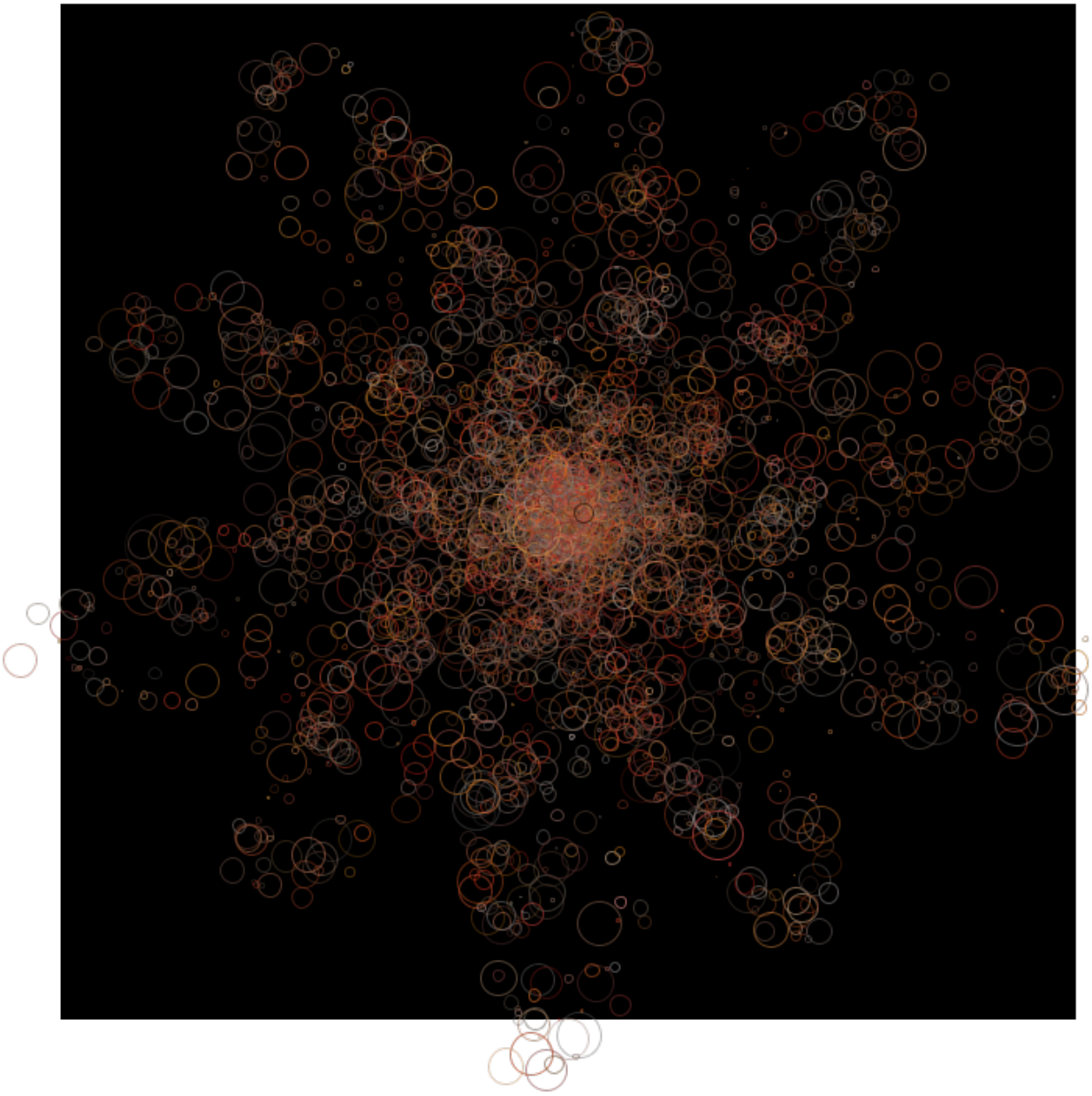

Figure 13.1: An example image generated using the techniques in this chapter

```
    ))
  }
```

Pictures constructed in this way are nice, but they are a bit boring in their regularity. What if we wanted to make a random alteration to the hue of the color at each step?

Scala provides some methods that produce *random numbers*. One such method is `math.random`. Each time we call it we get a different `Double` between 0.0 and 1.0[1].

```
math.random
// res0: Double = 0.3841409073635985
math.random
// res1: Double = 0.858657941107252
```

Given `math.random` we could produce a method that returns a random `Angle` like so.

```
def randomAngle: Angle =
  math.random.turns

randomAngle
// res2: Angle = Angle(1.0598850824318808)
randomAngle
// res3: Angle = Angle(5.741077450831025)
```

Why might we not want to do this? What principle does this break?

See the solution

What should we do? Suffer the slings and arrows of outrageous computational models, or take arms against a sea of side-effects, and by opposing end them! There's really only one choice.

13.2 Randomness without Effect

The solution to our problem is to separate describing how we'll use random numbers from the process of actually generating them. This sounds complicated, but it's exactly what we've be doing with `Image` throughout this book. We

- describe an `Image` using paths and methods like `beside`, `above`, and `on`; and
- we only draw an `Image` when we call `draw`.

We do the same thing with Doodle's `Random` type. To access this code we first need to import the `doodle.random` package.

```
import doodle.random._
```

Now we can create values that describe creating a random number

[1]These numbers are not truly random. The output is determined by a value known as the *seed*. If we know the seed we can perfectly predict all the result we'll get from calling `math.random`. However, going the other way—that is, predicting the seed given a sequence of outputs—is very difficult. The numbers so generated are called *pseudo-random numbers*, because they are not truly random but nonetheless are very difficult to predict.

```
val randomDouble = Random.double
// randomDouble: Random[Double] = Suspend(RDouble)
```

No random numbers are actually created until we call the `run` method.

```
randomDouble.run
// res0: Double = 0.6293890040202628
```

The type `Random[Double]` indicates we have something that will produce a random `Double` when we `run` it. Just like with `Image` and `draw`, substitution holds with `Random` up until the point we call `run`.

Table tbl. 13.1 shows some of the ways to construct `Random` values.

Table 13.1: Some of the methods to create `Random` values.

Method	Description	Example
`Random.always(value)`	Creates a `Random` that always produces the given value.	`Random.always(10)`
`Random.double`	Creates a `Random` that generates a `Double` uniformly distributed between `0.0` and `1.0`.	`Random.double`
`Random.int`	Creates a `Random` that generates an `Int` uniformly distributed across the entire range.	`Random.int`
`Random.natural(limit)`	Creates a `Random` that generates a `Int` uniformly distributed in the range greater than or equal to `0` and less than `1`.	`Random.natural(10)`
`Random.oneOf(value, ...)`	Creates a `Random` that generates one of the given values with equal chance.	`Random.oneOf("A", "B", "C")`

13.2.1 Composing Random

Now we've seen how to create a `Random`, how do we compose them into more interesting programs? For example, how could we turn a random `Double` into a random `Angle`? It might be tempting to call `run` every time we want to manipulate a random result, but this will break substitution and is exactly what we're trying to avoid.

Remember when we talked about `map` in the previous chapter we said it transforms the elements but keeps the structure (number of elements) in the `List`. The same analogy applies to the `map` method on `Random`. It lets us transform the element of a `Random`—the value it produces when it is run—but doesn't let us change the structure. Here the "structure" means introducing more randomness, or making a random choice.

We can create a random value and apply a *deterministic* transformation to it using `map`, but we can't create a random value and them use that value as input to a process that creates another random value.

Here's how we can create a random angle.

```
val randomAngle: Random[Angle] =
  Random.double.map(x => x.turns)
```

When we `run` `RandomAngle` we can generate randomly created `Angle`

```
randomAngle.run
// res1: Angle = Angle(4.201103170459831)
randomAngle.run
// res2: Angle = Angle(1.1433721813030546)
```

Exercises

Random Colors

Given `randomAngle` above, create a method that accepts saturation and lightness and generates a random color. Your method should have the signature

```
def randomColor(s: Normalized, l: Normalized): Random[Color] =
  ???
```

This is a deterministic transformation of the output of `randomAngle`, so we can implement it using `map`.

```
def randomColor(s: Double, l: Double): Random[Color] =
  randomAngle.map(hue => Color.hsl(hue, s, l))
```

Random Circles

Write a method that accepts a radius and a `Random[Color]`, and produces a circle of the given radius and filled with the given random color. It should have the signature

```
def randomCircle(r: Double, color: Random[Color]): Random[Image] =
  ???
```

Once again this is a deterministic transformation of the random color, so we can use `map`.

```
def randomCircle(r: Double, color: Random[Color]): Random[Image] =
  color.map(fill => Image.circle(r).fillColor(fill))
```

13.3 Combining Random Values

> In addition to the standard imports given at the start of the chapter, in this section we're assuming the following:
>
> ```
> import doodle.random._
> ```

So far we've seen how to represent functions generating random values using the `Random` type, and how to make deterministic transformations of a random value using `map`. In this section we'll see how we can make a random (or stochastic, if you prefer fancier words) transformation of a random value using `flatMap`.

To motivate the problem lets try writing `randomConcentricCircles`, which will generate concentric circles with randomly chosen hue using the utility methods we developed in the previous section.

We start with the code to create concentric circles with deterministic colors and the utilities we developed previously.

```
def concentricCircles(count: Int, size: Int, color: Color): Image =
  count match {
    case 0 => Image.empty
    case n =>
      Image.circle(size).fillColor(color).on(concentricCircles(n-1, size + 5, color.spin(15.degrees)
    ))
  }

val randomAngle: Random[Angle] =
  Random.double.map(x => x.turns)

def randomColor(s: Double, l: Double): Random[Color] =
  randomAngle map (hue => Color.hsl(hue, s, l))

def randomCircle(r: Double, color: Random[Color]): Random[Image] =
  color.map(fill => Image.circle(r).fillColor(fill))
```

Let's create a method skeleton for `randomConcentricCircles`.

```
def randomConcentricCircles(count: Int, size: Int): Random[Image] =
  ???
```

The important change here is we return a `Random[Image]` not an `Image`. We know this is a structural recursion over the natural numbers so we can fill out the body a bit.

```
def randomConcentricCircles(count: Int, size: Int): Random[Image] =
  count match {
    case 0 => ???
    case n => ???
  }
```

The base case will be `Random.always(Image.empty)`, the direct of equivalent of `Image.empty` in the deterministic case.

```
def randomConcentricCircles(count: Int, size: Int): Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n => ???
  }
```

What about the recursive case? We could try using

```
val randomPastel = randomColor(0.7, 0.7)

def randomConcentricCircles(count: Int, size: Int): Random[Image] =
  count match {
    case 0 => Image.empty
    case n =>
      randomCircle(size, randomPastel).on(randomConcentricCircles(n-1, size + 5))
  }
// error: type mismatch;
//  found   : doodle.image.Image
//  required: doodle.random.Random[doodle.image.Image]
//     (which expands to)  cats.free.Free[doodle.random.RandomOp,doodle.image.Image]
//     case 0 => Random.always(Image.empty)
//                             ^^^^^^^^^^^
// error: value on is not a member of doodle.random.Random[doodle.image.Image]
```

```
//       randomCircle(size, randomPastel).flatMap{ circle =>
//       ^
```

but this does not compile. Both `randomConcentricCircles` and `randomCircle` evaluate to `Random[Image]`. There is no method on on `Random[Image]` so this code doesn't work.

Since this is a transformation of two `Random[Image]` values, it seems like we need some kind of method that allows us to transform *two* `Random[Image]`, not just one like we can do with `map`.
We might call this method `map2` and we could imagine writing code like

```
randomCircle(size, randomPastel).map2(randomConcentricCircles(n-1, size + 5)){
  (circle, circles) => circle on circles
}
```

Presumably we'd also need `map3`, `map4`, and so on. Instead of these special cases we can use `flatMap` and `map` together.

```
randomCircle(size, randomPastel) flatMap { circle =>
  randomConcentricCircles(n-1, size + 5) map { circles =>
    circle on circles
  }
}
```

The complete code becomes

```
def randomConcentricCircles(count: Int, size: Int): Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n =>
      randomCircle(size, randomPastel).flatMap{ circle =>
        randomConcentricCircles(n-1, size + 5).map{ circles =>
          circle.on(circles)
        }
      }
  }
```

Example output is shown in fig. 13.2.

Let's now look closer at this use of `flatMap` and `map` to understand how this works.

13.3.1 Type Algebra

The simplest way, in my opinion, to understand why this code works is to look at the types. The code in question is

```
randomCircle(size, randomPastel) flatMap { circle =>
  randomConcentricCircles(n-1, size + 5) map { circles =>
    circle on circles
  }
}
```

Starting from the inside, we have

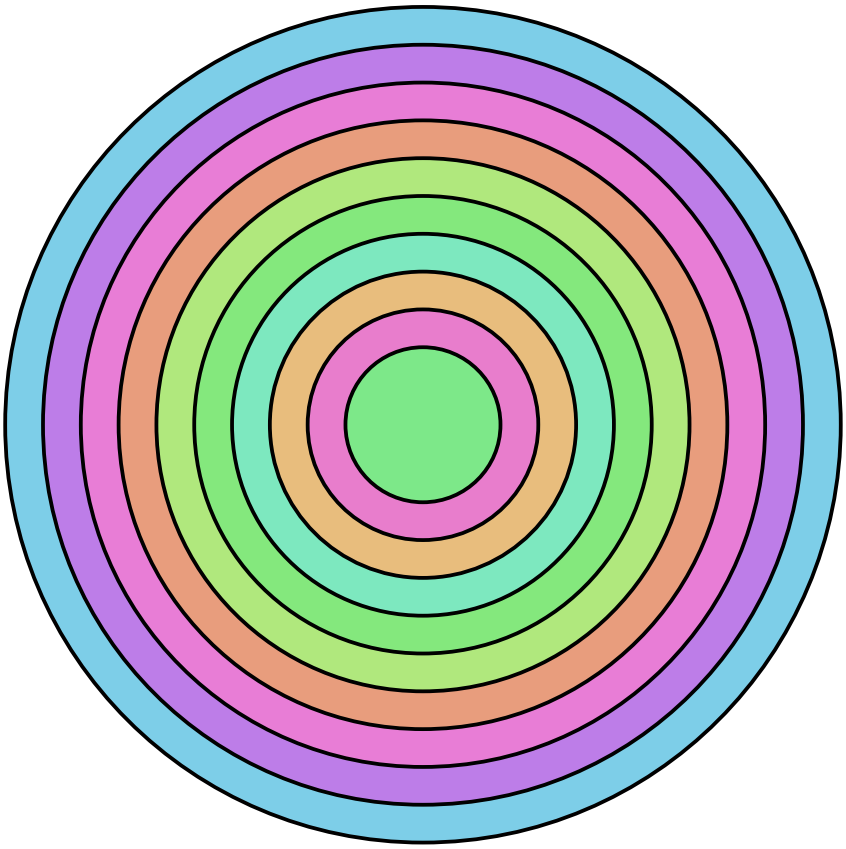

Figure 13.2: The output of one run of `randomConcentricCircles(10, 10).run.draw`

```
{ circles =>
    circle on circles
}
```

which is a function with type

```
Image => Image
```

Wrapping this we have

```
randomConcentricCircles(n-1, size + 5) map { circles =>
    circle on circles
  }
```

We known `randomConcentricCircles(n-1, size + 5)` has type `Random[Image]`. Substituting in the `Image => Image` type we worked out above we get

```
Random[Image] map (Image => Image)
```

Now we can deal with the entire expression

```
randomCircle(size, randomPastel) flatMap { circle =>
  randomConcentricCircles(n-1, size + 5) map { circles =>
    circle on circles
  }
}
```

`randomCircle(size, randomPastel)` has type `Random[Image]`. Performing substitution again gets us a type equation for the entire expression.

```
Random[Inage] flatMap (Random[Image] map (Image => Image))
```

Now we can apply the type equations for `map` and `flatMap` that we saw earlier:

```
F[A] map (A => B) = F[B]
F[A] flatMap (A => F[B]) = F[B]
```

Working again from the inside out, we first use the type equation for `map` which simplifies the type expression to

```
Random[Inage] flatMap (Random[Image])
```

Now we can apply the equation for `flatMap` yielding just

```
Random[Image]
```

This tells us the result has the type we want. Notice that we've been performing substitution at the type level—the same technique we usually use at the value level.

Exercises

Don't forget to import `doodle.random._` when you attempt these exercises.

Randomness and Randomness

What is the difference between the output of `programOne` and `programTwo` below? Why do they differ?

```
def randomCircle(r: Double, color: Random[Color]): Random[Image] =
  color.map(fill => Image.circle(r).fillColor(fill))

def randomConcentricCircles(count: Int, size: Int): Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n =>
      randomCircle(size, randomPastel).flatMap{ circle =>
        randomConcentricCircles(n-1, size + 5).map{ circles =>
          circle.on(circles)
        }
      }
  }

val circles = randomConcentricCircles(5, 10)
val programOne =
  circles.flatMap{ c1 =>
    circles.flatMap{ c2 =>
      circles.map{ c3 =>
        c1.beside(c2).beside(c3)
      }
    }
  }

val programTwo =
  circles map { c => c beside c beside c }
```

See the solution

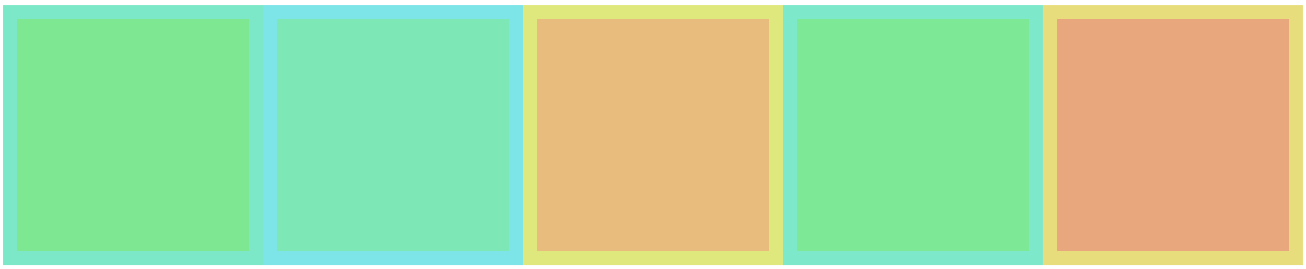

Figure 13.3: Boxes filled with random colors.

Colored Boxes

Let's return to a problem from the beginning of the book: drawing colored boxes. This time we're going to make the gradient a little more interesting, by making each color randomly chosen.

Recall the basic structural recursion for making a row of boxes

```
def rowOfBoxes(count: Int): Image =
  count match {
    case 0 => Image.rectangle(20, 20)
    case n => Image.rectangle(20, 20).beside(rowOfBoxes(n-1))
  }
```

Let's alter this, like with did with concentric circles, to have each box filled with a random color. *Hint:* you might find it useful to reuse some of the utilities we created for `randomConcentricCircles`. Example output is shown in fig. 13.3.

See the solution

13.4 Exploring Random

So far we've seen only the very basics of using `Random`. In this section we'll see more of its features, and use these features to create more interesting pictures.

> In addition to the standard imports given at the start of the chapter, in this section we're assuming the following:
>
> ```
> import doodle.random._
> ```

13.4.1 Normal Distributions

Often when using random numbers in generative art we will choose specific distributions for the shape they provide. For example, fig. 13.4 shows a thousand random points generated using a uniform, normal (or Gaussian) distribution, and a squared normal distribution respectively.

As you can see, the normal distribution tends to generate more points nearer the center than the uniform distribution.

Doodle provides two methods to create normally distributed numbers, from which we can create many other distributions. A normal distribution is defined by two parameters, it's *mean*, which specifies the center of the distribution, and it's *standard deviation*, which determines the spread of the distribution. The corresponding methods in Doodle are

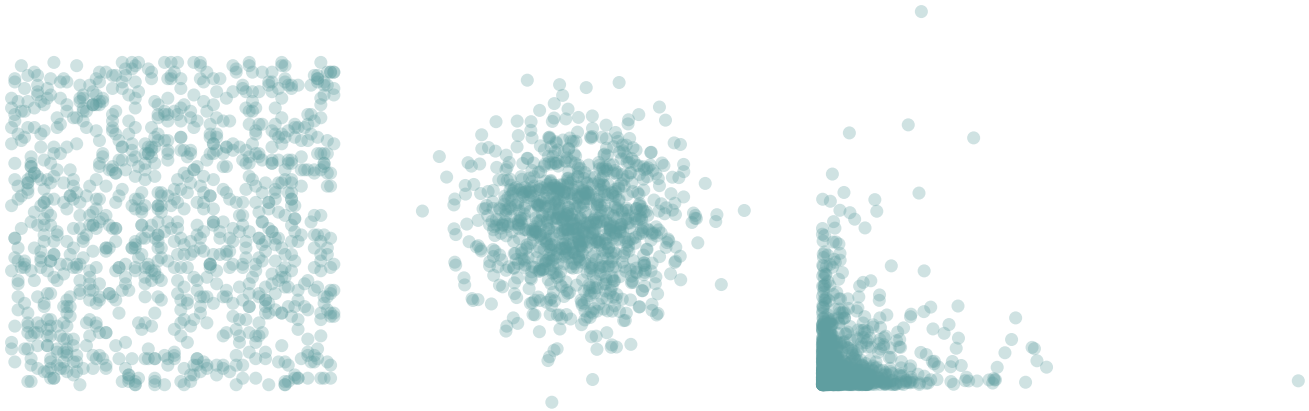

Figure 13.4: Points distributed according to uniform, normal, and squared normal distributions

- `Random.normal`, which generates a `Double` from a normal distribution with mean 0 and standard deviation 1.0; and
- `Random.normal(mean, stdDev)`, which generates a `Double` from a normal distribution with the specified mean and standard deviation.

13.4.2 Structured Randomness

We've gone from very structured to very random pictures. It would be nice to find a middle ground that incorporates elements of randomness and structure. We can use `flatMap` to do this—with `flatMap` we can use one randomly generated value to create another `Random` value. This creates a dependency between values—the prior random value has an influence on the next one we generate.

For example, we can create a method that given a color randomly perturbs it.

```
def nextColor(color: Color): Random[Color] = {
  val spin = Random.normal(15.0, 10.0)
  spin.map{ s => color.spin(s.degrees) }
}
```

Using `nextColor` we can create a series of boxes with a gradient that is partly random and partly structured: the next color in the gradient is a random perturbation of the previous one.

```
def coloredRectangle(color: Color, size: Int): Image =
  Image.rectangle(size, size)
       .strokeWidth(5.0)
       .strokeColor(color.spin(30.degrees))
       .fillColor(color)

def randomGradientBoxes(count: Int, color: Color): Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n =>
      val box = coloredRectangle(color, 20)
      val boxes = nextColor(color).flatMap{ c => randomGradientBoxes(n-1, c) }
      boxes.map{ b => box beside b }
  }
```

Example output is shown in fig. 13.5.

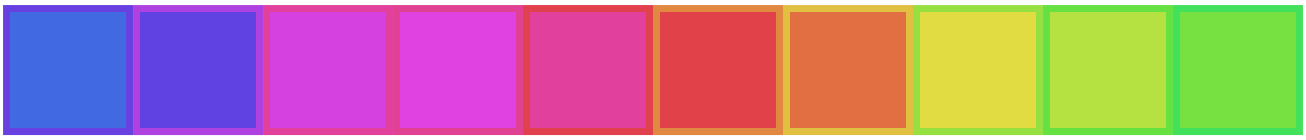

Figure 13.5: Boxes filled with gradient that is partly random.

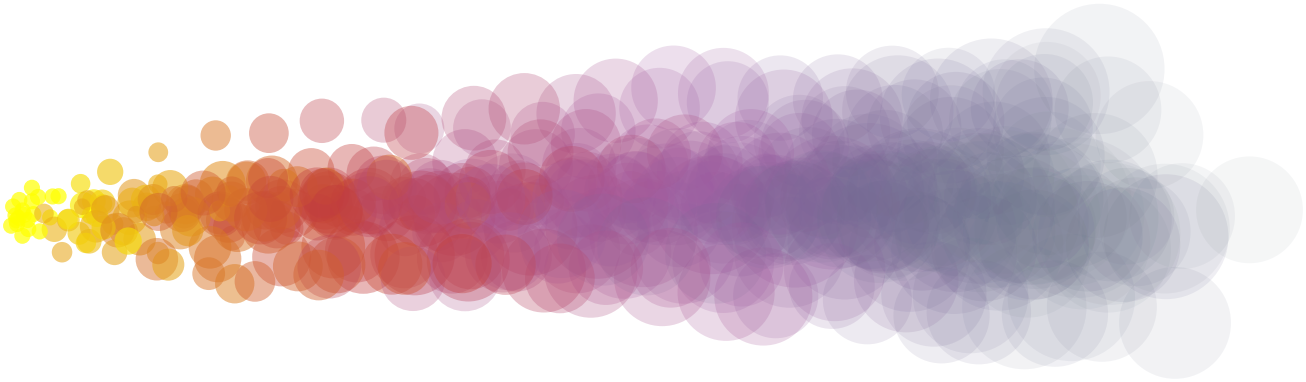

Figure 13.6: A simulation of a smoky fire, generating using a particle system.

Exercises

Particle Systems

A *particle system* is a technique used in computer graphics to create large numbers of "particles" that move according to simple rules. In fig. 13.6 there is an example of a particle system simulating a fire and smoke. For the mathematically inclined, a particle system is basically a *stochastic process* or *random walk*.

In this exercise we'll build a particle system, which will give you a flexible system to experiment with these ideas. We'll start with a fixed system and then abstract it to create reusable components.

Here's a sketch of how a particle system works. To draw a single particle we

- choose a starting position;
- decide how many time steps we want to move the particle system for; and
- at each timestep the new position of the particle is equal to the position at the previous timestep plus some random noise (and potentially some non-random (deterministic) movement such as velocity or acceleration).

A particle system is just a collection of a number of particles—20 particles over 20 steps in fig. 13.6.

In the above description we've broken down the components of a partcile system. Now we just need to implement them.

The starting position can be any `Random[Point]`. Create one now.

See the solution

Let's implement a method `step` that will take a single step in particle system. It will have skeleton

```
def step(current: Point): Random[Point] =
  ???
```

We need to decide how we will modify the `current` point to create the next point. I suggest adding some random noise, and a constant "drift" that will ensure the points tend to move in a particular direction. For

example, we could increment the x coordinate by 10, which will cause a drift towards the right of the screen, plus some normally distributed noise to the x and y coordinates.

See the solution

Now that we can `step` a particle we need to connect a sequence of steps to get a `walk`. There is one wrinkle here: we want to draw the intermediate stages so we're going to define two methods:

- a method that transforms a `Point` to an `Image`; and
- a method `walk` that produces a `Random[Image]`

The skeletons are

```
def render(point: Point): Image =
  ???

def walk(steps: Int): Random[Image] =
  ???
```

The implementation of `render` can be whatever you fancy. In the implementation of `walk`, you will have to call `step` to get the next `Point`, and then call `render` to convert the point to something that can be draw. You will also want to have an accumulator of the `Image` so far. Hint: you might find it useful to define an auxiliary parameter for `walk`.

See the solution

Now you should be able to call `walk` and render the result.

The final step is create a number of particles and render them all. Create a method `particleSystem` with skeleton

```
def particleSystem(particles: Int, steps: Int): Random[Image] =
  ???
```

that does just this.

See the solution

Now render the result, and tweak it till you have something you're happy with. I'm not particulary happy with the result of my code. I think the stars are too bunched up, and the colors are not very interesting. To make a more interesting result I'd consider adding more noise and changing the start color and perhaps compressing the range of colors.

Random Abstractions

The implementation of `particleSystem` above hard-codes in a particular choice of particle system. To make it easier to experiment with we might like to abstract over the particular choice of `walk` and `start`. How do you think we could do this?

See the solution

Implement this.

See the solution

This code doesn't make me happy. Most of the parameters to `particleSystem` are only needed to pass on to `walk`. These parameters don't change is any way within the structural recursion that makes up the body of `particleSystem`. At this point we can apply our principle of substitution—we can replace a method call with the value it evaluates to—to remove `walk` and associated parameters from `particleSystem`.

```
def particleSystem(particles: Int, walk: Random[Image]): Random[Image] = {
  particles match {
    case 0 => Random.always(Image.empty)
    case n => walk.flatMap{ img1 =>
      particleSystem(n-1, walk) map { img2 =>
        img1.on(img2)
      }
    }
  }
}
```

If you're used to programming in imperative languages this may seem mind-bending. Remember that we've gone to some lengths to ensure that working with random numbers obeys substitution, up to the point that `run` is called. The `walk` method doesn't actually create a random walk. It instead describes how to create a random walk when that code is actually run. This separation between description and action means that substitution can be used. The description of how to perform a random walk can be used to create many different random walks.

13.5 For Comprehensions

> In addition to the standard imports given at the start of the chapter, in this section we're assuming the following:
>
> ```
> import doodle.random._
> ```

Scala provides some special syntax, called a *for comprehension*, that makes it simpler to write long sequences of `flatMap` and `map`.

For example, the code for `randomConcentricCircles` has a call to `flatMap` and `map`.

```
def randomConcentricCircles(count: Int, size: Int): Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n =>
      randomCircle(size, randomPastel).flatMap{ circle =>
        randomConcentricCircles(n-1, size + 5).map{ circles =>
          circle.on(circles)
        }
      }
  }
```

This can be replaced with a for comprehension.

```
def randomConcentricCircles(count: Int, size: Int): Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n =>
      for {
        circle  <- randomCircle(size, randomPastel)
        circles <- randomConcentricCircles(n-1, size + 5)
      } yield circle.on(circles)
  }
```

The for comprehension is often easier to read than direct use of `flatMap` and `map`.

A general for comprehension

```
for {
  x <- a
  y <- b
  z <- c
} yield e
```

translates to:

```
a.flatMap(x => b.flatMap(y => c.map(z => e)))
```

Which is to say that every `<-`, except the last, turns into a `flatMap`, and the last `<-` becomes a `map`.

For comprehensions are translated by the compiler into uses of `flatMap` and `map`. There is no magic going on. It is just a different way of writing code that would use `flatMap` and `map` that avoids excessive nesting.

Note that the for comprehension syntax is more flexible than what we have presented here. For example, you can drop the `yield` keyword from a for comprehension and the code will still compile. It just won't return a result. We're not going to use any of these extensions in Creative Scala, however.

13.6 Exercises

13.6.1 Scatter Plots

In this exercise we'll implement scatter plots as in fig. 13.4. Experiment with different distributions (trying creating your own distributions by transforming ones defined on `Random`).

There are three main components of a scatter plot:

- we need to generate the points we'll plot;
- we need to overlay the images on top of each other in the same coordinate system to create the plot; and
- we need to convert a point to an image we can render.

We tackle each task in turn.

Start by writing a method `makePoint` that will accept a `Random[Double]` for the x and y coordinates of a point and return a `Random[Point]`. It should have the following skeleton:

```
def makePoint(x: Random[Double], y: Random[Double]): Random[Point] =
  ???
```

Use a for comprehension in your implementation.

See the solution

Now create, say, a thousand random points using the techniques we learned in the previous chapter on lists and a random distribution of your choice. You should end up with a `List[Random[Point]]`.

See the solution

Let's now transform our `List[Random[Point]]` into `List[Random[Image]]`. Do this in two steps: first write a method to convert a `Point` to an `Image`, then write code to convert `List[Random[Point]]` to `List[Random[Image]]`.

See the solution

Now create a method that transforms a `List[Random[Image]]` to a `Random[Image]` by placing all the points on each other. This is the equivalent of the `allOn` method we've developed previously, but it now works with data wrapped in `Random`.

See the solution

Now put it all together to make a scatter plot.

See the solution

13.6.2 Parametric Noise

In this exercise we will combine parametric equations, from a previous chapter, with randomness.

Let's start by making a method `perturb` that adds random noise to a `Point`. The method should have skeleton

```
def perturb(point: Point): Random[Point] =
  ???
```

Choose whatever noise function you like.

See the solution

Now create a parametric function, like we did in a previous chapter. You could use the rose function (the function we explored previously) or you could create one of your own devising. Here's the definition of rose.

```
def rose(k: Int): Angle => Point =
  (angle: Angle) => {
    Point.cartesian((angle * k).cos * angle.cos, (angle * k).cos * angle.sin)
  }
```

We can combine our parametric function and `perturb` to create a method with type `Angle => Random[Point]`. You can write this easily using the `andThen` method on functions, or you can write this out the long way. Here's a quick example of `andThen` showing how we write the fourth power in terms of the square.

```
val square = (x: Double) => x * x
val quartic = square andThen square
```

See the solution

Now using `allOn` create a picture that combines randomnes and structure. Be as creative as you like, perhaps adding color, transparency, and other features to your image.

See the solution

Chapter 14

Algebraic Data Types To Call Our Own

In this chapter we'll learn how to create our own algebraic data types, and bring together all the tools we've seen far.

So far in Creative Scala we've used (algebraic) data types provided by Scala or Doodle, such as `List` and `Point`. In this section we'll learn how to create our own algebraic data types in Scala, opening up new possibilities for the type of programs we can write.

> If you run the examples from the SBT console within Doodle they will just work. If not, you will need to start your code with the following imports to make Doodle available.
>
> ```
> import doodle.core._
> import doodle.image._
> import doodle.image.syntax._
> import doodle.image.syntax.core._
> import doodle.java2d._
> ```

14.1 Algebraic Data Types

We've used algebraic data types throughout Creative Scala, but we've been a bit informal in how we describe them. At this stage a bit more rigour is useful.

An algebraic data type is built from two components: - *logical ors*; and - *logical ands*.

The `List` data type is a great example of an algebraic data type, as it uses both patterns. A `List` is `Nil` *or* a pair (the logical or pattern) and a pair has a head *and* a tail (the logical and pattern). `Point` is another example. A `Point` is either `Cartesian` or `Polar`. A `Cartesian` has an x and y coordinate, while a `Polar` has a radius and an angle. Note it's not necessary to use both patterns to be an algebraic data type.

Being functional programmers we naturally have some fancy words for the logical or and logical and patterns. They are: - a *sum type* for the logical or; and - a *product type* for the logical and.

The concept of an algebraic data type is not specific to Scala. Let's get some practice working with the concept before we see how to write algebraic data types in Scala.

Exercises

14.1.0.0.1 Path Elements The `PathElement` type, used to construct paths, is a simple algebraic data type. You've used `PathElement` quite a bit so far. How do you think `PathElement` is defined in terms of sum and product types?

See the solution

14.1.0.0.2 Totally Turtles The `Instruction` type we used to control the turtle is also an algebraic data type. How do you think `Instruction` is defined?

See the solution

14.1.1 Defining Algebraic Data Types

No we understand how to model data with algebraic data types, let's see how to define our own.

The pattern is this:

- If `A` is a `B` or `C` write

```
sealed abstract class A extends Product with Serializable
final case class B() extends A
final case class C() extends A
```

There is a lot boilerplate here, which we can basically ignore beyond accepting it's stuff we have to write. However, if you're interested in what it means (and possibly have some prior object-oriented programming experience).

Describe `sealed` etc. here.

To define `PathElement` we might start with

```
sealed abstract class PathElement extends Product with Serializable
final case class MoveTo() extends PathElement
final case class LineTo() extends PathElement
final case class CurveTo() extends PathElement
```

The other half of the pattern is

- If `A` has a `B` and `C`, write

```
final case class A(b: B, c: C)
```

Describe constructor parameters here.

Returning to `PathElement`, `MoveTo` and `LineTo` each have a point (the destination) and `CurveTo` has a destination point and two control points. So we could write.

```
sealed abstract class PathElement extends Product with Serializable
final case class MoveTo(to: Point) extends PathElement
final case class LineTo(to: Point) extends PathElement
final case class CurveTo(cp1: Point, cp2: Point, to: Point) extends PathElement
```

And this is essentially how `PathElement` is defined in Doodle.

Exercise

Define your own algebraic data type to represent `Instruction`.

See the solution

14.2 Build Your Own Turtle

Here's the `Instruction` type we defined in the previous section.

```
sealed abstract class Instruction extends Product with Serializable
final case class Forward(distance: Double) extends Instruction
final case class Turn(angle: Angle) extends Instruction
final case class Branch(instructions: List[Instruction]) extends Instruction
final case class NoOp() extends Instruction
```

Now we've defined our own `Instruction` type, let's go one further and create our own `Turtle`. To complete our turtle we need to implement `draw`. We can start with

```
object Turtle {
  def draw(instructions: List[Instruction]): Image =
    ???
}
```

`Instruction` is an algebraic data type, so we know we can use structural recursion to process it. However to do so we need to also store the current state of the turtle: it's location (a `Vec`) and heading (an `Angle`). Implement a type to hold this data.

See the solution

When we process the instructions, we will turn them into a `List[PathElement]`, which we can later wrap with an open path to create an `Image`. For each instruction, the conversion will be a function of the current turtle state and the instruction, and will returnan updated state and a `List[PathElement]`.

Implement a method `process` to do this job with signature

```
def process(state: TurtleState, instruction: Instruction): (TurtleState, List[PathElement]) =
  ???
```

First implement this without branching instructions. We'll return to branches in a moment.

See the solution

Now using `process` write a structural recursion over `List[Instruction]` that converts the instructions to a `List[PathElement]`. Call this method `iterate` with signature

```
def iterate(state: TurtleState, instructions: List[Instruction]): List[PathElement] =
  ???
```

See the solution

Now add branching to `process`, using `iterate` as a utility.

```
def process(state: TurtleState, instruction: Instruction): (TurtleState, List[PathElement]) = {
  import PathElement._

  instruction match {
    case Forward(d) =>
      val nowAt = state.at + Vec.polar(d, state.heading)
      val element = lineTo(nowAt.toPoint)

      (state.copy(at = nowAt), List(element))
    case Turn(a) =>
      val nowHeading = state.heading + a

      (state.copy(heading = nowHeading), List())
    case Branch(is) =>
     val branchedElements = iterate(state, is)

     (state, moveTo(state.at.toPoint) :: branchedElements)
    case NoOp() =>
      (state, List())
  }
}
```

Now implement `draw` using `iterate`.

See the solution

14.2.1 Extensions

Turtles that can make random choices can lead to more organic images. Can you implement this?

Chapter 15

Summary

In this text we have covered a handful of the essential functional programming tools available in Scala.

15.1 Representations and Interpreters

We started by writing expressions to create and compose images. Each program we wrote went through two distinct phases:

1. Build an `Image`
2. Call the `draw` method to display the image

This process demonstrates two important functional programming patterns: *building intermediate representations* of the result we want, and *interpreting the representations* to produce output.

15.2 Abstraction

Building an intermediate representation allows us to only model the aspects of the result that we consider important and *abstract* irrelevant details.

For example, Doodle directly represents the primitive shapes and geometric relationships in our drawings, without worrying about implementation details such as screen coordinates. This keeps our code clear and maintainable, and limits the number of "magic numbers" we need to write. For example, it is a lot easier to determine that this Doodle program produces a house:

```
def myImage: Image =
  Triangle(50, 50) above Rectangle(50, 50)
// myImage: Image = // ...
```

than this implementation in Java2D:

```
def drawImage(g: Graphics2D): Unit = {
  g.setStroke(new BasicStroke(1.0f))
  g.setPaint(new Color(0, 0, 0))
  val path = new Path2D.Double()
  path.moveTo(25, 0)
  path.lineTo(50, 50)
  path.lineTo(0, 50)
  path.lineTo(25, 0)
```

```
  path.closePath()
  g.draw(path)
  f.drawRect(50, 50, 50, 50)
}
```

It's important to realise that all of the imperative Java2D code is still present in Doodle. The difference is we have hidden it away into the `draw` method. `draw` acts as *interpreter* for our `Images`, filling in all of the details about coordinates, paths, and graphics contexts that we don't want to think about in our code.

Separating the immediate value and the interpreter also allows us to change how interpretation is performed. Doodle already comes with two interpreters, one of which draws in the Java2D framework while the other draws in the HTML canvas. You can image yet more interpreters to, for example, achieve artistic effects such as drawing images in a hand-drawn style.

15.3 Composition

In addition to making our programs clearer, the functional approach employed by Doodle allows us to *compose* images from other images. For example, we can re-use our house to draw a street:

```
val house = Triangle(50, 50) above Rectangle(50, 50)
// house: Image = // ...

val street = house beside house beside house
// street: Image = // ...
```

The `Image` and `Color` values we create are immutable so we can easily re-use a single `house` three times within the same image.

This approach allows us to break down a complex image into simpler parts that we then combine together to create the desired result.

Reusing immutable data, a technique called *structure sharing*, is the basis of many fast, memory efficient immutable data structures. The quintissential example in Doodle is the Sierpinski triangle where we re-used a single `Triangle` object to represent an image containing nearly 20,000 distinct coloured triangles.

15.4 Expression-Oriented Programming

Scala provides convenient syntax to simplify creating data structures in a functional manner. Constructs such as conditionals, loops, and blocks are *expressions*, allowing us to write short method bodies without declaring lots of intermediate variables. We quickly adopt a pattern of writing short methods whose main purpose is to return a value, so omitting the `return` keyword is also a useful shorthand.

15.5 Types are a Safety Net

Scala's type system helps us by checking our code. Every expression has a type that is checked at compile time to see if it matches up with its surroundings. We can even define our own types with the explicit purpose of stopping ourselves from making mistakes.

A simple example of this is Doodle's `Angle` type, which prevents us confusing numbers and angles, and degrees and radians:

```
90
// res0: Int = 90

90.degrees
// res1: doodle.core.Angle = Angle(1.5707963267948966)

90.radians
// res2: doodle.core.Angle = Angle(2.0354056994857643)

90.degrees + 90.radians
// res3: doodle.core.Angle = Angle(3.606202026280661)

90 + 90.degrees
// <console>:20: error: overloaded method value + with alternatives:
//   (x: Double)Double <and>
//   (x: Float)Float <and>
//   (x: Long)Long <and>
//   (x: Int)Int <and>
//   (x: Char)Int <and>
//   (x: Short)Int <and>
//   (x: Byte)Int <and>
//   (x: String)String
// cannot be applied to (doodle.core.Angle)
//              90 + 90.degrees
//                 ^
```

15.6 Functions as Values

We spent a lot of time writing methods to produce values. Methods let us abstract over parameters. For example, the method below abstracts over colours to produce different coloured dots:

```
def dot(color: Color): Image =
  Circle(10) strokeWidth 0 fillColor color
// dot: Color => Image = // ...
```

Coming from object oriented languages, methods are nothing special. More interesting is Scala's ability to turn methods into *functions* that can be passed around as values:

```
def spectrum(shape: Color => Image): Image =
  shape(Color.red) beside shape(Color.blue) beside shape(Color.green)
// spectrum: (Color => Image) => Image = // ...

spectrum(dot)
// res0: Image = // ...
```

We wrote a number of programs that used functions as values, but the quintissential example was the `map` method of `List`. In the Collections chapter we saw how `map` lets us transform sequences without allocating and pushing values onto intermediate buffers:

```
List(1, 2, 3).map(x => x * 2)
// res0: List[Int] = List(2, 4, 6)
```

Functions, and their first class status as values, are hugely important for writing simple, boilerplate-free code.

15.7 Final Words

The intention of this book has been to introduce you to the functional parts of Scala. These are what differentiate Scala from older commercial languages such as Java and C. However, this is only part of Scala's story. Many modern languages support functional programming, including Ruby, Python, Javascript, and Clojure. How does Scala relate to these languages, and why would you want to choose it over the other available options?

Perhaps the most significant draw to Scala is its type system. This distinguishes Scala from popular languages such as Ruby, Python, Javascript, and Clojure, which are dynamically typed. Having static types in a language is undeniably a trade-off—writing code is slower because we have to satisfy the compiler at every stage. However, once our code compiles we gain confidence about its quality.

Another major draw is Scala's blending of object-oriented and functional programming paradigms. We saw a little of this in the first chapter—every value is an object with methods, fields, and a class (its type). However, we haven't created any of our own data types in this book. Creating types is synonymous with declaring classes, and Scala supports a full gamut of features such as classes, traits, interitance, and generics.

Finally, a major benefit of Scala is its compatibility with Java. In many ways Scala can be seen as a superset of Java, and interoperation between the two languages is quite straightforward. This opens up a world of Java libraries to our Scala applications, and allows flexibility when translating Java applications to Scala.

15.8 Next Steps

We hope you enjoyed Creative Scala and drawing diagrams with Doodle. If you would like to learn more about Scala, we recommend that you pick one of the many great books available on the language.

Our own book, Essential Scala, is available from our web site and continues Creative Scala's approach of teaching Scala by discussing and demonstrating core design patterns and the benefits they offer.

If you want to challenge yourself, try drawing something more complex with Doodle and sharing it with us via Gitter. There are lots of things you can try—check the `examples` directory in the Doodle codebase for some suggestions:

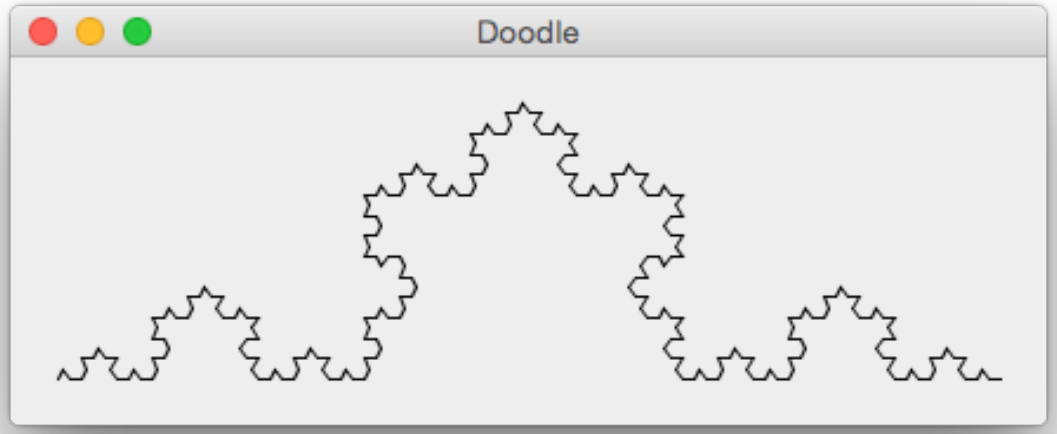

Figure 15.1: Koch Triangle (Koch.scala)

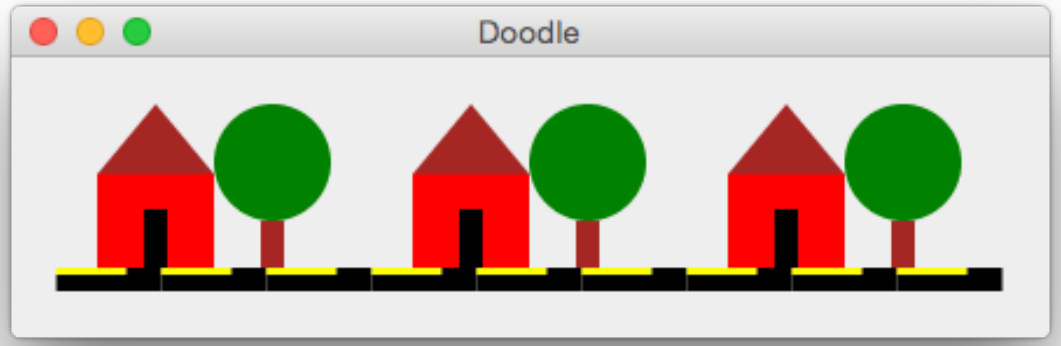

Figure 15.2: Suburban Scene (Street.scala)

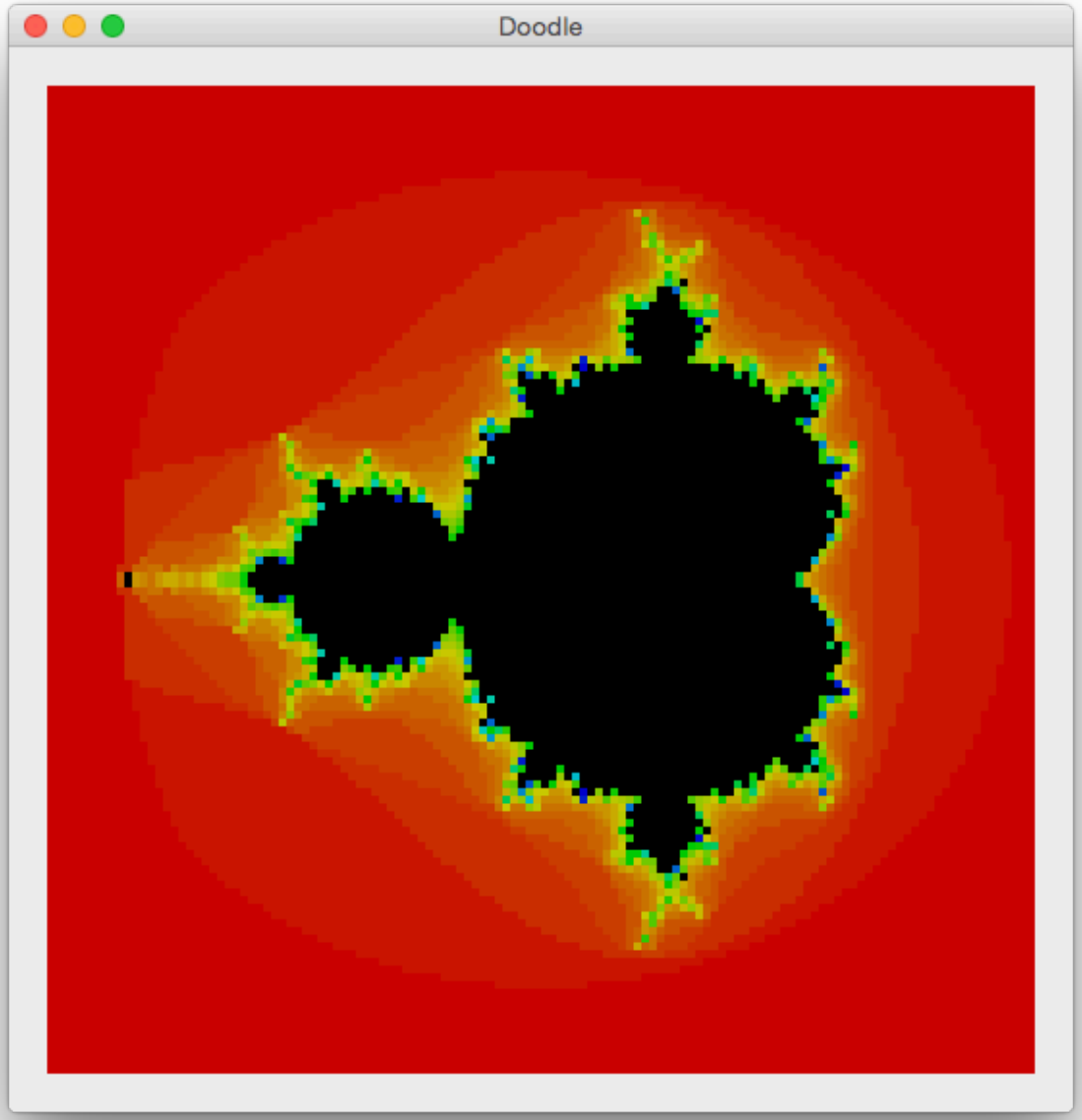

Figure 15.3: Mandelbrot Fractal by Mat Moore (Mandelbrot.scala)

Appendix A

Syntax Quick Reference

A.1 Literals and Expressions

```
// Literals:
123      // Int
123.0    // Double
"Hello!" // String
true     // Boolean

// Math:
10 + 2   // Int + Int    = Int
10 + 2.0 // Int + Double = Double
10 / 2   // Int / Int    = Double

// Boolean logic:
true && false // logical AND
true || false // logical OR
!true         // logical NOT

// String concatenation:
"abc" + "def" // String
"abc" + 123   // auto-conversion from Int to String

// Method calls and infix operators:
1.+(2)    // method call style
1 + 2     // infix operator style
1 + 2 + 3 // equivalent to 1.+(2).+(3)

// Conditionals:
if(booleanExpression) expressionA else expressionB

// Blocks:
{
  sideEffectExpression1
  sideEffectExpression2
  resultExpression
}
```

A.2 Value and Method Declarations

```
// Value declaration syntax:
val valueName: SomeType = resultExpression // declaration with explicit type
val valueName = resultExpression           // declaration with inferred type

// Method with parameter list and explicit return type:
def methodName(argName: ArgType, argName: ArgType): ReturnType =
  resultExpression

// Method with parameter list and inferred return type:
def methodName(argName: ArgType, argName: ArgType) =
  resultExpression

// Multi-expression method (using a block):
def methodName(argName: ArgType, argName: ArgType): ReturnType = {
  sideEffectExpression1
  sideEffectExpression2
  resultExpression
}

// Method with no parameter list:
def methodName: ReturnType =
  resultExpression

// Calling a method that has a parameter list:
methodName(arg, arg)

// Calling a method that has no parameter list:
methodName
```

A.3 Functions as Values

Function values are written `(argName: ArgType, ...) => resultExpression`:

```
val double = (num: Int) => num * 2
// double: Int => Int = <function1>

val sum = (a: Int, b: Int) => a + b
sum: (Int, Int) => Int = <function2>
```

Multi-line functions are written using block expressions:

```
val printAndDouble = (num: Int) => {
  println("The number was " + num)
  num * 2
}
// printAndDouble: Int => Int = <function1>

scala> printAndDouble(10)
// The number was 10
// res0: Int = 20
```

We have to write function types when declaring parameters and return types. The syntax is `ArgType => ResultType` or `(ArgType, ...) => ResultType`:

```
def doTwice(value: Int, func: Int => Int): Int =
  func(func(value))
// doTwice: (value: Int, func: Int => Int)Int

doTwice(1, double)
// res0: Int = 4
```

Function values can be written inline as normal expressions:

```
doTwice(1, (num: Int) => num * 10)
// res1: Int = 100
```

We can sometimes omit the argument types, assuming the compiler can figure things out for us:

```
doTwice(1, num => num * 10)
// res2: Int = 100
```

A.4 Doodle Reference Guide

A.4.1 Imports

```
// These imports get you everything you need:
import doodle.core._
import doodle.syntax._
```

A.4.2 Creating Images

```
// Primitive images (black outline, no fill):
val i: Image = Circle(radius)
val i: Image = Rectangle(width, height)
val i: Image = Triangle(width, height)

// Compound images written using operator syntax:
val i: Image = imageA beside imageB // horizontally adjacent
val i: Image = imageA above  imageB // vertically adjacent
val i: Image = imageA below  imageB // vertically adjacent
val i: Image = imageA on     imageB // superimposed
val i: Image = imageA under  imageB // superimposed

// Compound images written using method call syntax:
val i: Image = imageA.beside(imageB)
// etc...
```

A.4.3 Styling Images

```
// Styling images written using operator syntax:
val i: Image = image fillColor color   // new fill color (doesn't change line)
val i: Image = image strokeColor color   // new line color (doesn't change fill)
val i: Image = image strokeWidth integer // new line width (doesn't change fill)
val i: Image = image fillColor color strokeColor otherColor // new fill and line

// Styling images using method call syntax:
val i: Image = imageA.fillColor(color)
val i: Image = imageA.fillColor(color).strokeColor(otherColor)
```

```
// etc...
```

A.4.4 Colours

```
// Basic colors:
val c: Color = Color.red                        // predefined colors
val c: Color = Color.rgb(255.uByte, 127.uByte, 0.uByte)          // RGB color
val c: Color = Color.rgba(255.uByte, 127.uByte, 0.uByte, 0.5.normalized)    // RGBA color
val c: Color = Color.hsl(15.degrees, 0.25.normalized, 0.5.normalized)       // HSL color
val c: Color = Color.hsla(15.degrees, 0.25.normalized, 0.5.normalized, 0.5.normalized) // HSLA color

// Transforming/mixing colors using operator syntax:
val c: Color = someColor spin       10.degrees     // change hue
val c: Color = someColor lighten    0.1.normalized // change brightness
val c: Color = someColor darken     0.1.normalized // change brightness
val c: Color = someColor saturate   0.1.normalized // change saturation
val c: Color = someColor desaturate 0.1.normalized // change saturation
val c: Color = someColor fadeIn     0.1.normalized // change opacity
val c: Color = someColor fadeOut    0.1.normalized // change opacity

// Transforming/mixing colors using method call syntax:
val c: Color = someColor.spin(10.degrees)
val c: Color = someColor.lighten(0.1.normalized)
// etc...
```

A.4.5 Paths

```
// Create path from list of PathElements:
val i: Image = OpenPath(List(
  MoveTo(Vec(0, 0).toPoint),
  LineTo(Vec(10, 10).toPoint)
))

// Create path from other sequence of PathElements:
val i: Image = OpenPath(
  (0 until 360 by 30) map { i =>
    LineTo(Vec.polar(i.degrees, 100).toPoint)
  }
)

// Types of element:
val e1: PathElement = MoveTo(toVec.toPoint)                        // no line
val e2: PathElement = LineTo(toVec.toPoint)                        // straight line
val e3: PathElement = BezierCurveTo(cp1Vec.toPoint, cp2Vec.toPoint, toVec.toPoint) // curved line

// NOTE: If the first element isn't a MoveTo,
//       it is converted to one
```

A.4.6 Angles and Vecs

```
val a: Angle = 30.degrees               // angle in degrees
val a: Angle = 1.5.radians              // angle in radians
val a: Angle = math.Pi.radians          // π radians
val a: Angle = 1.turns                  // angle in complete turns

val v: Vec = Vec.zero                   // zero vector (0,0)
```

```
val v: Vec = Vec.unitX                  // unit x vector (1,0)
val v: Vec = Vec.unitY                  // unit y vector (0,1)

val v: Vec = Vec(3, 4)                  // vector from cartesian coords
val v: Vec = Vec.polar(30.degrees, 5)   // vector from polar coords
val v: Vec = Vec(2, 1) * 10             // multiply length
val v: Vec = Vec(20, 10) / 10           // divide length
val v: Vec = Vec(2, 1) + Vec(1, 3)      // add vectors
val v: Vec = Vec(5, 5) - Vec(2, 1)      // subtract vectors
val v: Vec = Vec(5, 5) rotate 45.degrees // rotate counterclockwise

val x: Double = Vec(3, 4).x             // x coordinate
val y: Double = Vec(3, 4).y             // y coordinate
val a: Angle  = Vec(3, 4).angle         // counterclockwise from (1, 0)
val l: Double = Vec(3, 4).length        // length
```

Appendix B

Solutions to Exercises

B.1 Expressions, Values, and Types

B.1.1 Solution to: Arithmetic

This exercise is just about getting used to writing Scala code. Here is one possible solution.

```
1 + 43 - 2
// res0: Int = 42
```

Return to the exercise

B.1.2 Solution to: Appending Strings

Something like the below should do.

```
"It is a truth ".++("universally acknowledged")
// res1: String = "It is a truth universally acknowledged"
"It is a truth " ++ "universally acknowledged"
// res2: String = "It is a truth universally acknowledged"
```

Return to the exercise

B.1.3 Solution to: Precedence

A bit of exploration at the console should convince you that yes, Scala does maintain the standard precedence rules. The example below demonstrates this.

```
1 + 2 * 3
// res3: Int = 7
1 + (2 * 3)
// res4: Int = 7
(1 + 2) * 3
// res5: Int = 9
```

Return to the exercise

B.1.4 Solution to: Types and Values

```
1 + 2
// res12: Int = 3
```

This expression has type `Int` and evaluates to 3.

```
"3".toInt
// res13: Int = 3
```

This expression has type `Int` and evaluates to 3.

```
"Electric blue".toInt
// java.lang.NumberFormatException: For input string: "Electric blue"
//  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
//  at java.lang.Integer.parseInt(Integer.java:580)
//  at java.lang.Integer.parseInt(Integer.java:615)
//  at scala.collection.immutable.StringLike.toInt(StringLike.scala:304)
//  at scala.collection.immutable.StringLike.toInt$(StringLike.scala:304)
//  at scala.collection.immutable.StringOps.toInt(StringOps.scala:33)
//  at repl.Session$App$$anonfun$17.apply$mcI$sp(exercises.md:100)
//  at repl.Session$App$$anonfun$17.apply(exercises.md:100)
//  at repl.Session$App$$anonfun$17.apply(exercises.md:100)
```

This expression has type `Int` but fails at run-time.

```
"Electric blue".take(1)
// res14: String = "E"
```

This expression has type `String` and evaluates to `"E"`.

```
"Electric blue".take("blue")
// error: type mismatch;
//  found   : String("blue")
//  required: Int
// "Electric blue".take("blue")
//                      ^^^^^^
```

This expression fails at compile-time and hence has no type.

```
1 + ("Moonage daydream".indexOf("N"))
// res16: Int = 0
```

This expression has type `Int` and evaluates to `0`.

```
1 / 1 + ("Moonage daydream".indexOf("N"))
// res17: Int = 0
```

This expression has type `Int` and, due to precedence, evaluates to `(1 / 1) + -1`, which is `0`.

```
1 / (1 + ("Moonage daydream".indexOf("N")))
// java.lang.ArithmeticException: / by zero
//  at repl.Session$App$$anonfun$22.apply$mcI$sp(exercises.md:132)
//  at repl.Session$App$$anonfun$22.apply(exercises.md:132)
//  at repl.Session$App$$anonfun$22.apply(exercises.md:132)
```

This expression has type `Int` but fails at run-time with a division by zero.

Return to the exercise

B.1.5 Solution to: Floating Point Failings

`Double` is an approximation because it has the fit within the computer's finite memory. A `Double` takes up precisely 64-bits, which is enough space to store a lot of digits but not enough to store a number that, like π, has an infinite expansion.

The number ⅓ also has an infinite expansion in decimal. Because Doubles are stored in binary there are some numbers that can be represented in a finite number of decimal digits but have no finite representation in binary. 0.1 turns out to be one such number.

In general, floating point numbers can lead to nasty surprises if you expect them to act like the reals. They are fine for our purposes in Creative Scala, but don't go using them to write accounting software!

Return to the exercise

B.1.6 Solution to: Beyond Expressions

This is very open ended question. There are several ways to go beyond the model we have so far.

To be useful our programs must be capable of creating effects—changes in the world that go beyond the computer's memory. For example, displaying things on the screen, making sound, sending messages to other computers, and the like. The console implicitly does some of this for us, by printing values on the screen. We'll need to go a bit beyond that for more useful programs.

We also don't have any way to define our own objects and methods, or reuse values in our programs. If we want to, say, use someone's name across a program we have to repeat that name everywhere. We need more methods of *abstraction* and that's what we'll turn to soon.

Return to the exercise

B.2 Computing With Pictures

B.2.1 Solution to: I Go Round in Circles

In this exercise we're checking that our Doodle install is working correctly and we're getting used to using the library. One of the important points in Doodle is we separate *defining the image* from *drawing the image*. We'll talk more about this throughout the book.

We can create circles with the code below.

```
Image.circle(1)
Image.circle(10)
Image.circle(100)
```

We can draw the circles by calling the `draw` method on each circle.

```
Image.circle(1).draw()
Image.circle(10).draw()
Image.circle(100).draw()
```

Return to the exercise

B.2.2 Solution to: My Type of Art

They all have type `Image`, as we can tell from the console.

```
:type Image.circle(10)
// doodle.core.Image
:type Image.rectangle(10, 10)
// doodle.core.Image
:type Image.triangle(10, 10)
// doodle.core.Image
```

Return to the exercise

B.2.3 Solution to: Not My Type of Art

Once again, we can ask the console this quesstion.

```
:type Image.circle(10).draw()
// Unit
```

We see that the type of drawing an image is `Unit`. `Unit` is the type of expressions that have no interesting value to return. This is the case for `draw`; we call it because we want something to appear on the screen, not because we have a use for the value it returns. There is only one value with type `Unit`. This value is also called unit, which written as a literal expression is `()`

You'll note that the console doesn't print unit by default.

```
()
```

We can ask the console for the type to show that there really is unit here.

```
:type ()
// Unit
```

Return to the exercise

B.2.4 Solution to: The Width of a Circle

It's three small circles on top of a bigger circle, and we can just about state this as is in code.

```
(Image
   .circle(20)
   .beside(Image.circle(20))
   .beside(Image.circle(20))).on(Image.circle(60))
// res0: Image = On(
//   Beside(Beside(Circle(20.0), Circle(20.0)), Circle(20.0)),
//   Circle(60.0)
```

```
// )
```

Return to the exercise

B.2.5 Solution to: Evil Eye

Here's my amulet:

```
Image
  .circle(10)
  .fillColor(Color.black)
  .on(Image.circle(20).fillColor(Color.cornflowerBlue))
  .on(Image.circle(30).fillColor(Color.white))
  .on(Image.circle(50).fillColor(Color.darkBlue))
// res0: Image = On(
//   On(
//     On(
//       FillColor(
//         Circle(10.0),
//         RGBA(
//           UnsignedByte(-128),
//           UnsignedByte(-128),
//           UnsignedByte(-128),
//           Normalized(1.0)
//         )
//       ),
//       FillColor(
//         Circle(20.0),
//         RGBA(
//           UnsignedByte(-28),
//           UnsignedByte(21),
//           UnsignedByte(109),
//           Normalized(1.0)
//         )
//       )
//     ),
//     FillColor(
//       Circle(30.0),
//       RGBA(
//         UnsignedByte(127),
//         UnsignedByte(127),
//         UnsignedByte(127),
//         Normalized(1.0)
//       )
//     )
//   ),
//   FillColor(
//     Circle(50.0),
//     RGBA(
//       UnsignedByte(-128),
//       UnsignedByte(-128),
//       UnsignedByte(11),
//       Normalized(1.0)
//     )
//   )
// )
```

Return to the exercise

B.2.6 Solution to: Analogous Triangles

These sort of examples are getting a bit too long to write out at the console. We'll look at a way around this next.

```
Image.triangle(40, 40)
  .strokeWidth(6.0)
  .strokeColor(Color.darkSlateBlue)
  .fillColor(Color.darkSlateBlue
              .lighten(0.3.normalized)
              .saturate(0.2.normalized)
              .spin(10.degrees))
  .above(Image.triangle(40, 40)
           .strokeWidth(6.0)
           .strokeColor(Color.darkSlateBlue.spin(-30.degrees))
           .fillColor(Color.darkSlateBlue
                       .lighten(0.3.normalized)
                       .saturate(0.2.normalized)
                       .spin(-20.degrees))
           .beside(Image.triangle(40, 40)
                     .strokeWidth(6.0)
                     .strokeColor(Color.darkSlateBlue.spin(30.degrees))
                     .fillColor(Color.darkSlateBlue
                                 .lighten(0.3.normalized)
                                 .saturate(0.2.normalized)
                                 .spin(40.degrees))))
// res15: Image = Above(
//   FillColor(
//     StrokeColor(
//       StrokeWidth(Triangle(40.0, 40.0), 6.0),
//       RGBA(
//         UnsignedByte(-56),
//         UnsignedByte(-67),
//         UnsignedByte(11),
//         Normalized(1.0)
//       )
//     ),
//     HSLA(
//       Angle(4.5110048359238055),
//       Normalized(0.5899999999999999),
//       Normalized(0.692156862745098),
//       Normalized(1.0)
//     )
//   ),
//   Beside(
//     FillColor(
//       StrokeColor(
//         StrokeWidth(Triangle(40.0, 40.0), 6.0),
//         HSLA(
//           Angle(3.812873135126074),
//           Normalized(0.3899999999999999),
//           Normalized(0.39215686274509803),
//           Normalized(1.0)
//         )
//       ),
//       HSLA(
//         Angle(3.987406060325507),
//         Normalized(0.5899999999999999),
//         Normalized(0.692156862745098),
//         Normalized(1.0)
```

```
//       )
//     ),
//     FillColor(
//       StrokeColor(
//         StrokeWidth(Triangle(40.0, 40.0), 6.0),
//         HSLA(
//           Angle(4.860070686322672),
//           Normalized(0.3899999999999999),
//           Normalized(0.39215686274509803),
//           Normalized(1.0)
//         )
//       ),
//       HSLA(
//         Angle(5.034603611522105),
//         Normalized(0.5899999999999999),
// ...
```

Return to the exercise

B.2.7 Solution to: Compilation Target

The simplest solution is to create three concentric circles using the on method:

```
Image
  .circle(20)
  .on(Image.circle(40))
  .on(Image.circle(60))
```

For the extra credit we can create a stand using two rectangles:

```
Image
  .circle(20)
  .on(Image.circle(40))
  .on(Image.circle(60))
  .above(Image.rectangle(6, 20))
  .above(Image.rectangle(20, 6))
```

Return to the exercise

B.2.8 Solution to: Stay on Target

The trick here is using parentheses to control the order of composition. The `fillColor()`, `strokeColor()`, and `strokeWidth()` methods apply to a single image—we need to make sure that image comprises the correct set of shapes:

```
Image
  .circle(20).fillColor(Color.red)
  .on(Image.circle(40).fillColor(Color.white))
  .on(Image.circle(60).fillColor(Color.red))
  .above(Image.rectangle(6, 20).fillColor(Color.brown))
  .above(Image.rectangle(20, 6).fillColor(Color.brown))
  .above(Image.rectangle(80, 25).noStroke.fillColor(Color.green))
```

Return to the exercise

B.3 Writing Larger Programs

B.3.1 Solution to: The Top-Level

No, Scala doesn't allow us to do this. For example, we can't write

```
object {}
```

We have to give a name to any object literal we create.

Return to the exercise

B.3.2 Solution to: The Top-Level Part 2

We sure can!

We can put a `val` inside an object literal like so:

```
object Example {
  val hi = "Hi!"
}
```

We can then refer to it using the `.` syntax we've been using already.

```
Example.hi
// res4: String = "Hi!"
```

Note that we can't use `hi` on it's own

```
hi
// error: not found: value hi
```

We have to tell Scala we want to refer to the name `hi` defined inside the object `Example`.

Return to the exercise

B.3.3 Solution to: Exercises

A simple example to get started with. `answer` is `1 + 2`, which is 3.

Return to the exercise

B.3.4 Solution to: Exercises Part 2

Another simple example. `answer` is `1 + 2`, which is 3. `Two.a` is not in scope where `answer` is defined.

Return to the exercise

B.3.5 Solution to: Exercises Part 3

Here `Answer.a` shadows `One.a` so `answer` is `1 + 2`, which is 3.

Return to the exercise

B.3.6 Solution to: Exercises Part 4

This is perfectly fine. The expression `a + 1` on the right hand side of the declaration of `b` is an expression like any other so `answer` is 3 again.

Return to the exercise

B.3.7 Solution to: Exercises Part 5

This code doesn't compile as `b` is not in scope where `answer` is declared.

Return to the exercise

B.3.8 Solution to: Exercises Part 6

Trick question! This code doesn't work. Here `a` and `b` are defined in terms of each other which leads to a circular dependency that can't be resolved.

Return to the exercise

B.3.9 Solution to: Archery Again

We decided to name the target, stand, and ground, as shown below. This makes is clear how the final image is constructed. Naming more components seemed to us that it would not aid comprehension.

```
val coloredTarget =
  (
    Image.circle(10).fillColor(Color.red) on
      Image.circle(20).fillColor(Color.white) on
      Image.circle(30).fillColor(Color.red)
  )

val stand =
  Image.rectangle(6, 20) above Image.rectangle(20, 6).fillColor(Color.brown)

val ground =
  Image.rectangle(80, 25).strokeWidth(0).fillColor(Color.green)

val image = coloredTarget above stand above ground
```

Return to the exercise

B.3.10 Solution to: Streets Ahead

Here's our solution. As you can see, by breaking the scene down into smaller components we were able to write relatively little code.

```
val roof = Image.triangle(50, 30) fillColor Color.brown

val frontDoor =
  (Image.rectangle(50, 15) fillColor Color.red) above (
    (Image.rectangle(10, 25) fillColor Color.black) on
    (Image.rectangle(50, 25) fillColor Color.red)
  )
```

```
val house = roof above frontDoor

val tree =
  (
    (Image.circle(25) fillColor Color.green) above
    (Image.rectangle(10, 20) fillColor Color.brown)
  )

val streetSegment =
  (
    (Image.rectangle(30, 3) fillColor Color.yellow) beside
    (Image.rectangle(15, 3) fillColor Color.black) above
    (Image.rectangle(45, 7) fillColor Color.black)
  )

val street = streetSegment beside streetSegment beside streetSegment

val houseAndGarden =
  (house beside tree) above street

val image = (
  houseAndGarden beside
  houseAndGarden beside
  houseAndGarden
) strokeWidth 0
```

Return to the exercise

B.4 The Substitution Model of Evaluation

B.4.1 Solution to: No Substitute for Println

Here is a simple example that illustrates this. The following two programs are observably different.

```
println("Happy birthday to you!")
// Happy birthday to you!
println("Happy birthday to you!")
// Happy birthday to you!
println("Happy birthday to you!")
// Happy birthday to you!

val happy = println("Happy birthday to you!")
// Happy birthday to you!
happy
happy
happy
```

Therefore we cannot freely use substitution in the presence of side effects, and we must be aware of the order of evaluation.

Return to the exercise

B.4.2 Solution to: Madness to our Methods

The following code demonstrates that method parameters are evaluated from left to right.

```
Color.hsl(
  {
    println("a")
    0.degrees
  },
  {
    println("b")
    1.0
  },
  {
    println("c")
    1.0
  }
)
// a
// b
// c
// res14: Color = HSLA(
//   Angle(0.0),
//   Normalized(1.0),
//   Normalized(1.0),
//   Normalized(1.0)
// )
```

We can write this more compactly as

```
Color.hsl({ println("a"); 0.degrees },
          { println("b"); 1.0 },
          { println("c"); 1.0 })
// a
// b
// c
// res15: Color = HSLA(
//   Angle(0.0),
//   Normalized(1.0),
//   Normalized(1.0),
//   Normalized(1.0)
// )
```

Return to the exercise

B.4.3 Solution to: The Last Order

We've already seen that expressions are evaluated from top-to-bottom, and method parameters are evaluated from left-to-right. We might want to check that expressions are in general evaluated left-to-right. We can show this fairly easily.

```
{ println("a"); 1 } + { println("b"); 2 } + { println("c"); 3}
// a
// b
// c
// res16: Int = 6
```

So in conclusion we can say that Scala expressions are evaluated from top-to-bottom and left-to-right.

Return to the exercise

B.5 Methods

B.5.1 Solution to: Square

The solution is

```
def square(x: Int): Int =
  x * x
```

We can arrive at the solution by the following steps.

We're given the name (`square`), the type of the parameter (`Int`), and the type of the result (`Int`). From this we can write the method skeleton

```
def square(x: Int): Int =
  ???
```

where we have chosen `x` as the name of the parameter. This is a fairly arbitrary choice. Where there is no meaningful name you often see one-letter names such as `x`, `v`, or `i` used.

By the way this is valid code. Enter it into the console and see! What happens if you call `square` when it's defined like so?

Now we need to complete the body. We've been told that squaring is multiplying a number by itself, so `x * x` is what we replace the `???` with. We don't need to wrap this in braces as there is only a single expression in the body.

Return to the exercise

B.5.2 Solution to: Halve

```
def halve(x: Double): Double =
 x / 2.0
```

We can follow the same process as for `square` above to arrive at the solution.

Return to the exercise

B.5.3 Solution to: Exercise

The following program demonstrates that parameters are evaluated before the method body.

```
def example(a: Int, b: Int): Int = {
  println("In the method body!")
  a + b
}

example({ println("a"); 1 }, { println("b"); 2 })
// a
// b
// In the method body!
// res6: Int = 3
```

The alternative we described above is used by some languages, most notably Haskell, and is known as lazy or non-strict evaluation.

Return to the exercise

B.6 Structural Recursion

B.6.1 Solution to: Stacking Boxes

All you to do is change `beside` to `above` in `boxes`.

```
def stackedBoxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox.beside(stackedBoxes(n-1))
  }
```

Return to the exercise

B.6.2 Solution to: Guess the Result

The first example evaluates to 2, as the pattern `"abcd"` is the only match for the literal `"abcd"` amongst the patterns.

The second example evaluates to `"one"`, because the first matching case is the one that is evaluated.

The third example evaluates to 2, because `case n` defines a wildcard pattern that matches anything.

The final example evaluates to 1 because the first matching case is evaluated.

Return to the exercise

B.6.3 Solution to: No Match

Here are three reasonable possibilities I can think of; perhaps you came up with something else?

- The expression could evaluate to some default, like `Image.empty` (but how should Scala pick the right default?)
- The Scala compiler should just not let you write code like that.
- The `match` expression will fail at runtime.

Here's a match expression that doesn't match.

```
2 match {
  case 0 => "zero"
  case 1 => "one"
}
// scala.MatchError: 2 (of class java.lang.Integer)
//  at repl.Session$App$$anonfun$5.apply(match.md:37)
```

The correct answer is one of the last two possibilities, failing to compile or failing at runtime. In this example we have an error at runtime. The exact answer depends on how Scala is configured (we can tell the compiler to refuse to compile matches that it can show are not exhaustive, but this is not the default behavior).

Return to the exercise

B.6.4 Solution to: Three (or More) Stacks

This is a modification of `boxes`, replacing `beside` with `above`.

```
def stacks(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox.above(boxes(n-1))
  }
```

Return to the exercise

B.6.5 Solution to: Alternating Images

Here's my solution. I used an `if` expression because it's a bit shorter than matching on the `Boolean`. Otherwise it's the same structural recursion pattern as before.

```
val star = Image
  .star(5, 30, 15, 45.degrees)
  .strokeColor(Color.teal)
  .on(Image.star(5, 12, 7, 45.degrees).strokeColor(Color.royalBlue))
  .strokeWidth(5.0)

val circle = Image
  .circle(60)
  .strokeColor(Color.midnightBlue)
  .on(Image.circle(24).strokeColor(Color.plum))
  .strokeWidth(5.0)

def alternatingRow(count: Int): Image = {
  count match {
    case 0 => Image.empty
    case n =>
      if(n % 2 == 0) star.beside(alternatingRow(n-1))
      else circle.beside(alternatingRow(n-1))
  }
}
```

Return to the exercise

B.6.6 Solution to: Getting Creative

Here's my solution. I made the size of the star and its color depend on the counter.

```
def funRow(count: Int): Image = {
  count match {
    case 0 => Image.empty
    case n =>
      Image
        .star(7, (10 * n), (7 * n), 45.degrees)
        .strokeColor(Color.azure.spin((5 * n).degrees))
        .strokeWidth(7.0)
        .beside(funRow(n - 1))
  }
}
```

Return to the exercise

B.6.7 Solution to: Cross

It's structural recursion over the natural numbers. You should end up with something like

```
def cross(count: Int): Image =
  count match {
    case 0 => <resultBase>
    case n => <resultUnit> <add> cross(n-1)
  }
```

Return to the exercise

B.6.8 Solution to: Cross Part 2

From the picture we can work out that the base case is the hexagon in red.

Successive elements in the picture add circles to the top, bottom, left, and right of the image. So our unit is a circle, but the addition operator is not a simple `beside` or `above` like we've seen before but `unit.beside(unit.above(cross(n-1)).above(unit)).beside(unit)`.

Return to the exercise

B.6.9 Solution to: Cross Part 3

Here's what we wrote.

```
def cross(count: Int): Image = {
    count match {
      case 0 =>
        Image.regularPolygon(6, 10, 0.degrees)
          .strokeColor(Color.deepSkyBlue.spin(-180.degrees))
          .strokeWidth(5.0)
      case n =>
        val circle = Image
          .circle(20)
          .strokeColor(Color.deepSkyBlue)
          .on(Image.circle(15).strokeColor(Color.deepSkyBlue.spin(-15.degrees)))
          .on(Image.circle(10).strokeColor(Color.deepSkyBlue.spin(-30.degrees)))
          .strokeWidth(5.0)
        circle.beside(circle.above(cross(n - 1)).above(circle)).beside(circle)
    }
  }
```

Return to the exercise

B.6.10 Solution to: Exercises

It sure does! The base case is straightforward. Looking at the recursive case, we assume that `identity(n-1)` returns the identity for `n-1` (which is exactly `n-1`). The identity for n is then `1 + identity(n-1)`.

Return to the exercise

B.6.11 Solution to: Exercises Part 2

No way! This method is broken in two different ways. Firstly, because we're multiplying in the recursive case we will eventualy end up multiplying by base case of zero, and therefore the entire result will be zero.

We might try and fix this by adding a case for 1 (and perhaps wonder why the structural recursion skeleton let us down).

```
def double(n: Int): Int =
  n match {
    case 0 => 0
    case 1 => 1
    case n => 2 * double(n-1)
  }
```

This doesn't give us the correct result, however! We're doing the wrong thing at the recursive case: we should be adding, not multiplying.

A bit of algebra:

```
2(n-1 + 1) == 2(n-1) + 2
```

So if `double(n-1)` is `2(n-1)` then we should *add* 2, not multiply by 2. The correct method is

```
def double(n: Int): Int =
  n match {
    case 0 => 0
    case n => 2 + double(n-1)
  }
```

Return to the exercise

B.7 Fractals

B.7.1 Solution to: The Chessboard

`chessboard` is a structural recursion over the natural numbers, so right away we can write down the skeleton for this pattern.

```
def chessboard(count: Int): Image =
  count match {
    case 0 => resultBase
    case n => resultUnit add chessboard(n-1)
  }
```

As before we must decide on the base, unit, and addition for the result. We've given you a hint by showing the progression of chessboards in fig. **??**. From this we can see that the base is a two-by-two chessboard.

```
val blackSquare = Image.rectangle(30, 30).fillColor(Color.black)
val redSquare   = Image.rectangle(30, 30).fillColor(Color.red)

val base =
  (redSquare.beside(blackSquare)).above(blackSquare.beside(redSquare))
```

Now to work out the unit and addition. Here we see a change from previous examples. The unit is the value we get from the recursive call `chessboard(n-1)`. The addition operation is `(unit beside unit) above (unit beside unit)`.

Putting it all together we get

```
def chessboard(count: Int): Image = {
  val blackSquare = Image.rectangle(30, 30).fillColor(Color.black)
  val redSquare   = Image.rectangle(30, 30).fillColor(Color.red)

  val base =
    (redSquare.beside(blackSquare)).above(blackSquare.beside(redSquare))
  count match {
    case 0 => base
    case n =>
      val unit = chessboard(n-1)
      (unit.beside(unit)).above(unit.beside(unit))
  }
}
```

Return to the exercise

B.7.2 Solution to: Sierpinkski Triangle

The key step is to recognise that the basic unit of the Sierpinski triangle is `triangle above (triangle beside triangle)`. Once we've worked this out, the code has exactly the same structure as `chessboard`. Here's our implementation.

```
def sierpinski(count: Int): Image = {
  val triangle = Image.triangle(10, 10).strokeColor(Color.magenta)
  count match {
    case 0 => triangle.above(triangle.beside(triangle))
    case n =>
      val unit = sierpinski(n-1)
      unit.above(unit.beside(unit))
  }
}
```

Return to the exercise

B.7.3 Solution to: Gradient Boxes

There are two ways to implement a solution. The auxiliary parameter method is to add an extra parameter to `gradientBoxes` and pass the `Color` through the structural recursion.

```
def gradientBoxes(n: Int, color: Color): Image =
  n match {
    case 0 => Image.empty
    case n =>
      aBox
        .fillColor(color)
        .beside(gradientBoxes(n - 1, color.spin(15.degrees)))
  }
```

We could also make the fill color a function of n, as we demonstrated with the box size in `growingBoxes` above.

```
def gradientBoxes(n: Int): Image =
  n match {
    case 0 => Image.empty
    case n =>
      aBox
        .fillColor(Color.royalBlue.spin((15 * n).degrees))
        .beside(gradientBoxes(n - 1))
  }
```

Return to the exercise

B.7.4 Solution to: Concentric Circles

This is almost identical to `growingBoxes`.

```
def concentricCircles(count: Int, size: Int): Image =
  count match {
    case 0 => Image.empty
    case n =>
      Image
        .circle(size)
        .on(concentricCircles(n-1, size + 5))
  }
```

Return to the exercise

B.7.5 Solution to: Once More, With Feeling

Here's our solution, where we've tried to break the problem into reusable parts and reduce the amount of repeated code. We still have a lot of repetition as we don't yet have the tools to get rid of more. We'll come to that soon.

```
def circle(size: Int, color: Color): Image =
  Image.circle(size).strokeWidth(3.0).strokeColor(color)

def fadeCircles(n: Int, size: Int, color: Color): Image =
  n match {
    case 0 => Image.empty
    case n =>
      circle(size, color)
        .on(fadeCircles(n-1, size+7, color.fadeOutBy(0.05.normalized)))
  }

def gradientCircles(n: Int, size: Int, color: Color): Image =
  n match {
    case 0 => Image.empty
    case n =>
      circle(size, color)
        .on(gradientCircles(n-1, size+7, color.spin(15.degrees)))
  }

def image: Image =
  fadeCircles(20, 50, Color.red)
    .beside(gradientCircles(20, 50, Color.royalBlue))
```

Return to the exercise

B.7.6 Solution to: Chessboard

Here's how we did it. It has exactly the same pattern we used with `boxes`.

```
def chessboard(count: Int): Image = {
  val blackSquare = Image.square(30) fillColor Color.black
  val redSquare   = Image.square(30) fillColor Color.red
  val base =
    (redSquare   beside blackSquare) above (blackSquare beside redSquare)
  def loop(count: Int): Image =
    count match {
      case 0 => base
      case n =>
        val unit = loop(n-1)
        (unit beside unit) above (unit beside unit)
    }

  loop(count)
}
```

Return to the exercise

B.7.7 Solution to: Boxing Clever

We can do this in two stages, first moving `aBox` within `boxes`.

```
def boxes(count: Int): Image = {
  val aBox = Image.square(20).fillColor(Color.royalBlue)
  count match {
    case 0 => Image.empty
    case n => aBox beside boxes(n-1)
  }
}
```

Then we can use an internal method to avoid recreating `aBox` on every recursion.

```
def boxes(count: Int): Image = {
  val aBox = Image.square(20).fillColor(Color.royalBlue)
  def loop(count: Int): Image =
    count match {
      case 0 => Image.empty
      case n => aBox beside loop(n-1)
    }

  loop(count)
}
```

Return to the exercise

B.8 Horticulture and Higher-order Functions

B.8.1 Solution to: Function Literals

The first function is

```
(x: Int) => x * x
// res7: Int => Int = <function1>
```

The second is

```
(c: Color) => c.spin(15.degrees)
// res8: Color => Color.HSLA = <function1>
```

The third is

```
(image: Image) =>
  image.beside(image.rotate(90.degrees))
    .beside(image.rotate(180.degrees))
    .beside(image.rotate(270.degrees))
    .beside(image.rotate(360.degrees))
// res9: Image => Image = <function1>
```

Return to the exercise

B.8.2 Solution to: Function Types

The type is `Angle => Point`. This means `roseFn` is a function that takes a single argument of type `Angle` and returns a value of type `Point`. In other words, `roseFn` transforms an `Angle` to a `Point`.

Return to the exercise

B.8.3 Solution to: Spirals

Here's a type of spiral, known as a logarithmic spiral, that has a particularly pleasing shape. `sample` it and see for yourself!

```
def parametricSpiral(angle: Angle): Point =
  Point((Math.exp(angle.toTurns) - 1) * 200, angle)
```

Return to the exercise

B.8.4 Solution to: Sample

The answer is a small modification of the original `sample`. We drop the `dot` parameter and the type of the `curve` parameter changes. The rest follows from this.

```
def sample(samples: Int, curve: Angle => Image): Image = {
  val step = Angle.one / samples
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => Image.empty
      case n =>
        curve(angle).on(loop(n - 1))
    }
  }

  loop(samples)
```

```
}
```

Return to the exercise

B.8.5 Solution to: The Colour and the Shape

The simplest solution is to define three `singleShapes` as follows:

```
def concentricShapes(count: Int, singleShape: Int => Image): Image =
  count match {
    case 0 => Image.empty
    case n => singleShape(n).on(concentricShapes(n-1, singleShape))
  }

def rainbowCircle(n: Int) = {
  val color = Color.blue.desaturate(0.5.normalized).spin((n * 30).degrees)
  val shape = Image.circle(50 + n*12)
  shape.strokeWidth(10).strokeColor(color)
}

def fadingTriangle(n: Int) = {
  val color = Color.blue.fadeOut((1 - n / 20.0).normalized)
  val shape = Image.triangle(100 + n*24, 100 + n*24)
  shape.strokeWidth(10).strokeColor(color)
}

def rainbowSquare(n: Int) = {
  val color = Color.blue.desaturate(0.5.normalized).spin((n * 30).degrees)
  val shape = Image.rectangle(100 + n*24, 100 + n*24)
  shape.strokeWidth(10).strokeColor(color)
}

val answer =
  concentricShapes(10, rainbowCircle)
    .beside(
      concentricShapes(10, fadingTriangle)
        .beside(concentricShapes(10, rainbowSquare))
    )
```

However, there is some redundancy here: `rainbowCircle` and `rainbowTriangle`, in particular, use the same definition of `color`. There are also repeated calls to `strokeWidth(10)` and `strokeColor(color)` that can be eliminated. The extra credit solution factors these out into their own functions and combines them with the `colored` combinator:

```
def concentricShapes(count: Int, singleShape: Int => Image): Image =
  count match {
    case 0 => Image.empty
    case n => singleShape(n) on concentricShapes(n-1, singleShape)
  }

def colored(shape: Int => Image, color: Int => Color): Int => Image =
  (n: Int) =>
    shape(n).strokeWidth(10).strokeColor(color(n))

def fading(n: Int): Color =
  Color.blue.fadeOut((1 - n / 20.0).normalized)

def spinning(n: Int): Color =
```

```
  Color.blue.desaturate(0.5.normalized).spin((n * 30).degrees)

def size(n: Int): Double =
  100 + 24 * n

def circle(n: Int): Image =
  Image.circle(size(n))

def square(n: Int): Image =
  Image.square(size(n))

def triangle(n: Int): Image =
  Image.triangle(size(n), size(n))

val answer =
  concentricShapes(10, colored(circle, spinning))
    .beside(
      concentricShapes(10, colored(triangle, fading))
        .beside(concentricShapes(10, colored(square, spinning)))
    )
// answer: Image = Beside(
//   On(
//     StrokeColor(
//       StrokeWidth(Circle(340.0), 10.0),
//       HSLA(
//         Angle(9.42477796076938),
//         Normalized(0.5),
//         Normalized(0.5),
//         Normalized(1.0)
//       )
//     ),
//     On(
//       StrokeColor(
//         StrokeWidth(Circle(316.0), 10.0),
//         HSLA(
//           Angle(8.901179185171081),
//           Normalized(0.5),
//           Normalized(0.5),
//           Normalized(1.0)
//         )
//       ),
//       On(
//         StrokeColor(
//           StrokeWidth(Circle(292.0), 10.0),
//           HSLA(
//             Angle(8.377580409572781),
//             Normalized(0.5),
//             Normalized(0.5),
//             Normalized(1.0)
//           )
//         ),
//         On(
//           StrokeColor(
//             StrokeWidth(Circle(268.0), 10.0),
//             HSLA(
//               Angle(7.853981633974483),
//               Normalized(0.5),
//               Normalized(0.5),
//               Normalized(1.0)
//             )
```

```
//           ),
//           On(
//             StrokeColor(
//               StrokeWidth(Circle(244.0), 10.0),
//               HSLA(
//                 Angle(7.330382858376184),
//                 Normalized(0.5),
//                 Normalized(0.5),
//                 Normalized(1.0)
// ...
```

Return to the exercise

B.8.6 Solution to: Components

When we draw the parametric curves we probably what to change the radius of different curves. We could abstract this into a function. What should the type of this function be? Perhaps the most obvious approach is to have function with type `(Point, Double) => Point`, where the `Double` parameter is the amount by which we scale the point. This is somehwat annoying to use, however. We have to continually pass around the `Double`, which never varies from its initial setting.

A better structure is to create a function with type `Double => (Point => Point)`. This is a function to which we pass the scaling factor. It returns a function that transforms a `Point` by the given scaling factor. In this way we separate out the fixed scaling factor. The implementation could be

```
def scale(factor: Double): Point => Point =
  (pt: Point) => {
    Point.polar(pt.r * factor, pt.angle)
  }
```

In our previous discussion we've said we'd like to abstract the parametric equation out from `sample`. This we can easily do with

```
def sample(start: Angle, samples: Int, location: Angle => Point): Image = {
  // Angle.one is one complete turn. I.e. 360 degrees
  val step = Angle.one / samples
  val dot = Image.triangle(10, 10)
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => Image.empty
      case n =>
        dot.at(location(angle).toVec).on(loop(n - 1))
    }
  }

  loop(samples)
}
```

We might like to abstract out the choice of image primitive (`dot` or `Image.triangle` above). We could change `location` to be a function `Angle => Image` to accomplish this.

```
def sample(start: Angle, samples: Int, location: Angle => Image): Image = {
  // Angle.one is one complete turn. I.e. 360 degrees
  val step = Angle.one / samples
  def loop(count: Int): Image = {
    val angle = step * count
```

```
    count match {
      case 0 => Image.empty
      case n => location(angle).on(loop(n - 1))
    }
  }

  loop(samples)
}
```

We could also abstract out the entire problem specific part of the structural recursion. Where we had

```
def loop(count: Int): Image = {
  val angle = step * count
  count match {
    case 0 => Image.empty
    case n => location(angle).on(loop(n - 1))
  }
}
```

we could abstract out the base case (`Image.empty`) and the problem specific part on the recursion (`location(angle) on loop(n - 1)`). The former would be just an `Image` but the latter is a function with type `(Angle, Image) => Image`. The final result is

```
def sample(start: Angle, samples: Int, empty: Image, combine: (Angle, Image) => Image): Image = {
  // Angle.one is one complete turn. I.e. 360 degrees
  val step = Angle.one / samples
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => empty
      case n => combine(angle, loop(n - 1))
    }
  }

  loop(samples)
}
```

This is a very abstract function. We don't expect most people will see this abstraction, but if you're interested in exploring this idea more you might like to read about folds and monoids.

Return to the exercise

B.8.7 Solution to: Combine

You might end up with something like.

```
def parametricCircle(angle: Angle): Point =
  Point.cartesian(angle.cos, angle.sin)

def rose(angle: Angle): Point =
  Point.cartesian((angle * 7).cos * angle.cos, (angle * 7).cos * angle.sin)

def scale(factor: Double): Point => Point =
  (pt: Point) => {
    Point.polar(pt.r * factor, pt.angle)
  }

def sample(start: Angle, samples: Int, location: Angle => Point): Image = {
```

```
  // Angle.one is one complete turn. I.e. 360 degrees
  val step = Angle.one / samples
  val dot = Image.triangle(10, 10)
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => Image.empty
      case n => dot.at(location(angle).toVec) on loop(n - 1)
    }
  }

  loop(samples)
}

def locate(scale: Point => Point, point: Angle => Point): Angle => Point =
  (angle: Angle) => scale(point(angle))

// Rose on circle
val flower = {
  sample(0.degrees, 200, locate(scale(200), rose _)) on
  sample(0.degrees, 40, locate(scale(150), parametricCircle _))
}
```

Return to the exercise

B.8.8 Solution to: Experiment

Our implementation used to create fig. 9.1 is at Flowers.scala. What did you come up with? Let us know! Our email addresses are `noel@underscore.io` and `dave@underscore.io`.

Return to the exercise

B.9 Shapes, Sequences, and Stars

B.9.1 Solution to: Polygons

Using polar coordinates makes it much simpler to define the location of the "corners" (vertices) of the polygons. Each vertex is located a fixed rotation from the previous vertex, and after we've marked all vertices we must have done a full rotation of the circle. This means, for example, that for a pentagon each vertex is (360 / 5) = 72 degrees from the previous one. If we start at 0 degrees, vertices are located at 0, 72, 144, 216, and 288 degrees. The distance from the origin is fixed in each case. We don't have to draw a line between the final vertex and the start—by using a closed path this will be done for us.

Here's our code to draw fig. 10.2, which uses this idea. In some cases we haven't started the vertices at 0 degrees so we can rotate the shape we draw.

```
import doodle.core.PathElement._
import doodle.core.Point._
import doodle.core.Color._
val triangle =
  Image.closedPath(List(
                     moveTo(polar(50, 0.degrees)),
                     lineTo(polar(50, 120.degrees)),
                     lineTo(polar(50, 240.degrees))
                   ))
```

```
val square =
  Image.closedPath(List(
                     moveTo(polar(50, 45.degrees)),
                     lineTo(polar(50, 135.degrees)),
                     lineTo(polar(50, 225.degrees)),
                     lineTo(polar(50, 315.degrees))
                   ))

val pentagon =
  Image.closedPath(List(
                     moveTo(polar(50, 72.degrees)),
                     lineTo(polar(50, 144.degrees)),
                     lineTo(polar(50, 216.degrees)),
                     lineTo(polar(50, 288.degrees)),
                     lineTo(polar(50, 360.degrees))
                   ))

val spacer =
  Image.rectangle(10, 100).noStroke.noFill

def style(image: Image): Image =
  image.strokeWidth(6.0).strokeColor(paleTurquoise).fillColor(turquoise)

val image =
  style(triangle).beside(spacer).beside(style(square)).beside(spacer).beside(style(pentagon))
```

Return to the exercise

B.9.2 Solution to: Curves

The core of the exercise is to replace the `lineTo` expressions with `curveTo`. We can generalise curve creation into a method that takes the starting angle and the angle increment, and constructs control points at predetermined points along the rotation. This is what we did in the method `curve` below, and it gives us consistent looking curves without having to manually repeat the calculations each time. Making this generalisation also makes it easier to play around with different control points to create different outcomes.

```
import doodle.core.Point._
import doodle.core.PathElement._
import doodle.core.Color._

def curve(radius: Int, start: Angle, increment: Angle): PathElement = {
  curveTo(
    polar(radius *  .8, start + (increment * .3)),
    polar(radius * 1.2, start + (increment * .6)),
    polar(radius, start + increment)
  )
}

val triangle =
  Image.closedPath(List(
                     moveTo(polar(50, 0.degrees)),
                     curve(50, 0.degrees, 120.degrees),
                     curve(50, 120.degrees, 120.degrees),
                     curve(50, 240.degrees, 120.degrees)
                   ))

val square =
```

```
  Image.closedPath(List(
                     moveTo(polar(50, 45.degrees)),
                     curve(50, 45.degrees, 90.degrees),
                     curve(50, 135.degrees, 90.degrees),
                     curve(50, 225.degrees, 90.degrees),
                     curve(50, 315.degrees, 90.degrees)
                   ))

val pentagon =
  Image.closedPath((List(
                      moveTo(polar(50, 72.degrees)),
                      curve(50, 72.degrees, 72.degrees),
                      curve(50, 144.degrees, 72.degrees),
                      curve(50, 216.degrees, 72.degrees),
                      curve(50, 288.degrees, 72.degrees),
                      curve(50, 360.degrees, 72.degrees)
                    )))

val spacer =
  Image.rectangle(10, 100).noStroke.noFill

def style(image: Image): Image =
  image.strokeWidth(6.0).strokeColor(paleTurquoise).fillColor(turquoise)

val image = style(triangle).beside(spacer).beside(style(square)).beside(spacer).beside(style(
    pentagon))
```

Return to the exercise

B.9.3 Solution to: Building Lists

It's structural recursion over the natural numbers!

```
def ones(n: Int): List[Int] =
  n match {
    case 0 => Nil
    case n => 1 :: ones(n - 1)
  }

ones(3)
// res7: List[Int] = List(1, 1, 1)
```

Return to the exercise

B.9.4 Solution to: Building Lists Part 2

Once more, we can employ structural recursion over the natural numbers.

```
def descending(n: Int): List[Int] =
  n match {
    case 0 => Nil
    case n => n :: descending(n - 1)
  }

descending(0)
// res11: List[Int] = List()
descending(3)
```

```
// res12: List[Int] = List(3, 2, 1)
```

Return to the exercise

B.9.5 Solution to: Building Lists Part 3

It's structural recursion over the natural numbers, but this time with an internal accumulator.

```
def ascending(n: Int): List[Int] = {
  def iter(n: Int, counter: Int): List[Int] =
    n match {
      case 0 => Nil
      case n => counter :: iter(n - 1, counter + 1)
    }

  iter(n, 1)
}

ascending(0)
// res16: List[Int] = List()
ascending(3)
// res17: List[Int] = List(1, 2, 3)
```

Return to the exercise

B.9.6 Solution to: Building Lists Part 4

In this exercise we're asking you to use a type variable. Otherwise it is the same pattern as before.

```
def fill[A](n: Int, a: A): List[A] =
  n match {
    case 0 => Nil
    case n => a :: fill(n-1, a)
  }

fill(3, "Hi")
// res21: List[String] = List("Hi", "Hi", "Hi")
fill(3, Color.blue)
// res22: List[Color] = List(
//   RGBA(
//     UnsignedByte(-128),
//     UnsignedByte(-128),
//     UnsignedByte(127),
//     Normalized(1.0)
//   ),
//   RGBA(
//     UnsignedByte(-128),
//     UnsignedByte(-128),
//     UnsignedByte(127),
//     Normalized(1.0)
//   ),
//   RGBA(
//     UnsignedByte(-128),
//     UnsignedByte(-128),
//     UnsignedByte(127),
//     Normalized(1.0)
//   )
```

```
// )
```

Return to the exercise

B.9.7 Solution to: Transforming Lists

This is a structural recursion over a list, building a list at each step. The destructuring of the input is mirrored by the construction of the output.

```
def double(list: List[Int]): List[Int] =
  list match {
    case Nil => Nil
    case hd :: tl => (hd * 2) :: double(tl)
  }

double(List(1, 2, 3))
// res26: List[Int] = List(2, 4, 6)
double(List(4, 9, 16))
// res27: List[Int] = List(8, 18, 32)
```

Return to the exercise

B.9.8 Solution to: Transforming Lists Part 2

This is a structural recursion over a list using the same pattern as `sum` in the examples above.

```
def product(list: List[Int]): Int =
  list match {
    case Nil => 1
    case hd :: tl => hd * product(tl)
  }

product(Nil)
// res31: Int = 1
product(List(1,2,3))
// res32: Int = 6
```

Return to the exercise

B.9.9 Solution to: Transforming Lists Part 3

Same pattern as before, but using a type variable to allow type of the elements to vary.

```
def contains[A](list: List[A], elt: A): Boolean =
  list match {
    case Nil => false
    case hd :: tl => (hd == elt) || contains(tl, elt)
  }

contains(List(1,2,3), 3)
// res36: Boolean = true
contains(List("one", "two", "three"), "four")
// res37: Boolean = false
```

Return to the exercise

B.9.10 Solution to: Transforming Lists Part 4

This method is similar to `contains` above, except we now use the type variable in the return type as well as in the parameter types.

```
def first[A](list: List[A], elt: A): A =
  list match {
    case Nil => elt
    case hd :: tl => hd
  }

first(Nil, 4)
// res41: Int = 4
first(List(1,2,3), 4)
// res42: Int = 1
```

Return to the exercise

B.9.11 Solution to: Challenge Exercise: Reverse

The trick here is to use an accumulator to hold the partially reversed list. If you managed to work this one out, congratulations—you really understand structural recursion well!

```
def reverse[A](list: List[A]): List[A] = {
  def iter(list: List[A], reversed: List[A]): List[A] =
    list match {
      case Nil => reversed
      case hd :: tl => iter(tl, hd :: reversed)
    }

  iter(list, Nil)
}

reverse(List(1, 2, 3))
// res46: List[Int] = List(3, 2, 1)
reverse(List("a", "b", "c"))
// res47: List[String] = List("c", "b", "a")
```

Return to the exercise

B.9.12 Solution to: Polygons!

Here's our code. Note how we factored the code into small components—though we could have taken the factoring further is we wanted to. (Can you see how? Hint: do we need to pass, say, `start` to every call of `makeColor` when it's not changing?)

```
import Point._
import PathElement._

def polygon(sides: Int, size: Int, initialRotation: Angle): Image = {
  def iter(n: Int, rotation: Angle): List[PathElement] =
    n match {
```

```
      case 0 =>
        Nil
      case n =>
        LineTo(polar(size, rotation * n + initialRotation)) :: iter(n - 1, rotation)
    }
  Image.closedPath(moveTo(polar(size, initialRotation)) :: iter(sides, 360.degrees / sides))
}

def style(img: Image): Image = {
  img.
    strokeWidth(3.0).
    strokeColor(Color.mediumVioletRed).
    fillColor(Color.paleVioletRed.fadeOut(0.5.normalized))
}

def makeShape(n: Int, increment: Int): Image =
    polygon(n+2, n * increment, 0.degrees)

def makeColor(n: Int, spin: Angle, start: Color): Color =
  start.spin(spin * n)

val baseColor = Color.hsl(0.degrees, 0.7, 0.7)

def makeImage(n: Int): Image = {
  n match {
    case 0 =>
      Image.empty
    case n =>
      val shape = makeShape(n, 10)
      val color = makeColor(n, 30.degrees, baseColor)
      makeImage(n-1).on(shape.fillColor(color))
  }
}

val image = makeImage(15)
```

Return to the exercise

B.9.13 Solution to: Ranges, Lists, and map

We can just map over a Range to achieve this.

```
def ones(n: Int): List[Int] =
  (0 until n).toList.map(x => 1)

ones(3)
// res20: List[Int] = List(1, 1, 1)
```

Return to the exercise

B.9.14 Solution to: Ranges, Lists, and map Part 2

We can use a Range but we have to set the step size or the range will be empty.

```
def descending(n: Int): List[Int] =
  (n until 0 by -1).toList

descending(0)
// res24: List[Int] = List()
descending(3)
// res25: List[Int] = List(3, 2, 1)
```

Return to the exercise

B.9.15 Solution to: Ranges, Lists, and map Part 3

Again we can use a Range but we need to take care to start at `0` and increment the elements by `1` so we have the correct number of elements.

```
def ascending(n: Int): List[Int] =
  (0 until n).toList.map(x => x + 1)

ascending(0)
// res29: List[Int] = List()
ascending(3)
// res30: List[Int] = List(1, 2, 3)
```

Return to the exercise

B.9.16 Solution to: Ranges, Lists, and map Part 4

This is a straightforward application of `map`.

```
def double(list: List[Int]): List[Int] =
  list map (x => x * 2)

double(List(1, 2, 3))
// res34: List[Int] = List(2, 4, 6)
double(List(4, 9, 16))
// res35: List[Int] = List(8, 18, 32)
```

Return to the exercise

B.9.17 Solution to: Polygons, Again!

Here's one possible implementation. Much easier to read than the previous implementation!

```
def polygon(sides: Int, size: Int, initialRotation: Angle): Image = {
  import Point._
  import PathElement._

  val step = (Angle.one / sides).toDegrees.toInt
  val path =
    (0 to 360 by step).toList.map{ deg =>
      lineTo(polar(size, initialRotation + deg.degrees))
    }
```

```
  Image.closedPath(moveTo(polar(size, initialRotation)) :: path)
}
```

Return to the exercise

B.9.18 Solution to: Challenge Exercise: Beyond Map

We've seen many examples that we cannot implement with `map`: the methods `product`, `sum`, `find`, and more in the previous section cannot be implemented with `map`.

In abstract, methods implemented with map obey the following equation:

```
List[A] map A => B = List[B]
```

If the result is not of type `List[B]` we cannot implement it with `map`. For example, methods like `product` and `sum` transform `List[Int]` to `Int` and so cannot be implemented with `map`.

`Map` transforms the elements of a list, but cannot change the number of elements in the result. Even if a method fits the equation for `map` above it cannot be implemented with `map` if it requires changing the number of elements in the list.

Return to the exercise

B.9.19 Solution to: Using Open Intervals

Now that we now about `to` this is trivial to implement.

```
def ascending(n: Int): List[Int] =
  (1 to n).toList

ascending(0)
// res42: List[Int] = List()
ascending(3)
// res43: List[Int] = List(1, 2, 3)
```

Return to the exercise

B.9.20 Solution to: My God, It's Full of Stars!

Here's the `star` method. We've renamed `p` and `n` to `points` and `skip` for clarity:

```
def star(sides: Int, skip: Int, radius: Double): Image = {
  import Point._
  import PathElement._

  val rotation = 360.degrees * skip / sides

  val start = moveTo(polar(radius, 0.degrees))
  val elements = (1 until sides).toList map { index =>
    val point = polar(radius, rotation * index)
    lineTo(point)
  }
```

```
  Image.closedPath(start :: elements) strokeWidth 2
}
```

Return to the exercise

B.9.21 Solution to: My God, It's Full of Stars! Part 2

We can use the structural recursion skeleton to write this method.

We start with

```
def allBeside(images: List[Image]): Image =
  images match {
    case Nil => ???
    case hd :: tl => ???
  }
```

Remembering the recursion gives us

```
def allBeside(images: List[Image]): Image =
  images match {
    case Nil => ???
    case hd :: tl => /* something here */ allBeside(tl)
  }
```

Finally we can fill in the base and recursive cases.

```
def allBeside(images: List[Image]): Image =
  images match {
    case Nil => Image.empty
    case hd :: tl => hd.beside(allBeside(tl))
  }
```

Return to the exercise

B.9.22 Solution to: My God, It's Full of Stars! Part 3

To create the image in fig. **??** we started by creating a method to style a star.

```
def style(img: Image, hue: Angle): Image = {
  img.
    strokeColor(Color.hsl(hue, 1.0, 0.25)).
    fillColor(Color.hsl(hue, 1.0, 0.75))
}
```

We then created `allAbove`, which you will notice is very similar to `allBeside` (wouldn't it be nice if we could abstract this pattern?)

```
def allAbove(imgs: List[Image]): Image =
  imgs match {
    case Nil => Image.empty
    case hd :: tl => hd above allAbove(tl)
  }
```

The updated scene then becomes:

```
allAbove((3 to 33 by 2).toList map { sides =>
  allBeside((1 to sides/2).toList map { skip =>
    style(star(sides, skip, 20), 360.degrees * skip / sides)
  })
})
```

Return to the exercise

B.10 Turtle Algebra and Algebraic Data Types

B.10.1 Solution to: Polygons

Here's our solution. It's a structural recursion over the natural numbers. The turn angle is exactly the same as the rotation angle used to create polygons in polar coordinates in the previous chapter, though the derivation is quite different.

```
def polygon(sides: Int, sideLength: Double): Image = {
  val rotation = Angle.one / sides
  def iter(n: Int): List[Instruction] =
    n match {
      case 0 => Nil
      case n => turn(rotation) :: forward(sideLength) :: iter(n-1)
    }

  Turtle.draw(iter(sides))
}
```

Return to the exercise

B.10.2 Solution to: The Square Spiral

The key insights to draw the square spiral are realising:

- each turn is a little bit less than 90 degrees
- each step forward is a little bit longer than the last one

Once we have this understood this, the structure is basically the same as drawing a polyon. Here's our solution.

```
def squareSpiral(steps: Int, distance: Double, angle: Angle, increment: Double): Image = {
  def iter(n: Int, distance: Double): List[Instruction] = {
    n match {
      case 0 => Nil
      case n => forward(distance) :: turn(angle) :: iter(steps-1, distance + increment)
    }
  }

  Turtle.draw(iter(steps, distance))
}
```

Return to the exercise

B.10.3 Solution to: Turtles vs Polar Coordinates

Each side of the polygon requires two turtle instructions: a `forward` and a `turn`. Thus drawing a pentagon requires ten instructions, and in general n sides requires 2n instructions. Using `map` we cannot change the number of elements in a list. Therefore mapping `1 to n`, as we did int the code above, won't work. We could map over `1 to (n*2)`, and on, say, odd numbers move forward and on even numbers turn, but this is rather inelegant. It seems it would be simpler if we had an abstraction like `map` that allowed us to change the number of elements in the list as well as transform the individual elements.

Return to the exercise

B.10.4 Solution to: Branching Structures

In this case `map` is not the right solution, as the types tell us. Remember the type equation for `map` is

```
List[A] map (A => B) = List[B]
```

If - we have `List[Instruction]`; and - we `map` a function `Instruction => List[Instruction]`; then - we'll get a `List[List[Instruction]]`

as we can see from the type equation.

Our turtle doesn't know how to draw `List[List[Instruction]]` so this won't work.

Return to the exercise

B.10.5 Solution to: Double

There are two points to this:

- recognising how to use `flatMap`; and
- remembering how to use type variables.

```
def double[A](in: List[A]): List[A] =
  in.flatMap { x => List(x, x) }
```

Return to the exercise

B.10.6 Solution to: Or Nothing

We could easily write this method as

```
def nothing[A](in: List[A]): List[A] =
  List() // or List.empty or Nil
```

but the point of this exercise is to get more familiarity with using `flatMap`. With `flatMap` we can write the method as

```
def nothing[A](in: List[A]): List[A] =
  in.flatMap { x => List.empty }
```

Return to the exercise

B.10.7 Solution to: Rewriting the Rules

There are two parts to this:

- recognising that we need to use `flatMap`, for reasons discussed above; and
- realising that we need to recursively call `rewrite` to process the contents of a branch.

The latter is an example of structural recursion, though a slighlty more complex pattern than we've seen before.

```
def rewrite(instructions: List[Instruction], rule: Instruction => List[Instruction]): List[
    Instruction] =
  instructions.flatMap { i =>
    i match {
      case Branch(i) =>
        List(branch(rewrite(i, rule):_*))
      case other =>
        rule(other)
    }
  }
```

Return to the exercise

B.10.8 Solution to: Your Own L-System

This is just a simple structural recursion of the natural numbers, with all the hard work done by `rewrite`.

```
def iterate(steps: Int,
            seed: List[Instruction],
            rule: Instruction => List[Instruction]): List[Instruction] =
  steps match {
    case 0 => seed
    case n => iterate(n - 1, rewrite(seed, rule), rule)
  }
```

Return to the exercise

B.10.9 Solution to: Flat Polygon

Using `flatMap` we can make the code more compact than the explicit structural recursion we had to use before.

```
def polygon(sides: Int, sideLength: Double): Image = {
  val rotation = Angle.one / sides

  Turtle.draw((1 to sides).toList.flatMap { n =>
    List(turn(rotation), forward(sideLength))
  })
}
```

Return to the exercise

B.10.10 Solution to: Flat Spiral

Again, the result is more compact than the previous implementation without `flatMap`. Isthis easier to read? I find it about the same. I belive comprehensibility is a function of familiarity, and we're (hopefully) by now becoming familiar with `flatMap`.

```
def squareSpiral(steps: Int, distance: Double, angle: Angle, increment: Double): Image = {
  Turtle.draw((1 to steps).toList.flatMap { n =>
   List(forward(distance + (n * increment)), turn(angle))
  })
}
```

Return to the exercise

B.11 Composition of Generative Art

B.11.1 Solution to: Generative Art

Generating random numbers in this way breaks substitution. Remember substitution says wherever we see an expression we should be able to substitute the value it evaluates to without changing the meaning of the program. Concretely, this means

```
val result1 = randomAngle
// result1: Angle = Angle(3.1284665815466974)
val result2 = randomAngle
// result2: Angle = Angle(1.4307100761881815)
```

and

```
val result1 = randomAngle
// result1: Angle = Angle(2.772262271097978)
val result2 = result1
// result2: Angle = Angle(2.772262271097978)
```

should be the same program and clearly they are not.

Return to the exercise

B.11.2 Solution to: Randomness and Randomness

`programOne` displays three different circles in a row, while `programTwo` repeats the same circle three times. The value `circles` represents a program that generates an image of randomly colored concentric circles. Remember `map` represents a deterministic transform, so the output of `programTwo` must be the same same circle repeated thrice as we're not introducing new random choices. In `programOne` we merge `circle` with itself three times. You might think that the output should be only one random image repeated three times, not three, but remember `Random` preserves substitution. We can write `programOne` equivalently as

```
val programOneRewritten =
  randomConcentricCircles(5, 10) flatMap { c1 =>
    randomConcentricCircles(5, 10) flatMap { c2 =>
      randomConcentricCircles(5, 10) map { c3 =>
        c1 beside c2 beside c3
      }
    }
```

```
  }
// programOneRewritten: cats.free.Free[RandomOp, Image] = FlatMapped(
//   FlatMapped(
//     FlatMapped(
//       FlatMapped(
//         FlatMapped(
//           Suspend(RDouble),
//           cats.free.Free$$Lambda$9668/2041403374@24f4d3ce
//         ),
//         cats.free.Free$$Lambda$9668/2041403374@1c1e1241
//       ),
//       cats.free.Free$$Lambda$9668/2041403374@1f52079a
//     ),
//     <function1>
//   ),
//   <function1>
// )
```

which makes it clearer that we're generating three different circles.

Return to the exercise

B.11.3 Solution to: Colored Boxes

This code uses exactly the same pattern as `randomConcentricCircles`.

```
val randomAngle: Random[Angle] =
  Random.double.map(x => x.turns)
// randomAngle: Random[Angle] = FlatMapped(
//   Suspend(RDouble),
//   cats.free.Free$$Lambda$9668/2041403374@16228f6f
// )

val randomColor: Random[Color] =
  randomAngle.map(hue => Color.hsl(hue, 0.7, 0.7))
// randomColor: Random[Color] = FlatMapped(
//   FlatMapped(Suspend(RDouble), cats.free.Free$$Lambda$9668/2041403374@16228f6f),
//   cats.free.Free$$Lambda$9668/2041403374@14ca54c6
// )

def coloredRectangle(color: Color): Image =
  Image.rectangle(20, 20).fillColor(color)

def randomColorBoxes(count: Int): Random[Image] =
  count match {
    case 0 => randomColor.map{ c => coloredRectangle(c) }
    case n =>
      val box = randomColor.map{ c => coloredRectangle(c) }
      val boxes = randomColorBoxes(n-1)
      box.flatMap{ b =>
        boxes.map{ bs => b.beside(bs) }
      }
  }
```

Return to the exercise

B.11.4 Solution to: Particle Systems

This will do. You can create a more complicated (and interesting) distribution over starting position if you want.

```
val start = Random.always(Point.zero)
```

Return to the exercise

B.11.5 Solution to: Particle Systems Part 2

I've chosen to use normally distributed noise that is the same in both directions. Changing the noise will change the shape of the result—it's worth playing around with different settings.

```
def step(current: Point): Random[Point] = {
  val drift = Point(current.x + 10, current.y)
  val noise =
    Random.normal(0.0, 5.0) flatMap { x =>
      Random.normal(0.0, 5.0) map { y =>
        Vec(x, y)
      }
    }

  noise.map(vec => drift + vec)
}
```

Return to the exercise

B.11.6 Solution to: Particle Systems Part 3

In my definition of `render` I've shown how we can use information from the point to modify the shape in an interesting way.

The definition of `walk` is a structural recursion over the natural numbers with an internal accumulator and the recursion going through `flatMap`.

```
def render(point: Point): Image = {
  val length = (point - Point.zero).length
  val sides = (length / 20).toInt + 3
  val hue = (length / 200).turns
  val color = Color.hsl(hue, 0.7, 0.5)
  Image
    .star(sides, 5, 3, 0.degrees)
    .noFill
    .strokeColor(color)
    .at(point.toVec)
}

def walk(steps: Int): Random[Image] = {
  def loop(count: Int, current: Point, image: Image): Random[Image] = {
    count match {
      case 0 => Random.always(image on render(current))
      case n =>
        val next = step(current)
        next.flatMap{ pt =>
          loop(count - 1, pt, image on render(current))
        }
```

```
    }
  }

  start.flatMap{ pt => loop(steps, pt, Image.empty) }
}
```

Return to the exercise

B.11.7 Solution to: Particle Systems Part 4

Once again we have a structural recursion over the natural numbers. Unlike `walk` the recursion goes through `map`, not `flatMap`. This is because `particleSystem` adds no new random choices.

```
def particleSystem(particles: Int, steps: Int): Random[Image] = {
  particles match {
    case 0 => Random.always(Image.empty)
    case n => walk(steps).flatMap{ img1 =>
      particleSystem(n-1, steps) map { img2 =>
        img1 on img2
      }
    }
  }
}
```

Return to the exercise

B.11.8 Solution to: Random Abstractions

We could make `walk` `start`, and `render` parameters to `particleSystem`, and make `start` and `render` parameters to `walk`.

Return to the exercise

B.11.9 Solution to: Random Abstractions Part 2

If we add parameters with the correct name and type the code changes required are minimal. This is like doing the opposite of substitution—lifting concrete representations out of our code and replacing them with method parameters.

```
def walk(
  steps: Int,
  start: Random[Point],
  render: Point => Image
): Random[Image] = {
  def loop(count: Int, current: Point, image: Image): Random[Image] = {
    count match {
      case 0 => Random.always(image on render(current))
      case n =>
        val next = step(current)
        next.flatMap{ pt =>
          loop(count - 1, pt, image on render(current))
        }
    }
  }
```

```
  start.flatMap{ pt => loop(steps, pt, Image.empty) }
}

def particleSystem(
  particles: Int,
  steps: Int,
  start: Random[Point],
  render: Point => Image,
  walk: (Int, Random[Point], Point => Image) => Random[Image]
): Random[Image] = {
  particles match {
    case 0 => Random.always(Image.empty)
    case n => walk(steps, start, render).flatMap{ img1 =>
      particleSystem(n-1, steps, start, render, walk).map{ img2 =>
        img1.on(img2)
      }
    }
  }
}
```

Return to the exercise

B.11.10 Solution to: Scatter Plots

This is a nice example of composition of Randoms.

```
def makePoint(x: Random[Double], y: Random[Double]): Random[Point] =
  for {
    theX <- x
    theY <- y
  } yield Point.cartesian(theX, theY)
```

Return to the exercise

B.11.11 Solution to: Scatter Plots Part 2

Something like the following should work.

```
val normal = Random.normal(50, 15)
val normal2D = makePoint(normal, normal)

val data = (1 to 1000).toList.map(_ => normal2D)
```

Return to the exercise

B.11.12 Solution to: Scatter Plots Part 3

We can convert a `Point` to an `Image` using a method `point` below. Note I've made each point on the scatterplot quite transparent—this makes it easier to see where a lot of points are grouped together.

```
def point(loc: Point): Image =
  Image.circle(2).fillColor(Color.cadetBlue.alpha(0.3.normalized)).noStroke.at(loc.toVec)
```

Converting between the lists is just a matter of calling `map` a few times.

```
val points = data.map(r => r.map(point _))
```

Return to the exercise

B.11.13 Solution to: Scatter Plots Part 4

You might recognise this pattern. It's what we used in `allOn` with the addition of `flatMap`, which is exactly what `randomConcentricCircles` (and many other examples) use.

```
def allOn(points: List[Random[Image]]): Random[Image] =
  points match {
    case Nil => Random.always(Image.empty)
    case img :: imgs =>
      for {
        i  <- img
        is <- allOn(imgs)
      } yield (i on is)
  }
```

Return to the exercise

B.11.14 Solution to: Scatter Plots Part 5

This is just calling methods and using values we've already defined.

```
val plot = allOn(points)
```

Return to the exercise

B.11.15 Solution to: Parametric Noise

Here's our solution. We've already seen very similar code in the scatter plot.

```
def perturb(point: Point): Random[Point] =
  for {
    x <- Random.normal(0, 10)
    y <- Random.normal(0, 10)
  } yield Point.cartesian(point.x + x, point.y + y)
```

Return to the exercise

B.11.16 Solution to: Parametric Noise Part 2

Writing this with andThen is nice and neat.

```
def perturbedRose(k: Int): Angle => Random[Point] =
  rose(k) andThen perturb
```

Return to the exercise

B.11.17 Solution to: Parametric Noise Part 3

Here's the code we used to create [#fig:generative:volcano]. It's quite a bit larger than code we've seen up to this point, but you should understand all the components this code is built from.

```
object ParametricNoise {
  def rose(k: Int): Angle => Point =
    (angle: Angle) => {
      Point.cartesian((angle * k).cos * angle.cos, (angle * k).cos * angle.sin)
    }

  def scale(factor: Double): Point => Point =
    (pt: Point) => {
      Point.polar(pt.r * factor, pt.angle)
    }

  def perturb(point: Point): Random[Point] =
    for {
      x <- Random.normal(0, 10)
      y <- Random.normal(0, 10)
    } yield Point.cartesian(point.x + x, point.y + y)

  def smoke(r: Normalized): Random[Image] = {
    val alpha = Random.normal(0.5, 0.1)
    val hue = Random.double.map(h => (h * 0.1).turns)
    val saturation = Random.double.map(s => s * 0.8)
    val lightness = Random.normal(0.4, 0.1)
    val color =
      for {
        h <- hue
        s <- saturation
        l <- lightness
        a <- alpha
      } yield Color.hsla(h, s, l, a)
    val c = Random.normal(5, 5) map (r => Image.circle(r))

    for {
      circle <- c
      line   <- color
    } yield circle.strokeColor(line).noFill
  }

  def point(
    position: Angle => Point,
    scale: Point => Point,
    perturb: Point => Random[Point],
    image: Normalized => Random[Image],
    rotation: Angle
  ): Angle => Random[Image] = {
    (angle: Angle) => {
      val pt = position(angle)
      val scaledPt = scale(pt)
      val perturbed = perturb(scaledPt)

      val r = pt.r.normalized
      val img = image(r)

      for {
        i  <- img
        pt <- perturbed
```

```
      } yield (i at pt.toVec.rotate(rotation))
    }
  }

  def iterate(step: Angle): (Angle => Random[Image]) => Random[Image] = {
    (point: Angle => Random[Image]) => {
      def iter(angle: Angle): Random[Image] = {
        if(angle > Angle.one)
          Random.always(Image.empty)
        else
          for {
            p  <- point(angle)
            ps <- iter(angle + step)
          } yield (p on ps)
      }

      iter(Angle.zero)
    }
  }

  val image: Random[Image] = {
    val pts =
      for(i <- 28 to 360 by 39) yield {
        iterate(1.degrees){
          point(
            rose(5),
            scale(i),
            perturb _,
            smoke _,
            i.degrees
          )
        }
      }
    val picture = pts.foldLeft(Random.always(Image.empty)){ (accum, img) =>
      for {
        a <- accum
        i <- img
      } yield (a on i)
    }
    val background = (Image.rectangle(650, 650).fillColor(Color.black))

    picture map { _ on background }
  }
}
```

Return to the exercise

B.12 Algebraic Data Types To Call Our Own

B.12.1 Solution to: Path Elements

A `PathElement` is a sum type, as it is: - a `MoveTo`; or - a `LineTo`; or - a `CurveTo`.

A `MoveTo` is a product type that holds a single point (where to move to).

A `LineTo` is a product type that holds a single point (the end point of the line).

A `CurveTo` is a product type that holds three points: two control points and the end point of the line.

Return to the exercise

B.12.2 Solution to: Totally Turtles

An `Instruction` is: - a `Forward`; or - a `Turn`; or - a `Branch`; or - a `NoOp`

Therefore `Instruction` is a sum type. `Forward`, `Turn`, and `Branch` are all product types.

A `Forward` holds a distance, which is a `Double`.

A `Turn` holds an angle, which is an `Angle`.

A `Branch` holds a `List[Instruction]`—therefore the `Instruction` type is defined in terms of itself, just like `List`.

A `NoOp` holds no data.

Return to the exercise

B.12.3 Solution to: Exercise

We can directly turn the textual description into code using the patterns above.

```
sealed abstract class Instruction extends Product with Serializable
final case class Forward(distance: Double) extends Instruction
final case class Turn(angle: Angle) extends Instruction
final case class Branch(instructions: List[Instruction]) extends Instruction
final case class NoOp() extends Instruction
```

Return to the exercise

B.12.4 Solution to: Build Your Own Turtle

This is a product type.

```
final case class TurtleState(at: Vec, heading: Angle)
```

Return to the exercise

B.12.5 Solution to: Build Your Own Turtle Part 2

The core pattern is a structural recursion but the details are a bit more intricate in this case than we've seen before. We need to both create the path elements and update the state.

```
def process(state: TurtleState, instruction: Instruction): (TurtleState, List[PathElement]) = {
  import PathElement._

  instruction match {
    case Forward(d) =>
      val nowAt = state.at + Vec.polar(d, state.heading)
      val element = lineTo(nowAt.toPoint)

      (state.copy(at = nowAt), List(element))
    case Turn(a) =>
      val nowHeading = state.heading + a

      (state.copy(heading = nowHeading), List())
    case Branch(i) =>
      // Ignoring for now
```

```
      (state, List())
    case NoOp() =>
      (state, List())
  }
}
```

Return to the exercise

B.12.6 Solution to: Build Your Own Turtle Part 3

```
def iterate(state: TurtleState, instructions: List[Instruction]): List[PathElement] =
  instructions match {
    case Nil =>
      Nil
    case i :: is =>
      val (newState, elements) = process(state, i)
      elements ++ iterate(newState, is)
  }
```

Return to the exercise

B.12.7 Solution to: Build Your Own Turtle Part 4

Here's the complete turtle.

```
object Turtle {
  import Image._

  def draw(instructions: List[Instruction]): Image = {
    def iterate(state: TurtleState, instructions: List[Instruction]): List[PathElement] =
      instructions match {
        case Nil =>
          Nil
        case i :: is =>
          val (newState, elements) = process(state, i)
          elements ++ iterate(newState, is)
      }

    def process(state: TurtleState, instruction: Instruction): (TurtleState, List[PathElement]) = {
      import PathElement._

      instruction match {
        case Forward(d) =>
          val nowAt = state.at + Vec.polar(d, state.heading)
          val element = lineTo(nowAt.toPoint)

          (state.copy(at = nowAt), List(element))
        case Turn(a) =>
          val nowHeading = state.heading + a

          (state.copy(heading = nowHeading), List())
        case Branch(is) =>
         val branchedElements = iterate(state, is)

         (state, moveTo(state.at.toPoint) :: branchedElements)
        case NoOp() =>
          (state, List())
```

```
      }
    }

    openPath(iterate(TurtleState(Vec.zero, Angle.zero), instructions))
  }
}
```

Return to the exercise