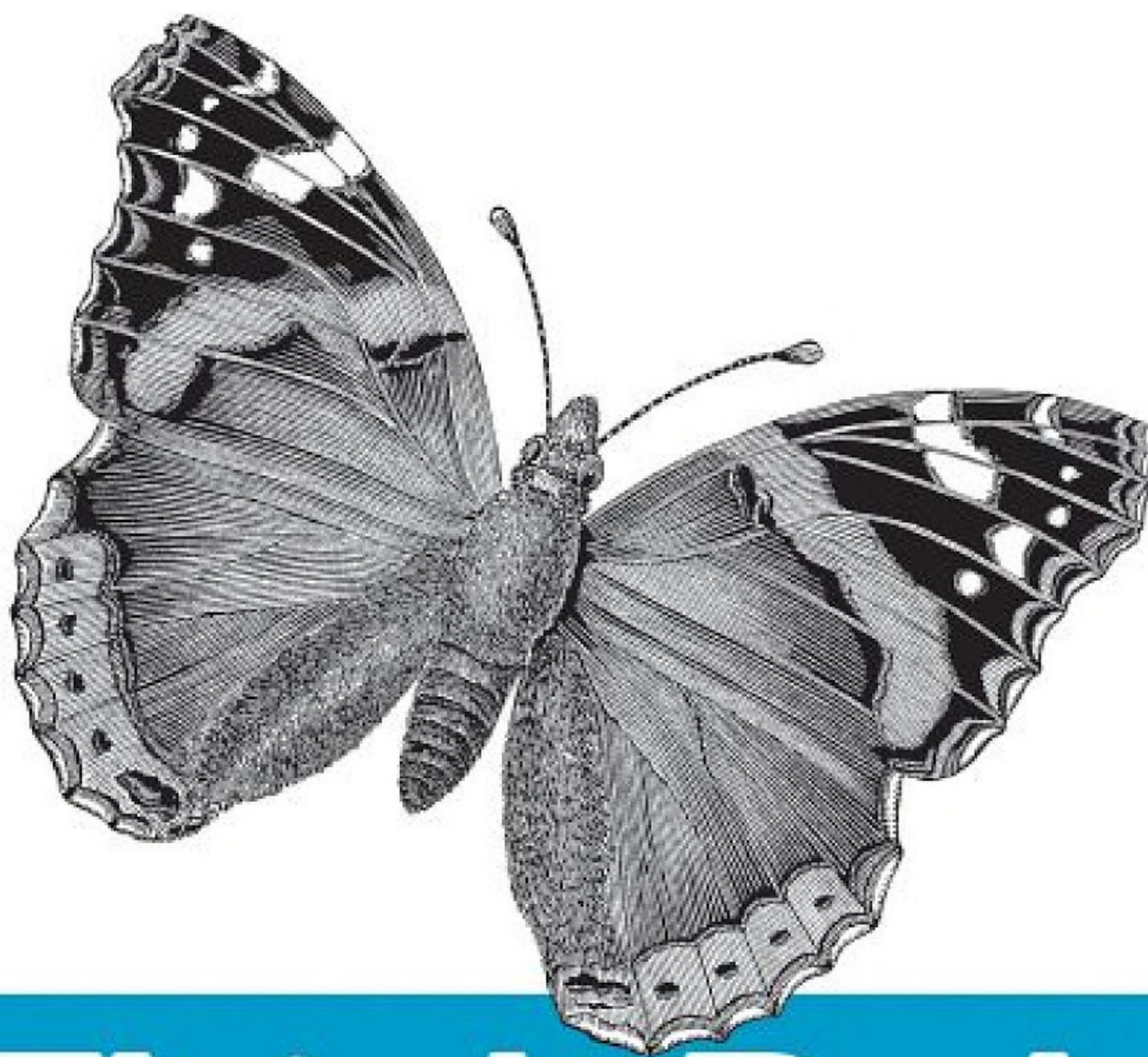


O'REILLY®



# Think Perl 6

HOW TO THINK LIKE A COMPUTER SCIENTIST

Laurent Rosenfeld  
with Allen B. Downey

## Think Perl 6

by Laurent Rosenfeld, with Allen B. Downey

Copyright © 2017 Allen Downey, Laurent Rosenfeld. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com/safari>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

Editors: Dawn Schanafelt and Brian Foster

Production Editor: Kristen Brown

Copyeditor: Charles Roumeliotis

Proofreader: Molly Ives Brower

Indexer: Laurent Rosenfeld and Allen B. Downey

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

May 2017: First Edition

## Revision History for the First Edition

- 2017-05-05: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491980552> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Think Perl 6*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-98055-2

[LSI]

## **Preface**

Welcome to the art of computer programming and to the new Perl 6 language. This will probably be the first published book using Perl 6 (or one of the first), a powerful, expressive, malleable, and highly extensible programming language. But this book is less about Perl 6, and more about learning how to write programs for computers.

This book is intended for beginners and does not require any prior programming knowledge, but it is my hope that even those of you with programming experience will benefit from reading it.

### **The Aim of This Book**

This aim of this book is not primarily to teach Perl 6, but instead to teach the art of programming, using the Perl 6 language. After having completed this book, you should hopefully be able to write programs to solve relatively difficult problems in Perl 6, but my main aim is to teach computer science, software programming, and problem solving rather than solely to teach the Perl 6 language itself.

This means that I will not cover every aspect of Perl 6, but only a (relatively large, but yet incomplete) subset of it. By no means is this book intended to be a reference on the language.

It is not possible to learn programming or to learn a new programming language by just reading a book; practicing is essential. This book contains a lot of exercises. You are strongly encouraged to make a real effort to do them. And, whether successful or not in solving the exercises, you should take a look at the solutions in the Appendix, as, very often, several solutions are suggested with further discussion on the subject and the issues involved. Sometimes, the solution section of the Appendix also introduces examples of topics that will be covered in the next chapter—and sometimes even things that are not covered elsewhere in the book. So, to get the most out of the book, I suggest you try to solve the exercises as well as review

the solutions and attempt them.

There are more than one thousand code examples in this book; study them, make sure to understand them, and run them. When possible, try to change them and see what happens. You're likely to learn a lot from this process.

## **The History of This Book**

In the course of the last three to four years, I have translated or adapted to French a number of tutorials and articles on Perl 6, and I've also written a few entirely new ones in French.<sup>1</sup> Together, these documents represented by the end of 2015 somewhere between 250 and 300 pages of material on Perl 6. By that time, I had probably made public more material on Perl 6 in French than all other authors taken together.

In late 2015, I began to feel that a Perl 6 document for beginners was something missing that I was willing to undertake. I looked around and found that it did not seem to exist in English either. I came to the idea that, after all, it might be more useful to write such a document initially in English, to give it a broader audience. I started contemplating writing a beginner introduction to Perl 6 programming. My idea at the time was something like a 50- to 70-page tutorial and I started to gather material and ideas in this direction.

Then, something happened that changed my plans.

In December 2015, friends of mine were contemplating translating into French Allen B. Downey's *Think Python, Second Edition*.<sup>2</sup> I had read an earlier edition of that book and fully supported the idea of translating it.<sup>3</sup> As it turned out, I ended up being a co-translator and the technical editor of the French translation of that book.<sup>4</sup>

While working on the French translation of Allen's Python book, the idea came to me that, rather than writing a tutorial on Perl 6, it might be more useful to make a "Perl 6 translation" of *Think Python*. Since I was in contact with Allen in the context of the French translation, I suggested this

to Allen, who warmly welcomed the idea. This is how I started to write this book late January 2016, just after having completed the work on the French translation of his Python book.

This book is thus largely derived on Allen's *Think Python*, but adapted to Perl 6. As it happened, it is also much more than just a "Perl 6 translation" of Allen's book: with quite a lot of new material, it has become a brand new book, largely indebted to Allen's book, but yet a new book for which I take all responsibility. Any errors are my own, not Allen's.

My hope is that this will be useful to the Perl 6 community, and more broadly to the open source and general computer programming communities. In an interview with *LinuxVoice* (July 2015), Larry Wall, the creator of Perl 6, said: "We do think that Perl 6 will be learnable as a first language." Hopefully this book will contribute to making this happen.

## **Conventions Used in This Book**

The following typographical conventions are used in this book:

### *Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

### Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

### **Constant width bold**

Shows commands or other text that should be typed literally by the user.

### *Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.

## Tip

This element signifies a tip or suggestion.

## Note

This element signifies a general note.

## Caution

This element indicates a warning or caution.

## Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://github.com/LaurentRosenfeld/thinkperl6/>.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Think Perl 6* by Laurent Rosenfeld with Allen B. Downey (O'Reilly). Copyright 2017 Allen Downey, Laurent Rosenfeld, 978-1-491-98055-2.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at [permissions@oreilly.com](mailto:permissions@oreilly.com).

## O'Reilly Safari

**Safari** (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit <http://oreilly.com/safari>.

## **How to Contact Us**

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://bit.ly/thinkPerl6>.

To comment or ask technical questions about this book, send email to



*bookquestions@oreilly.com.*

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

## **Acknowledgments**

I just don't know how I could thank Larry Wall to the level of gratitude he deserves for having created Perl in the first place, and Perl 6 more recently. Be blessed for eternity, Larry, for all of that.

And thank you to all of you who took part in this adventure (in no particular order), Tom, Damian, chromatic, Nathan, brian, Jan, Jarkko, John, Johan, Randall, Mark Jason, Ovid, Nick, Tim, Andy, Chip, Matt, Michael, Tatsuhiko, Dave, Rafael, Chris, Stevan, Saraty, Malcolm, Graham, Leon, Ricardo, Gurusamy, Scott, and too many others to name.

All my thanks also to those who believed in this Perl 6 project and made it happen, including those who quit at one point or another but contributed for some time; I know that this wasn't always easy.

Many thanks to Allen Downey, who very kindly supported my idea of adapting his book to Perl 6 and helped me in many respects, but also refrained from interfering into what I was putting in this new book.

I very warmly thank the people at O'Reilly who accepted the idea of this book and suggested many corrections or improvements. I want to thank especially Dawn Schanafelt, my editor at O'Reilly, whose advice has truly contributed to making this a better book. I also want to thank Kristen Brown for her helpful comments and work on publishing this book, and Charles Roumeliotis and Molly Ives Brower for their constructive review

and edits.

Thanks a lot in advance to readers who will offer comments or submit suggestions or corrections, as well as encouragement.

If you see anything that needs to be corrected or that could be improved, please kindly send your comments to [think.perl6@gmail.com](mailto:think.perl6@gmail.com).

## **Contributor List**

I would like to thank especially Moritz Lenz and Elizabeth Mattijsen, who reviewed in detail drafts of this book and suggested quite a number of improvements and corrections. Liz spent a lot of time on a detailed review of the full content of this book and I am especially grateful to her for her numerous and very useful comments. Thanks also to Timo Paulssen and ryanschoppe who also reviewed early drafts and provided some useful suggestions. Many thanks also to Uri Guttman, who reviewed this book and suggested a number of small corrections and improvements shortly before publication.

<sup>1</sup> See, for example, <http://perl.developpez.com/cours/#TutorielsPerl6>.

<sup>2</sup> See <http://greenteapress.com/wp/think-python-2e/>.

<sup>3</sup> I know, it's about Python, not Perl. But I don't believe in engaging in "language wars" and think that we all have to learn from other languages; to me, Perl's motto, "there is more than one way to do it," also means that doing it in Python (or some other language) is truly an acceptable possibility.

<sup>4</sup> See <http://allen-downey.developpez.com/livres/python/pensez-python/>.

## Part I. Starting with the Basics

This book has been divided into two parts. The main reason for that is that I wanted to make a distinction between, on the one hand, relatively basic notions that are really necessary for any programmer using Perl 6; and on the other hand, more advanced concepts that a good programmer needs to know but are possibly used less often in day-to-day development work.

The first eleven chapters (a bit more than 200 pages) that make up this first part are meant to teach the concepts that every programmer should know: variables, expressions, statements, functions, conditionals, recursion, operator precedence, and loops, as well as commonly used basic data structures and the most useful algorithms. These chapters can, I believe, be the basis for a one-semester introductory course on programming.

Of course, the professor or teacher who wishes to use this material is entirely free to skip some details from Part I (and also to include sections from Part II), but, at least, I have provided some guidelines on how I think this book could be used to teach programming with the Perl 6 language.

Part II focuses on different programming paradigms and more advanced programming techniques that are (in my opinion) of paramount importance, but should probably be studied in the context of a second, more advanced, semester.

For now, let's get down to the basics. It is my hope that you will enjoy the trip.

## Chapter 1. The Way of the Program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is *problem solving*. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, "The Way of the Program."

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

### What Is a Program?

A *program* is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document, or something graphical, like processing an image or playing a video.

The details look different in different languages, but a few basic instructions appear in just about every language:

#### Input

Get data from the keyboard, a file, the network, a sensor, a GPS chip, or some other device.

## Output

Display data on the screen, save it in a file, send it over the network, act on a mechanical device, etc.

## Math

Perform basic mathematical operations like addition and multiplication.

## Conditional execution

Check for certain conditions and run the appropriate code.

## Repetition

Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

Using or calling these subtasks makes it possible to create various levels of *abstraction*. You have probably been told that computers only use 0's and 1's at the lowest level; but we usually don't have to worry about that. When we use a word processor to write a letter or a report, we are interested in files containing text and some formatting instructions, and with commands to change the file or to print it; fortunately, we don't have to care about the underlying 0's and 1's; the word-processing program offers us a much higher view (files, commands, etc.) that hides the gory underlying details.

Similarly, when we write a program, we usually use and/or create several layers of abstraction, so that, for example, once we have created a subtask that queries a database and stores the relevant data in memory, we no

longer have to worry about the technical details of the subtask. We can use it as a sort of black box that will perform the desired operation for us. The essence of programming is to a large extent this art of creating these successive layers of abstraction so that performing the higher level tasks becomes relatively easy.

## Running Perl 6

One of the challenges of getting started with Perl 6 is that you might have to install Perl 6 and related software on your computer. If you are familiar with your operating system, and especially if you are comfortable with the shell or command-line interface, you will have no trouble installing Perl 6. But for beginners, it can be painful to learn about system administration and programming at the same time.

To avoid that problem, you can start out running Perl 6 in a web browser. You might want to use a search engine to find such a site. Currently, the easiest is probably to connect to the [glot.io site](http://glot.io), where you can type some Perl 6 code in the main window, run it, and see the result in the output window below.

Sooner or later, however, you will really need to install Perl 6 on your computer.

The easiest way to install Perl 6 on your system is to download Rakudo Star (a distribution of Perl 6 that contains the Rakudo Perl 6 *compiler*, documentation and useful modules): follow the instructions for your operating system at the [Rakudo](http://rakudo.org) and [Perl 6](http://perl6.org) websites.

As of this writing, the most recent specification of the language is Perl 6 version 6c (v6.c), and the most recent release available for download is Rakudo Star 2016.07;<sup>1</sup> the examples in this book should all run with this version. You can find out the installed version by issuing the following command at the operating system prompt:

```
$ perl6 -v
This is Rakudo version 2016.07.1 built on MoarVM version 2016.07
implementing Perl 6.c.
```

However, you should probably download and install the most recent version you can find. The output (warnings, error messages, etc.) you'll get from your version of Perl might in some cases slightly differ from what is printed in this book, but these possible differences should essentially be only cosmetic.

Compared to Perl 5, Perl 6 is not just a new version of Perl. It is more like a new little sister of Perl 5. It does not aim to replace Perl 5. Perl 6 is really a new programming language, with a syntax that is similar to earlier versions of Perl (such as Perl 5), but still markedly different. Unless stated otherwise, this book is about Perl 6 only, not about Perl 5 and preceding versions of the Perl programming language. From now on, whenever we speak about *Perl* with no further qualification, we mean Perl 6.

The Perl 6 *interpreter* is a program that reads and executes Perl 6 code. It is sometimes called REPL (for “read, evaluate, print, loop”). Depending on your environment, you might start the interpreter by clicking on an icon, or by typing `perl6` on a command line.

When it starts, you should see output like this:

```
To exit type 'exit' or '^D'
(Possibly some information about Perl and related software)
>
```

The last line with `>` is a *prompt* that indicates that the REPL is ready for you to enter code. If you type a line of code and hit Enter, the interpreter displays the result:

```
> 1 + 1
```

```
2  
>
```

You can type `exit` at the REPL prompt to exit the REPL.

Now you're ready to get started. From here on, we assume that you know how to start the Perl 6 REPL and run code.

## The First Program

Traditionally, the first program you write in a new language is called “Hello, World” because all it does is display the words “Hello, World”. In Perl 6, it looks like this:

```
> say "Hello, World";  
Hello, World  
>
```

This is an example of what is usually called a *print statement*, although it doesn't actually print anything on paper and doesn't even use the `print` keyword<sup>2</sup> (keywords are words which have a special meaning to the language and are used by the interpreter to recognize the structure of the program). The print statement displays a result on the screen. In this case, the result is the words `Hello, World`. The quotation marks in the program indicate the beginning and end of the text to be displayed; they don't appear in the result.

The semicolon (“;”) at the end of the line indicates that this is the end of the current statement. Although a semicolon is technically not needed when running simple code directly under the REPL, it is usually necessary when writing a program with several lines of code, so you might as well just get into the habit of ending code instructions with a semicolon.

Many other programming languages would require parentheses around the sentence to be displayed, but this is usually not necessary in Perl 6.



## Arithmetic Operators

After “Hello, World,” the next step is arithmetic. Perl 6 provides *operators*, which are special symbols that represent computations like addition and multiplication.

The operators `+`, `-`, `*`, and `/` perform addition, subtraction, multiplication, and division, as in the following examples under the REPL:

```
> 40 + 2
42
> 43 - 1
42
> 6 * 7
42
> 84 / 2
42
```

Since we use the REPL, we don’t need an explicit print statement in these examples, as the REPL automatically prints out the result of the statements for us. In a real program, you would need a print statement to display the result, as we’ll see later. Similarly, if you run Perl statements in the web browser mentioned in [“Running Perl 6”](#), you will need a print statement to display the result of these operations. For example:

```
say 40 + 2;    # -> 42
```

Finally, the operator `**` performs exponentiation; that is, it raises a number to a power:

```
> 6**2 + 6
42
```

In some other languages, the caret (“`^`”) or circumflex accent is used for

exponentiation, but in Perl 6 it is used for some other purposes.

## Values and Types

A *value* is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are 2, 42, and "Hello, World".

These values belong to different *types*: 2 is an *integer*,  $40 + 2$  is also an integer,  $84/2$  is a *rational number*, and "Hello, World" is a *string*, so called because the characters it contains are strung together.

If you are not sure what type a value has, Perl can tell you:

```
> say 42.WHAT;  
(Int)  
> say (40 + 2).WHAT;  
(Int)  
> say (84 / 2).WHAT;  
(Rat)  
> say (42.0).WHAT  
(Rat)  
> say ("Hello, World").WHAT;  
(Str)  
>
```

In these instructions, `.WHAT` is known as an *introspection method*; that is, a kind of method which will tell you *what* (of which type) the preceding expression is. `42.WHAT` is an example of the dot syntax used for method invocation: it calls the `.WHAT` built-in on the "42" expression (the *invocant*) and provides to the `say` function the result of this invocation, which in this case is the type of the expression.

Not surprisingly, integers belong to the type `Int`, strings belong to `Str`, and rational numbers belong to `Rat`.

Although  $40 + 2$  and  $84 / 2$  seem to yield the same result (42), the first expression returns an integer (`Int`) and the second a rational number (`Rat`).

The number 42.0 is also a rational.

The rational type is somewhat uncommon in most programming languages. Internally, these numbers are stored as two integers representing the numerator and the denominator (in their simplest terms). For example, the number 17.3 might be stored as two integers, 173 and 10, meaning that Perl is really storing something meaning the  $\frac{173}{10}$  fraction. Although this is usually not needed (except for introspection or debugging), you might access these two integers with the following methods:

```
> my $num = 17.3;
17.3
> say $num.WHAT;
(Rat)
> say $num.numerator, " ", $num.denominator; # say can print a
list
173 10
> say $num.nude; # "nude" stands for numerator-
denominator
(173 10)
```

This may seem anecdotal, but, for reasons which are beyond the scope of this book, this makes it possible for Perl 6 to perform arithmetical operations on rational numbers with a much higher accuracy than most common programming languages. For example, if you try to perform the arithmetical operation  $0.3 - 0.2 - 0.1$  with most general purpose programming languages (and depending on your machine architecture), you might obtain a result such as  $-2.77555756156289e-17$  (in Perl 5),  $-2.775558e-17$  (in C under GCC), or  $-2.7755575615628914e-17$  (Java, Python 3, Ruby, TCL). Don't worry about these values if you don't understand them; let us just say that they are extremely small but they are not 0, whereas, obviously, the result should really be zero. In Perl 6, the result is 0 (even to the fiftieth decimal digit):

```
> my $result-should-be-zero = 0.3 - 0.2 - 0.1;
0
> printf "%.50f", $result-should-be-zero; # prints 50 decimal
digits
0.000000000000000000000000000000000000000000000000000000000000000000
```

In Perl 6, you might even compare the result of the operation with 0:

```
> say $result-should-be-zero == 0;
True
```

Don't do such a comparison with most common programming languages; you're very likely to get a wrong result.

What about values like "2" and "42.0"? They look like numbers, but they are in quotation marks like strings.

```
> say '2'.perl; # perl returns a Perlsh representation of the
invocant
"2"
> say "2".WHAT;
(Str)
> say '42'.WHAT;
(Str)
```

They're strings because they are defined within quotes. Although Perl will often perform the necessary conversions for you, it is generally a good practice not to use quotation marks if your value is intended to be a number.

When you type a large integer, you might be tempted to use commas between groups of digits, as in 1,234,567. This is not a legal *integer* in Perl 6, but it is a legal expression:

```
> 1,234,567
```

```
(1 234 567)
>
```

That's actually a list of three different integer numbers, and not what we expected at all!

```
> say (1,234,567).WHAT
(List)
```

Perl 6 interprets `1,234,567` as a comma-separated sequence of three integers. As we will see later, the comma is a separator used for constructing lists.

You can, however, separate groups of digits with the underscore character (“\_”) for better legibility and obtain a proper integer:

```
> 1_234_567
1234567
> say 1_234_567.WHAT
(Int)
>
```

## Formal and Natural Languages

*Natural languages* are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

*Formal languages* are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

*Programming languages are formal languages that have been designed to express computations.*

Formal languages tend to have strict *syntax* rules that govern the structure of statements. For example, in mathematics the statement  $3 + 3 = 6$  has correct syntax, but not  $3 + = 3\$6$ . In chemistry  $H_2O$  is a syntactically correct formula, but  $_2Zz$  is not.

Syntax rules come in two flavors, pertaining to *tokens* and *structure*. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with  $3 + = 3\$6$  is that  $\$$  is not a legal token in mathematics (at least as far as I know). Similarly,  $_2Zz$  is not legal because there is no chemical element with the abbreviation  $Zz$ .

The second type of syntax rule, structure, pertains to the way tokens are combined. The equation  $3 + = 3$  is illegal in mathematics because even though  $+$  and  $=$  are legal tokens, you can't have one right after the other. Similarly, in a chemical formula, the subscript representing the number of atoms in a chemical compound comes after the element name, not before.

This is @ well-structured Engli\$h sentence with invalid t\*kens in it. This sentence all valid tokens has, but invalid structure with.

When you read a sentence in English or a statement in a formal language, you have to figure out the structure (although in a natural language you do this subconsciously). This process is called *parsing*.

Although formal and natural languages have many features in common—tokens, structure, and syntax—there are some differences:

### Ambiguity

Natural languages are full of ambiguity, which people deal with by

using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning.

## Redundancy

In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

## Literalness

Natural languages are full of idiom and metaphor. If we say, “The penny dropped,” there is probably no penny and nothing dropping (this idiom means that someone understood something after a period of confusion). Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. The difference between formal and natural language is like the difference between poetry and prose, but more so:

## Poetry

Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

## Prose

The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

## Programs

The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

## **Debugging**

Programmers make mistakes. Programming errors are called *bugs* and the process of tracking them down is called *debugging*.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

There is evidence that people naturally respond to computers as if they were people. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people<sup>3</sup>.

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a section, like this one, with our suggestions for debugging. I hope they help!



## **Glossary**

### abstraction

A way of providing a high-level view of a task and hiding the underlying technical details so that this task becomes easy.

### bug

An error in a program.

### compiler

A program that reads another program and transforms it into executable computer code; there used to be a strong difference between interpreted and compiled languages, but this distinction has become blurred over the last two decades or so.

### debugging

The process of finding and correcting bugs.

### formal language

Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

### integer

A type that represents whole numbers.

### interpreter

A program that reads another program and executes it.

### natural language

Any one of the languages that people speak that evolved naturally.

operator

A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

parse

To examine a program and analyze the syntactic structure.

print statement

An instruction that causes the Perl 6 interpreter to display a value on the screen.

problem solving

The process of formulating a problem, finding a solution, and expressing it.

program

A set of instructions that specifies a computation.

prompt

Characters displayed by the interpreter to indicate that it is ready to take input from the user.

rational

A type that represents numbers with fractional parts. Internally, Perl stores a rational as two integers representing respectively the numerator and the denominator of the fractional number.

string

A type that represents sequences of characters.

syntax

The rules that govern the structure of a program.

token

One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

type

A category of values. The types we have seen so far are integers (type `Int`), rational numbers (type `Rat`), and strings (type `Str`).

value

One of the basic units of data, like a number or string, that a program manipulates.

## **Exercises**

### **Exercise 1-1.**

It is a good idea to read this book in front of a computer so you can try out the examples as you go.

Whenever you are experimenting with a new feature, you should try to make mistakes. For example, in the “Hello, world!” program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell `say` wrong?

This kind of experiment helps you remember what you read; it also helps when you are programming, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.

Please note that most exercises in this book are provided with a solution in the appendix. However, the exercises in this chapter and in the next chapter are not intended to let you solve an actual problem but are designed to simply let you experiment with the Perl interpreter; there is no good solution, just try out what is proposed to get a feeling on how it works.

1. If you are trying to print a string, what happens if you leave out one of the quotation marks, or both?
2. You can use a minus sign to make a negative number like `-2`. What happens if you put a plus sign before a number? What about `2++2`?
3. In math notation, leading zeros are OK, as in `02`. What happens if you try this in Perl?
4. What happens if you have two values with no operator between them, such as `say 2 2;`?

## Exercise 1-2.

Start the Perl 6 REPL interpreter and use it as a calculator.

1. How many seconds are there in 42 minutes, 42 seconds?
2. How many miles are there in 10 kilometers? Hint: there are 1.61 kilometers in a mile.
3. If you run a 10-kilometer race in 42 minutes, 42 seconds, what is your average pace (time per mile in minutes and seconds)? What is your average speed in miles per hour?

<sup>1</sup> As we go to press, the latest version is 2017.01.

<sup>2</sup> Perl also has a `print` function, but the `say` built-in function is used here because it adds a newline character to the output.

<sup>3</sup> Byron Reeves and Clifford Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places* (Center for the Study of Language and Information, 2003)

## Chapter 2. Variables, Expressions, and Statements

One of the most powerful features of a programming language is the ability to manipulate *variables*. Broadly speaking, a variable is a name that refers to a value. It might be more accurate to say that a variable is a container that has a name and holds a value.

### Assignment Statements

An *assignment statement* uses the equals sign (=) and gives a value to a variable, but, before you can assign a value to a variable, you first need to create the variable by declaring it (if it does not already exist):

```
> my $message;           # variable declaration, no value yet
> $message = 'And now for something completely different';
And now for something completely different
> my $number = 42;      # variable declaration and assignment
42
> $number = 17;        # new assignment
17
> my $phi = 1.618033988;
1.618033988
>
```

This example makes four assignment statements. The first assigns a string to a new variable named `$message`, the second assigns the integer 42 to `$number`, the third reassigns the integer 17 to `$number`, and the fourth assigns the (approximate) value of the golden ratio to `$phi`.

There are two important syntax features to understand here.

First, in Perl, variable names start with a so-called *sigil*, i.e., a special non-alphanumeric character such as `$`, `@`, `%`, `&`, and some others. This special character tells us and the Perl compiler (the program that reads the code of our program and transforms it into computer instructions) which kind of variable it is. For example, the `$` character indicates that the variables above are all *scalar variables*, which means that they can contain only one value

at any given time. We'll see later other types of variables that may contain more than one value.

Second, notice that all three variables above are first introduced by the keyword `my`, which is a way of declaring a new variable. Whenever you create a new variable in Perl, you need to *declare* it, i.e., tell Perl that you're going to use that new variable; this is most commonly done with the `my` keyword, which declares a *lexical* variable. We will explain later what a lexical variable is; let's just say for the time being that it enables you to make your variable local to a limited part of your code. One of the good consequences of the requirement to declare variables before you use them is that, if you accidentally make a typo when writing a variable name, the compiler will usually be able to tell you that you are using a variable that has not been declared previously and thus help you find your error. This has other far-reaching implications, which we will examine later.

When we wrote at the beginning of this section that a variable has to be declared before it is used (or just when it is used), it plainly means that the declaration has to be before (or at the point of) the variable's first use in the text file containing the program. We will see later that programs don't necessarily run from top to bottom in the order in which the lines or code appear in the program file; still, the variable declaration must be before its use in the text file containing the program.

If you neglect to declare a variable, you get a syntax error:

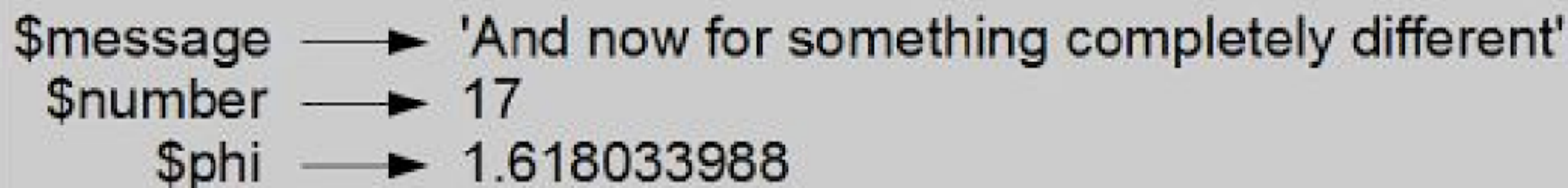
```
> $number = 5;
===SORRY!=== Error while compiling <unknown file>
Variable '$number' is not declared
at <unknown file>:1
-----> <BOL><HERE>$number = 5;
>
```

Please remember that you may obtain slightly different error messages

depending on the version of Rakudo you run. The above message was obtained in February 2016; with a newer version (October 2016), the same error is now displayed somewhat more cleanly as:

```
>  
> $number = 5;  
===SORRY!=== Error while compiling:  
Variable '$number' is not declared  
at line 2  
-----> <BOL><HERE>$number = 5;  
>
```

A common way to represent variables on paper is to write the name with an arrow pointing to its value. This kind of figure is called a *state diagram* because it shows what state each of the variables is in (think of it as the variable's state of mind). **Figure 2-1** shows the result of the previous example.



```
$message —> 'And now for something completely different'  
$number —> 17  
$phi —> 1.618033988
```

*Figure 2-1. State diagram*

## Variable Names

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

Variable names can be as long as you like. They can contain both letters and numbers, but user-defined variable names can't begin with a number. Variable names are case-sensitive, i.e., `$message` is not the same variable as `$Message` or `$MESSAGE`. It is legal to use uppercase letters, but it is conventional to use only lowercase for most variables names. Some people

nonetheless like to use `$TitleCase` for their variables or even pure `$UPPERCASE` for some special variables.

Unlike most other programming languages, Perl 6 does not require the letters and digits used in variable names to be plain ASCII. You can use all kinds of Unicode letters, i.e., letters from almost any language in the world, so that, for example, `$brücke`, `$payé`, or `$niño` are valid variable names, which can be useful for non-English programmers (provided that these Unicode characters are handled correctly by your text editor and your screen configuration). Similarly, instead of using `$phi` for the name of the golden ratio variable, we might have used the Greek small letter phi, (Unicode code point U+03C6), just as we could have used the Greek small letter pi, `π`, for the well-known circle circumference-to-diameter ratio:

```
> my $φ = (5 ** .5 + 1)/2;          # golden ratio
1.61803398874989
> say 'Variable $φ = ', $φ;
Variable $φ = 1.61803398874989
> my $π = 4 * atan 1;
3.14159265358979
> # you could also use the pi or n built-in constant:
> π
3.14159265358979
```

The underscore character, `_`, can appear anywhere in a variable name. It is often used in names with multiple words, such as `$your_name` or `$airspeed_of_unladen_swallow`.

You may even use dashes to create so-called “kebab case”<sup>1</sup> and name those variables `$your-name` or `$airspeed-of-unladen-swallow`, and this might make them slightly easier to read: a dash is valid in variable names provided it is immediately followed by an alphabetical character and preceded by an alphanumerical character. For example, `$double-click` or `$la-niña` are legitimate variable names. Similarly, you can use an



apostrophe ' (or single quote) between letters, so \$i'sn't or \$o'brien's-age are valid identifiers.

If you give a variable an illegal name, you get a syntax error:

```
> my $76trombones = 'big parade'
===SORRY!=== Error while compiling <unknown file>
Cannot declare a numeric variable
at <unknown file>:1
-----> my $76<HERE>trombones = "big parade";
>
> my $more§ = 100000;
===SORRY!=== Error while compiling <unknown file>
Bogus postfix
at <unknown file>:1
-----> my $more<HERE>§ = 100000;
(...)
```

\$76trombones is illegal because it begins with a number. \$more§ is illegal because it contains an illegal character, §.

If you've ever used another programming language and stumbled across a terse message such as "SyntaxError: invalid syntax", you will notice that the Perl designers have made quite a bit of effort to provide detailed, useful, and meaningful error messages.

Many programming languages have *keywords* or *reserved words* that are part of the syntax, such as `if`, `while`, or `for`, and thus cannot be used for identifying variables because this would create ambiguity. There is no such problem in Perl: since variable names start with a sigil, the compiler is always able to tell the difference between a keyword and a variable. Names such as `$if` or `$while` are syntactically valid variable identifiers in Perl (whether such names make sense is a different matter).

## **Expressions and Statements**

An *expression* is a combination of terms and operators. Terms may be

variables or literals, i.e., constant values such as a number or a string. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

```
> 42
42
> my $n = 17;
17
> $n;
17
> $n + 25;
42
>
```

When you type an expression at the prompt, the interpreter *evaluates* it, which means that it finds the value of the expression. In this example, `$n` has the value 17 and `$n + 25` has the value 42.

A *statement* is a unit of code that has an effect, like creating a variable or displaying a value, and usually needs to end with a semicolon `;` (but the semicolon can sometimes be omitted, as we will see later):

```
> my $n = 17;
17
> say $n;
17
```

The first line is an assignment statement that gives a value to `$n`. The second line is a print statement that displays the value of `$n`.

When you type a statement and then press `Enter`, the interpreter *executes* it, which means that it does whatever the statement says.

An assignment can be combined with expressions using arithmetic operators. For example, you might write:

```
> my $answer = 17 + 25;
42
> say $answer;
42
```

The `+` symbol is obviously the addition operator and, after the assignment statement, the `$answer` variable contains the result of the addition. The terms on each side of the operator (here 17 and 25) are sometimes called the *operands* of the operation (an addition in this case).

Note that the REPL actually displays the result of the assignment (the first line with “42”), so the print statement was not really necessary in this example *under the REPL*; from now on, for the sake of brevity, we will generally omit the print statements in the examples where the REPL displays the result.

In some cases, you want to add something to a variable and assign the result to that same variable. This could be written:

```
> my $answer = 17;
17
> $answer = $answer + 25;
42
```

Here, `$answer` is first declared with a value of 17. The next statement assigns to `$answer` the current value of `$answer` (i.e., 17) + 25. This is such a common operation that Perl, like many other programming languages, has a shortcut for this:

```
> my $answer = 17;
17
> $answer += 25;
42
```

The `+=` operator combines the arithmetic addition operator and the assignment operator to modify a value and apply the result to a variable in one go, so that `$n += 2` means take the current value of `$n`, add 2, and assign the result to `$n`. This syntax works with all other arithmetic operators. For example, `-=` similarly performs a subtraction and an assignment, `*=` a multiplication and an assignment, etc. It can even be used with operators other than arithmetic operators, such as the string concatenation operator that we will see later.

Adding 1 to a variable is a very common version of this, so that there is a shortcut to the shortcut, the *increment* operator, which increments its argument by one, and returns the incremented value:

```
> my $n = 17;
17
> ++$n;
18
> say $n;
18
```

This is called the prefix increment operator, because the `++` operator is placed before the variable to be incremented. There is also a postfix version, `$n++`, which first returns the current value and then increments the variable by one. It would not make a difference in the code snippet above, but the result can be very different in slightly more complex expressions.

There is also a decrement operator `--`, which decrements its argument by one and also exists in a prefix and a postfix form.

## **Script Mode**

So far we have run Perl in *interactive mode*, which means that you interact directly with the interpreter (the REPL). Interactive mode is a good way to get started, but if you are working with more than a few lines of code, it can be clumsy and even tedious.

The alternative is to use a text editor and save code in a file called a *script* and then run the interpreter in *script mode* to execute the script. By convention, Perl 6 scripts have names that end with `.pl`, `.p6`, or `.pl6`.

Please make sure that you're really using a *text editor* and not a *word-processing program* (such as MS Word, OpenOffice, or LibreOffice Writer). There is a very large number of text editors available for free. On Linux, you might use `vi` (or `vim`), `emacs`, `gEdit`, or `nano`. On Windows, you may use `notepad` (very limited) or `notepad++`. There are also many cross-platform editors or integrated development environments (IDEs) providing text editor functionality, including `padre`, `eclipse`, or `atom`. Many of these provide various syntax highlighting capabilities, which might help you use correct syntax (and find some syntax errors).

Once you've saved your code in a file (say, for example, `my_script.pl6`), you can run the program by issuing the following command at the operating system prompt (for example in a Linux console or in a `cmd` window under Windows):

```
perl6 my_script.pl6
```

Because Perl provides both modes, you can test bits of code in interactive mode before you put them in a script. But there are differences between interactive mode and script mode that can be confusing.

For example, if you are using Perl 6 as a calculator, you might type:

```
> my $miles = 26.2;
26.2
> $miles * 1.61;
42.182
```

The first line assigns a value to `$miles` and displays that value. The second

line is an expression, so the interpreter evaluates it and displays the result. It turns out that a marathon is about 42 kilometers.

But if you type the same code into a script and run it, you get no output at all. In script mode, an expression, all by itself, has no visible effect. Perl actually evaluates the expression, but it doesn't display the value unless you tell it to:

```
my $miles = 26.2;
say $miles * 1.61;
```

This behavior can be confusing at first. Let's examine why.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the print statements execute.

For example, consider the following script:

```
say 1;
my $x = 2;
say $x;
```

It produces the following output:

```
1
2
```

The assignment statement produces no output.

To check your understanding, type the following statements in the Perl interpreter and see what they do:

```
5;
my $x = 5;
$x + 1;
```

Now put the same statements in a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again.

## One-Liner Mode

Perl also has a *one-liner mode*, which enables you to type directly a very short script at the operating system prompt. Under Windows, it might look like this:

```
C:\Users\Laurent>perl6 -e "my $value = 42; say 'The answer is ',  
$value;"  
The answer is 42
```

The `-e` option tells the compiler that the script to be run is not saved in a file but instead typed at the prompt between quotation marks immediately after this option.

Under Unix and Linux, you would replace double quotation marks with apostrophes (or single quotes) and apostrophes with double quotation marks:

```
$ perl6 -e 'my $value = 42; say "The answer is $value";'  
The answer is 42
```

The one-liner above may not seem to be very useful, but throwaway one-liners can be very practical to perform simple one-off operations, such as quickly modifying a file not properly formatted, without having to save a script in a separate file before running it.

We will not give any additional details about the one-liner mode here, but will give some more useful examples later in this book, for example, “[Words Longer Than 20 Characters \(Solution\)](#)”, “[Exercise 7-3: Caesar’s Cipher](#)” (solving the “rot-13” exercise), or “[Exercise 8-7: Consecutive Double Letters](#)” (solving the exercise on consecutive double letters).

## Order of Operations

When an expression contains more than one operator, the order of evaluation depends on the *order of operations* or *operator precedence*. For mathematical operators, Perl follows mathematical convention. The acronym *PEMDAS*<sup>2</sup> is a useful way to remember the rules:

- *P*arentheses have the highest (or tightest) precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first,  $2 * (3 - 1)$  is 4, and  $(1 + 1) ** (5 - 2)$  is 8. You can also use parentheses to make an expression easier to read, as in  $(\$minute * 100) / 60$ , even if it doesn't change the result.
- *E*xponentiation has the next highest precedence, so  $1 + 2 ** 3$  is 9 ( $1 + 8$ ), not 27, and  $2 * 3 ** 2$  is 18, not 36.
- *M*ultiplication and *D*ivision have higher precedence than *A*ddition and *S*ubtraction. So  $2 * 3 - 1$  is 5, not 4, and  $6 + 4 / 2$  is 8, not 5.
- Operators with the same precedence are usually evaluated from left to right (except exponentiation). So in the expression  $\$degrees / 2 * pi$ , the division happens first and the result is multiplied by  $pi$ , which is not the expected result. (Note that  $pi$  is not a variable, but a predefined constant in Perl 6, and therefore does not require a sigil.) To divide by  $2\pi$ , you can use parentheses:

```
my $result = $degrees / (2 * pi);
```

or write  $\$degrees / 2 / pi$  or  $\$degrees / 2 / \pi$ , which will divide  $\$degrees$  by 2, and then divide the result of that operation by  $pi$  (which is equivalent  $\$degrees$  by  $2\pi$ ).



I don't work very hard to remember the precedence of operators. If I can't tell by looking at the expression, I use parentheses to make it obvious. If I don't know for sure which of two operators has the higher precedence, then the next person reading or maintaining the code may also not know.

## String Operations

In general, you can't perform mathematical operations on strings, unless the strings look so much like numbers that Perl can transform or *coerce* them into numbers and still make sense, so the following are illegal:

```
'2' - '1a'    'eggs' / 'easy'    'third' * 'a charm'
```

For example, this produces an error:

```
> '2' - '1a'
Cannot convert string to number: trailing characters after number
in '1?a' (indicated by ?)
in block <unit> at <unknown file>:1
```

But the following expressions are valid because these strings can be coerced to numbers without any ambiguity:

```
> '2' - '1'
1
> '3' / '4'
0.75
```

The `~` operator performs *string concatenation*, which means it joins the strings by linking them end-to-end. For example:

```
> my $first = 'throat'
throat
> my $second = 'warbler'
warbler
```



```
$percentage = ($minute * 100) / 60;      # percentage of an hour
```

Everything from the `#` to the end of the line is ignored—it has no effect on the execution of the program.

Comments are most useful when they document nonobvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is more useful to explain *why*.

This comment is redundant with the code and useless:

```
my $value = 5;          # assign 5 to $value
```

This comment, by contrast, contains useful information that is not in the code:

```
my $velocity = 5;      # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

## Debugging

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

### Syntax error

*Syntax* refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `8)` is a syntax error.<sup>3</sup>

If there is a syntax error anywhere in your program, Perl displays an error message and quits without even starting to run your program, and

you will obviously not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

## Runtime error

The second type of error is a *runtime error*, so called because the error does not appear until after the program has started running. These errors are also called *exceptions* because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one. We have seen one example of such errors, though, at the beginning of “**String Operations**”, when we tried to subtract '2' - '1a'.

## Semantic error

The third type of error is *semantic*, which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you *told* it to do, but not what you *intended* it to do.

Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

## Glossary

### assignment

A statement that assigns a value to a variable.

### comment

Information in a program that is meant for other programmers (or

anyone reading the source code) and has no effect on the execution of the program.

concatenate

To join two string operands end-to-end.

evaluate

To simplify an expression by performing the operations in order to yield a single value.

exception

An error that is detected while the program is running.

execute

To run a statement and do what it says.

expression

A combination of operators and terms that represents a single result.

interactive mode (or interpreter mode)

A way of using the Perl interpreter by typing code at the prompt.

keyword

A reserved word that is used to parse a program; in many languages, you cannot use keywords like `if`, `for`, and `while` as variable names. This problem usually does not occur in Perl because variable names begin with a *sigil*.

one-liner mode

A way of using the Perl interpreter to read code passed at the operating system prompt and run it.

operand

One of the values used by an operator.

order of operations

Rules governing the order in which expressions involving multiple operators and operands are evaluated. It is also called operator precedence.

script

A program stored in a file.

script mode

A way of using the Perl interpreter to read code from a script and run it.

semantic error

An error in a program that makes it do something other than what the programmer intended.

semantics

The meaning of a program.

state diagram

A graphical representation of a set of variables and the values they refer to.

statement

A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements. Statements usually end with a semicolon.

syntax error

An error in a program that makes it impossible to parse (and therefore impossible to compile and to run).

term

A variable or a literal value.

variable

Informally, a name that refers to a value. More accurately, a variable is a container that has a name and holds a value.

## Exercises

### Exercise 2-1.

Repeating our advice from the previous chapter, whenever you learn a new feature, you should try it out in interactive mode (under the REPL) and make errors on purpose to see what goes wrong.

- We've seen that `$n = 42` is legal. What about `42 = $n`?
- How about `$x = $y = 1`? (Hint: note that you will have to declare both variables, for example with a statement such as `my $x; my $y;` or possibly `my ($x, $y);`, before you can run the above.)
- In some languages, statements don't have to end with a semicolon, `;`. What happens in script mode if you omit a semicolon at the end of a Perl statement?
- What if you put a period at the end of a statement?
- In math notation you can multiply  $x$  and  $y$  like this:  $xy$ . What happens if you try that in Perl?

### Exercise 2-2.

Practice using the Perl interpreter as a calculator:

1. The volume of a sphere with radius  $r$  is  $\frac{4}{3}\pi r^3$ . What is the volume of a sphere with radius 5?
2. Suppose the cover price of a book is \$24.95, but bookstores get a 40% discount. Shipping costs \$3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?
3. If I leave my house at 6:52 a.m. and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time is it when I complete my running exercise?

<sup>1</sup> Because the words appear to be skewered like pieces of food prepared for a barbecue.

<sup>2</sup> US students are sometimes taught to use the “Please Excuse My Dear Aunt Sally” mnemonic to remember the right order of the letters in the acronym

<sup>3</sup> We are using “syntax error” here as a quasi-synonym for “compile-time error”; they are not exactly the same thing (you may in theory have syntax errors that are not compile-time errors and the other way around), but they can be deemed to be the same for practical purposes here. In Perl 6, compile-time errors have the “===SORRY!===” string at the beginning of the error message.



## Chapter 3. Functions

In the context of programming, a *function* is usually a named sequence of statements that performs a computation. In Perl, functions are often also called *subroutines*, and the two terms can (for now) be considered more or less equivalent. When you define a function, you specify the name and the sequence of statements. Later, when you want to perform a computation, you can “call” the function by name and this will run the sequence of statements contained in the *function definition*.

Perl comes with many built-in functions that are quite handy. You’ve already seen some of them: for example, `say` is a built-in function, and we will see many more in the course of this book. And if Perl doesn’t already have a function that does what you want, you can build your own. This teaches you the basics of functions and how to build new ones.

### Function Calls

We have already seen examples of *function calls*:

```
> say 42;  
42
```

The name of the function is `say`. The expression following the function name is called the *argument* of the function. The `say` function causes the argument to be displayed on the screen. If you need to pass several values to a function, then just separate the arguments with commas:

```
> say "The answer to the ultimate question is ", 42;  
The answer to the ultimate question is 42
```

Many programming languages require the arguments of a function to be inserted between parentheses. This is not required (and usually not recommended) in Perl 6 for most built-in functions (except when needed

for precedence), but if you do use parentheses, you should make sure to avoid inserting spaces between the function name and the opening parenthesis. For example, the `round` function usually takes two arguments: the value to be rounded and the unit or scale. You may call it in any of the following ways:

```
> round 42.45, 1;
42
> round 42.45, .1;
42.5
> round(42.45, .1);      # But not: round (42.45, .1);
42.5
> round( 42.45, .1);    # Space is OK *after* the opening paren
42.5
```

Experienced Perl programmers usually prefer to omit the parentheses when they can. Doing so makes it possible to chain several functions with a visually cleaner syntax. Consider for example the differences between these two calls:

```
> say round 42.45, 1;
42
> say(round(42.45, 1));
42
```

The second statement is explicitly saying what is going on, but the accumulation of parentheses actually makes things not very clear. By contrast, the first statement can be seen as a pipeline to be read from right to left: the last function on the right, `round`, is taking two arguments, `42.45`, `1`, and the value produced by `round` is passed as an argument to `say`.

It is common to say that a function “takes” one or several arguments and “returns” a result. The result is also called the *return value*.

Perl provides functions that convert values from one type to another. When called with only one argument, the `round` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
> round 42.3;
42
> round "yes"
Cannot convert string to number: base-10 number must begin with
valid
digits or '.' in '<HERE>yes' (indicated by <HERE>)
  in block <unit> at <unknown file> line 1
```

Note that, in Perl 6, many built-in functions can also use a *method invocation* syntax with the so-called *dot notation*. The following statements display the same result:

```
> round 42.7;      # Function call syntax
43
> 42.7.round;     # Method invocation syntax
43
```

The `round` function can round off rational and floating-point values to integers. There is an `Int` method that can also convert noninteger numerical values into integers, but it doesn't round off; it chops off the fraction part:

```
> round 42.7
43
> 42.7.Int
42
```

We'll come back to methods in the next section.

The `Rat` built-in function converts integers and strings to rational numbers (if possible):

```
> say 4.Rat;
4
> say 4.Rat.WHAT;
(Rat)
> say Rat(4).WHAT
(Rat)
> say Rat(4).nude
(4 1)
> say Rat('3.14159')
3.14159
> say Rat('3.14159').nude
(314159 100000)
```

(As you might remember, the `nude` method displays the *numerator* and *denominator* of a rational number.)

Finally, `Str` converts its argument to a string:

```
> say 42.Str.WHAT
(Str)
> say Str(42).WHAT;
(Str)
```

Note that these type conversion functions often don't need to be called explicitly, as Perl will in many cases try to do the right thing for you. For example, if you have a string that looks like an integer number, Perl will coerce the string to an integer for you if you try to apply an arithmetic operation on it:

```
> say "21" * "2";
42
```

Similarly, integers will be coerced to strings if you apply the string concatenation operator to them:

```
> say 4 ~ 2;
```

```
42
> say (4 ~ 2).WHAT;
(Str)
```

The coercion can even happen twice within the same expression if needed:

```
> say (4 ~ 1) + 1;
42
> say ((4 ~ 1) + 1).WHAT;
(Int)
```

## Functions and Methods

A method is similar to a function—it takes arguments and returns a value—but the calling syntax is different. With a function, you specify the name of the function followed by its arguments. A method, by contrast, uses the dot notation: you specify the name of the object on which the method is called, followed by a dot and the name of the method (and possibly additional arguments).

A method call is often called an *invocation*. The deeper differences between functions and methods will become apparent much later, when studying object-oriented programming (in [Chapter 12](#)).

For the time being, we can consider that the difference is essentially a matter of a different calling syntax when using Perl’s built-ins. Most Perl built-ins accept both a function call syntax and a method invocation syntax. For example, the following statements are equivalent:

```
> say 42;           # function call syntax
42
> 42.say;          # method invocation syntax
42
```

You can also chain built-in routines with both syntactic forms:

```
> 42.WHAT.say;          # method syntax
(Int)
> say WHAT 42;         # function syntax
(Int)
> say 42.WHAT;        # mixed syntax
(Int)
```

It is up to you to decide whether you prefer one form or the other, but we will use both forms, if only to get you used to both of them.

## Math Functions

Perl provides most of the familiar mathematical functions.

For some less common functions, you might need to use a specialized module such as `Math::Matrix` or `Math::Trig`. A *module* is a file that contains a collection of related functions.

Before we can use the functions in a module, we have to import it with a *use statement*:

```
use Math::Trig;
```

This statement will import a number of functions that you will then be able to use as if you had defined them in your main source file, for example `deg2rad` to perform conversion of angular values from degrees to radians, or `rad2deg` to perform the opposite conversion.

For most common mathematical functions, however, you don't need any `math` module, as they are included in the core of the language:

```
> my $noise-power = 5.5;
5.5
> my $signal-power = 125.6;
125.6
> my $decibels = 10 * log10 $signal-power / $noise-power;
13.5862694990693
```

This example uses `log10` (common logarithm) to compute a signal-to-noise ratio in decibels (assuming that `signal-power` and `noise-power` are defined in the proper units). Perl also provides a `log` function which, when receiving one argument, computes logarithm base `e` of the argument, and, when receiving two arguments, computes the logarithm of the first argument to the base of the second argument:

```
> say e;                # e is predefined as Euler's constant
2.71828182845905
> my $val = e ** e;
15.1542622414793
> say log $val;         # natural logarithm
2.71828182845905
> say log $val, e;     # logarithm base e or natural logarithm
2.71828182845905
> say log 1024, 2;    # binary logarithm or logarithm base 2
10
```

Perl also provides most common trigonometric functions:

```
> my $radians = 0.7;
0.7
> my $height = sin $radians;
0.644217687237691
```

This example finds the sine of `$radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, you may use the `deg2rad` function of the `Math::Trig` module, or simply divide by 180 and multiply by `pi`:

```
> my $degrees = 45;
45
> my $radians = $degrees / 180.0 * pi;    # pi, predefined
constant
```

```
0.785398163397448
> say sin $radians;      # should be square root of 2 divided by
2
0.707106781186547
```

The expression `pi` is a predefined constant for an approximation of  $\pi$ , accurate to about 14 digits.

If you know trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
> say sqrt(2) / 2;
0.707106781186548
```

## Composition

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and *compose* them, i.e., to combine them in such a way that the result of one is the input of another one. For example, the argument of a function can be any kind of expression, including arithmetic operations:

```
> my $degrees = 45;
45
> my $height = sin($degrees / 360.0 * 2 * pi);
0.707106781186547
```

Here, we have used parentheses for the argument to the `sin` function to clarify that all the arithmetic operations within the parentheses are completed before the `sin` function is actually called, so that it will use the result of these operations as its argument.



You can also compose function calls:

```
> my $x = 10;
10
> $x = exp log($x+1)
11
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name, possibly along with its declaration. Almost any other expression on the left side is a syntax error (we will see rare exceptions to this rule later):

```
> my $hours = 1;
1
> my $minutes = 0;
0
> $minutes = $hours * 60;          # right
60
> $hours * 60 = $minutes;          # wrong !!
Cannot modify an immutable Int
in block <unit> at <unknown file> line 1
```

## Adding New Functions (a.k.a. Subroutines)

So far, we have only been using the functions that come with Perl, but it is also possible to add new functions. In Perl, user-defined functions are often called subroutines, but you might choose either word for them.

A *function definition* starts with the `sub` keyword (for subroutine) and specifies the name of a new subroutine and the sequence of statements that run when the function is called.

Here is an example of a subroutine quoting Martin Luther King's famous "I Have a Dream" speech at the Lincoln Memorial in Washington (1963):

```
sub print-speech() {  
    say "Let freedom ring from the prodigious hilltops of New  
Hampshire."  
    say "Let freedom ring from the mighty mountains of New York."  
}
```

`sub` is a keyword that indicates that this is a subroutine definition. The name of the function is `print-speech`. The rules for subroutine names are the same as for variable names: letters, numbers, and underscores are legal, as well as a dash or an apostrophe between letters, but the first character must be a letter or an underscore. You shouldn't use a language keyword (such as `if` or `while`) as the name of a function (in some cases, it might actually work, but it would be very confusing, at least for the human reader).

The empty parentheses after the name indicate that this function doesn't take any arguments. They are optional in that case, but are required when parameters need to be defined for the subroutine.

The first line of the subroutine definition is sometimes called the *header*; the rest is called the *body*. The body has to be a code block placed between curly braces and it can contain any number of statements. Although there is no requirement to do so, it is good practice (and highly recommended) to indent body statements by a few leading spaces, since it makes it easier to figure out visually where the function body starts and ends.

Please note that you cannot use a method-invocation syntax for subroutines (such as `print-speech`) that you write: you must call them with a function call syntax.

The strings in the `print` statements are enclosed in double quotes. In this specific case, single quotes could have been used instead to do the same thing, but there are many cases where they wouldn't do the same thing, so you'll have to choose one or the other depending on the circumstances.

Most people use double quotes in cases where a single quote (which is also an apostrophe) appears in the string:

```
say "And so we've come here today to dramatize a shameful  
condition.";
```

Conversely, you might use single quotes when double quotes appear in the string:

```
say 'America has given the Negro people a bad check,  
a check which has come back marked "insufficient funds."';
```

There is, however, a more important difference between single quotes and double quotes: double quotes allow *variable interpolation*, and single quotes don't. Variable interpolation means that if a variable name appears within the double-quoted string, this variable name will be replaced by the variable value; within a single-quoted string, the variable name will appear verbatim. For example:

```
my $var = 42;  
say "His age is $var.";           # -> His age is 42.  
say 'Her age is $var.';         # -> Her age is $var.
```

The reason is not that the lady's age should be kept secret. In the first string, `$var` is simply replaced within the string by its value, 42, because the string is quoted with double quotes; in the second one, it isn't because single quotes are meant to provide a more verbatim type of quoting mechanism. There are other quoting constructs offering finer control over the way variables and special characters are displayed in the output, but simple and double quotes are the most useful ones.

The syntax for calling the new subroutine is the same as for built-in functions:

```
> print-speech();  
Let freedom ring from the prodigious hilltops of New Hampshire.  
Let freedom ring from the mighty mountains of New York.
```

However, you cannot use the method-invocation syntax with such subroutines. We will see much later in this book ([Chapter 12](#)) how to create methods. For the time being, we'll stick to the function-call syntax.

Once you have defined a subroutine, you can use it inside another subroutine. For example, to repeat the previous excerpts of King's address, we could write a subroutine called `repeat-speech`:

```
sub repeat-speech() {  
    print-speech();  
    print-speech();  
}
```

And then call `repeat-speech`:

```
> repeat-speech();  
Let freedom ring from the prodigious hilltops of New Hampshire.  
Let freedom ring from the mighty mountains of New York.  
Let freedom ring from the prodigious hilltops of New Hampshire.  
Let freedom ring from the mighty mountains of New York.
```

But that's not really how the speech goes.

## Definitions and Uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
sub print-speech () {  
    say "let freedom ring from the prodigious hilltops of New  
Hampshire.";
```

```

    say "Let freedom ring from the mighty mountains of New York.";
}
sub repeat-speech () {
    print-speech();
    print-speech();
}
repeat-speech();

```

This program contains two subroutine definitions: `print-speech` and `repeat-speech`. Function definitions get executed just like other statements, but the effect is to create the function. The statements inside the function do not run until the function is called, and the function definition generates no output.

You don't have to create a subroutine before you can run it; the function definition may come after its call:

```

repeat-speech;
sub repeat-speech() {
    print-speech;
    print-speech;
}
sub print-speech() {
    # ...
}

```

## Flow of Execution

To ensure, for example, that a variable is defined (i.e., populated) before its first use, you have to know the order statements run in, which is called the *flow of execution*.

Execution always begins at the first statement of the program (well, really *almost* always, but let's say always for the time being). Statements are run one at a time, in order from top to bottom.

Subroutine definitions do not alter the flow of execution of the program,

but remember that statements inside a function don't run until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another function. Then, while running that new function, the program might have to run yet another function!

Fortunately, Perl is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

In summary, when you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

## **Parameters and Arguments**

Some of the functions we have seen require arguments. For example, when you call `sin` you pass a number as an argument. Some functions take more than one argument: for example the `round` function seen at the beginning of this chapter took two, the number to be rounded and the scale (although the `round` function may accept a single argument, in which case the scale is defaulted to 1).

Inside the subroutine, the arguments are assigned to variables called *parameters*. Here is a definition for a subroutine that takes a single argument:

```
sub print-twice($value) {  
    say $value;  
    say $value
```

```
}
```

This subroutine assigns the argument to a parameter named `$value`. Another common way to say it is that the subroutine binds the parameter defined in its header to the argument with which it was called. When the above subroutine is called, it prints the content of the parameter (whatever it is) twice.

This function works with any argument value that can be printed:

```
> print-twice("Let freedom ring")
Let freedom ring
Let freedom ring
> print-twice(42)
42
42
> print-twice(pi)
3.14159265358979
3.14159265358979
```

The same rules of composition that apply to built-in functions also apply to programmer-defined subroutines, so we can use any kind of expression as an argument for `print-twice`:

```
> print-twice('Let freedom ring! ' x 2)
Let freedom ring! Let freedom ring!
Let freedom ring! Let freedom ring!
> print-twice(cos pi)
-1
-1
```

The argument is evaluated before the function is called, so in the examples the expressions `'Let freedom ring! ' x 2` and `cos pi` are only evaluated once.

You can also use a variable as an argument:

```
> my $declaration = 'When in the Course of human events, ...'  
> print-twice($declaration)  
When in the Course of human events, ...  
When in the Course of human events, ...
```

The name of the variable we pass as an argument (`$declaration`) has nothing to do with the name of the parameter (`$value`). It doesn't matter what the variable was called back home (in the caller); here, within `print-twice`, we call the parameter `$value`, irrespective of the name or content of the argument passed to the subroutine.

## Variables and Parameters Are Local

When you create a variable inside a subroutine with the `my` keyword, it is *local*, or, more accurately, *lexically scoped*, to the function block, which means that it only exists inside the function. For example:

```
sub concat_twice($part1, $part2) {  
    my $concatenation = $part1 ~ $part2;  
    print-twice($concatenation)  
}
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
> my $start = 'Let freedom ring from '  
> my $end = 'the mighty mountains of New York.';  
> concat_twice($start, $end);  
Let freedom ring from the mighty mountains of New York.  
Let freedom ring from the mighty mountains of New York.
```

When `concat_twice` terminates, the variable `$concatenation` is destroyed. If we try to print it, we get an exception:

```
> say $concatenation;
```



```

===SORRY!=== Error while compiling <unknown file>
Variable '$concatenation' is not declared
at <unknown file>:1
-----> say <HERE>$concatenation;

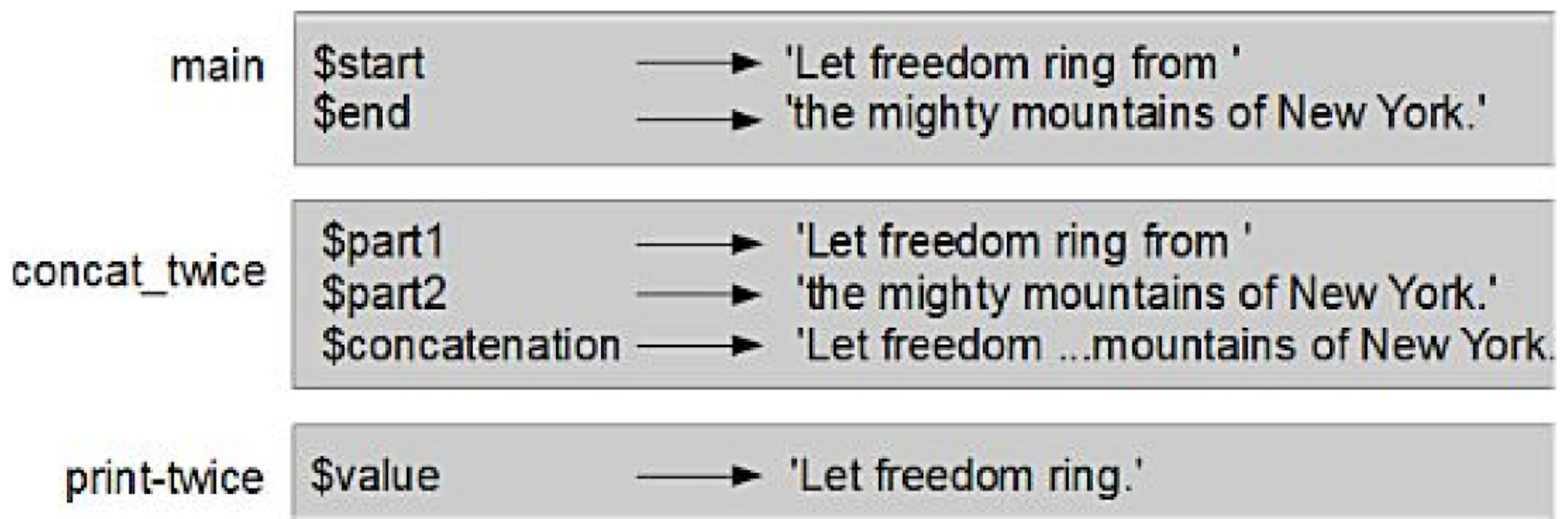
```

Parameters are also scoped to the subroutine. For example, outside `print-twice`, there is no such thing as `$value`.

## Stack Diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a *stack diagram*. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

Each function is represented graphically by a *frame*. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in [Figure 3-1](#).



*Figure 3-1. Stack diagram*

The frames are arranged in a stack that indicates which function called which, and so on. In this example, `print-twice` was called by `concat_twice`, and `concat_twice` was called by `main`, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to `main`.

Each parameter refers to the same value as its corresponding argument. So, `$part1` has the same value as `$start`, `$part2` has the same value as `$end`, and `$value` has the same value as `$concatenation`.

## **Fruitful Functions and Void Functions**

Some of the functions we have used, such as the math functions, return results and are useful only insofar we use that return value; for lack of a better name, we may call them *fruitful functions*. Other functions, like `print-twice`, perform an action but don't appear to return a value (it does in fact return a value, `True`, but we don't care about it). They are sometimes called empty or *void functions* in some other programming languages.

In some programming languages, such as Pascal or Ada, there is a strong distinction between a *function* (which returns a value) and a *procedure* (which doesn't); they are even defined with different keywords. This distinction does not apply to Perl and to most modern programming languages.

In fact, from a pure syntactic standpoint, Perl functions always return a result. So the distinction between “fruitful” and “void” functions does not really exist syntactically, but only semantically, i.e., from the standpoint of the meaning of the program: maybe we need to use the return value, or maybe we don't.

Another distinction commonly made is between functions and mutators: functions do not change the initial state of the arguments they were called on, and mutators do modify it. We will not use this distinction here, but it is useful to keep it in mind.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
my $height = sin $radians;
my $golden = (sqrt(5) + 1) / 2;
```

When you call a function in interactive mode (under the REPL), Perl usually displays the result:

```
> sqrt 5;
2.23606797749979
```

But in a script, if you call a fruitful function all by itself, the return value is lost forever! In some cases, the compiler will be able to warn you, but not always. For example, consider the following program:

```
my $five = 5;
sqrt $five;
say $five;
```

It produces the following warning:

```
WARNINGS for /home/Laurent/perl6_tests/sqrt.pl6:
Useless use of "sqrt $five" in expression "sqrt $five" in sink
context (line 2)
5
```

This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful.

Void functions might display something on the screen, save some data to a file, modify a variable or an object, or have some other effect, but they generally don't have a return value, or at least not a useful one. If you assign the result to a variable, you may get the return value of the subroutine, the value of the last expression which was evaluated in the function, or a special value such as `Any`, which essentially means something

that has not been defined, or `Nil`.

The subroutines we have written so far were essentially printing things to the screen. In that case, they usually return `True`, at least when the printing was successful. Although they return a true value, what they return isn't very useful and we can consider them all void for our practical purposes.

The following is an example of a very simple fruitful subroutine:

```
> sub square($number) { return $number ** 2 }  
sub square ($number) { #`(Sub|118134416) ... }  
> say square 5;  
25
```

The `Sub|118134416` message displayed by the REPL is just an internal identifier for the subroutine we've just defined.

The `return` statement instructs the function to terminate the execution of the function at this statement and to return the value of the following expression to the caller. In such a simple case where the program is in fact running the last statement of a function, the `return` keyword can be omitted since the function will return the value of the last evaluated statement, so that the `square` subroutine could be written this way:

```
sub square($number) {  
  $number ** 2  
}
```

We will be using fruitful functions more intensively in a few chapters.

## Function Signatures

When a function receives arguments, which are stored into parameters, the part of the function definition describing the parameters between parentheses is called the *function signature*. The function signatures we

have seen so far are very simple and consist only of one parameter or possibly a parameter list.

Signatures can provide a lot more information about the parameters used by a function. First, you may define the type of the parameters. Some functions make sense only if their parameters are numeric and should probably raise an error if they get a string that cannot be converted to a numeric value. For example, if you define a function `half` that computes a value equal to its argument divided by 2, it does not make sense to try to compute half of a string that is not numeric. It could be written as follows:

```
sub half(Int $number) {  
    return $number / 2  
}  
say half 84; # -> 42
```

If this function is called with a string, we get the following error:

```
> say half "Douglas Adams"  
===SORRY!=== Error while compiling <unknown file>  
Calling half(Str) will never work with declared signature (Int  
$number)  
at <unknown file>:1  
-----> say <HERE>half "Douglas Adams"
```

The `Int` type included in the function signature is a type constraint that can help prevent subtle bugs. In some cases, it can also be an annoyance.

Consider this code snippet:

```
sub half(Int $number) { $number / 2 }  
say half "84"; # -> ERROR
```

Because the argument to the `half` subroutine is `"84"`, i.e., a string, this code will fail with a type error. If we had not included the `Int` type in the

signature, the script would have converted (or coerced) the "84" string to a number, divided it by two, and printed out the expected result:

```
sub half( $number ) { $number / 2 }  
say half "84"; # -> 42
```

In some cases, you want this conversion to occur, in others you don't. It is up to you to decide whether you want strict typing or not, depending on the specific situation and needs. It is probably helpful to use parameter typing in many cases, but it can also become a straitjacket in some situations. Perl 6 lets you decide how strict you want to be about these things.

Our original `half` subroutine has another limitation: it can work only on integers. But a function halving its argument should presumably be useful for rational or even other numbers. You can use the `Real` or `Numeric` types to make the function more general (the difference between the two types is that the `Numeric` type will accept not only `Real` but also `Complex` numbers). As it turns out that this `half` function will also work correctly with complex numbers,<sup>1</sup> choosing a `Numeric` type opens more possibilities:

```
sub half(Numeric $number) { $number / 2 }  
say half(3+4i); # -> 1.5+2i
```

The following table sums up and illustrates some of the various types we have seen so far:

<b>Type</b>	<b>Example</b>
String	"A string", 'Another string', "42"
Integer	-3, -2, 0, 2, 42
Rational	1/2, 0.5, 3,14159, 22/7, 42.0
Real	, pi, 2, e, <b>log 42</b> , <b>sin 0.7</b>
Complex	<b>5.4 + 3i</b>

## Immutable and Mutable Parameters

By default, subroutine parameters are *immutable* aliases for the arguments passed to the subroutine. In other words, they cannot be changed within the function and you cannot accidentally modify the argument in the caller:

```
sub plus-three(Int $number) { $number += 3}
my $value = 5;
say plus-three $value; # ERROR: Cannot assign to an immutable
value
```

In some other languages, this behavior is named a “call by value” semantic: loosely speaking, the subroutine receives (by default) a value rather than a variable, and the parameter therefore cannot be modified.

If you want to change the value of the parameter within the subroutine (but without changing the argument in the caller) you can add the `is copy` trait to the signature:

```
sub plus-three(Int $number is copy) { $number += 3}
my $value = 5;
say plus-three $value; # 8
```

```
say $value;           # 5 (unchanged)
```

A trait is a property of the parameter defined at compile time. Here, the `$number` parameter is modified within the subroutine and the incremented value is returned to the caller and printed as 8, but, within the caller, the variable used as an argument to the function, `$value`, is not modified (it is still 5).

Although this can sometimes be dangerous, you may also want to write a subroutine that modifies its argument at the caller side. For this, you can use the `is rw` trait in the signature:

```
sub plus-three(Int $number is rw) { $number += 3}
my $value = 5;
say plus-three $value; # 8
say $value;           # 8 ($value modified)
```

With the `is rw` trait, the `$number` parameter is now *bound* to the `$value` argument, so that any change made using `$number` within the subroutine will immediately be applied to `$value` at the caller side, because `$number` and `$value` are just different names for the same thing (they both refer to the same memory location). The argument is now fully *mutable*.

In some other languages, this is named a “call by reference” parameter passing mechanism, because, in those languages, if you pass a reference (or a pointer) to a variable to a function, then it is possible for the function to modify the variable referred to by the reference.

## Functions and Subroutines as First-Class Citizens

Subroutines and other code objects can be passed around as values, just like any variable, literal, or object. Functions are said to be *first-class objects* or sometimes first-class citizens or higher-order functions. This means that a Perl function (its code, not the value returned by it) is a value you can



assign to a variable or pass around as an argument. For example, `do-twice` is a subroutine that takes a function as an argument and calls it twice:

```
sub do-twice($code) {  
    $code();  
    $code();  
}
```

Here, the `$code` parameter refers to a function or some other callable code object. This is an example that uses `do-twice` to call a function named `greet` twice:

```
sub greet {  
    say "Hello World!";  
}  
do-twice &greet;
```

This will print:

```
Hello World!  
Hello World!
```

The `&` sigil placed before the subroutine name in the argument list tells Perl that you are passing around a subroutine or some other callable code object (and not calling the subroutine at the moment).

In fact, it would be more idiomatic to also use the `&` sigil in the `do-twice` subroutine definition, to better specify that the parameter is a callable code object:

```
sub do-twice(&code) {  
    &code();  
    &code();  
}
```

or even:

```
sub do-twice(&code) {
    code();
    code();
}
```

The syntax with the & sigil has the benefit that it will provide a better error message if you make a mistake and pass something noncallable to `do-twice`.

All the functions we have seen so far had a name, but a function does not need to have a name and can be *anonymous*. For example, it may be stored directly in a scalar variable:

```
my $greet = sub {
    say "Hello World!";
};
$greet();           # prints "Hello World"
do-twice $greet;    # prints "Hello World" twice
```

It could be argued that the above `$greet` subroutine is not really anonymous, since it is stored in a scalar variable that could in a certain way be considered its name. But the subroutine really has no name; it just happens to be assigned to a scalar variable. Just to show that the subroutine can really have no name at all, consider this:

```
do-twice(sub {say "Hello World!"} );
```

It will happily print “Hello World” twice. If the `$do-twice` function was declared earlier, you can even simplify the syntax and omit the parentheses:

```
do-twice sub {say "Hello World!"};
```

For such a simple case where there is no need to pass an argument or return a value; you can even omit the `sub` keyword and pass a code block directly to the function:

```
do-twice {say "Hello World!";}
do-twice {say "what's up doc"};
```

As you can see, `do-twice` is a *generic* subroutine in charge of just performing twice any function or code block passed to it, without any knowledge about what this function or code block is doing. This is a powerful concept for some relatively advanced programming techniques that we will cover later in this book.

Subroutines may also be passed as return values from other subroutines:

```
> sub create-func ($person) { return sub { say "Hello $person!"}}
# Creating two greeting functions
sub create-func ($person) { #`(Sub|176738440) ... }
> my $greet_world = create-func "World";
sub () { #`(Sub|176738592) ... }
> my $greet_friend = create-func "dear friend";
sub () { #`(Sub|176739048) ... }
# Using the greet functions
> $greet_world();
Hello World!
> $greet_friend();
Hello dear friend!
```

Here, `create-func` returns a subroutine greeting someone. It is called twice with two different arguments in order to create two different functions at runtime, `$greet_world` and `$greet_friend`. A function such as `create-func` is sometimes a *function factory* because you may create as many functions as you like by just calling `create-func`. This example may seem to be a slightly complicated way of doing something quite simple. At this point, it is just a bit too early to give really useful examples, but this is

also a very powerful programming technique.

We'll come back to these techniques in various places in this book and even devote an entire chapter ([Chapter 14](#)) to this subject and related topics.

## **Why Functions and Subroutines?**

It may not be clear why it is worth the trouble to divide a program into functions or subroutines. There are several reasons:

- Creating a new subroutine gives you an opportunity to name a group of statements, which makes your program easier to read and debug. Subroutines also help make the flow of execution clearer to the reader.
- Subroutines can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into subroutines allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed subroutines are often useful for many programs. Once you write and debug one, you can reuse it.
- Creating subroutines is one of the major ways to break up a difficult problem into smaller easier subtasks and to create successive layers of abstraction, which are the key to solve complex problems.
- Writing good subroutines lets you create black boxes, with a known input and a known output. So you don't have to think about them anymore when you're working on something else. They've become a tool. Once you've assembled an electric screwdriver, you don't need to think about how it works internally when you use it to build or repair something.
- In the current open source world, chances are that your code will have to be understood, maintained, or enhanced by people other than you. Coding has become much more of a social activity than before.

Breaking up your code into small subroutines whose purpose is easy to understand will make their work easier. And you'll be even more delighted when the person having to maintain or refactor your code is...you.

## Debugging

One of the most important programming skills you will acquire is debugging. Although it can sometimes be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "...When you have eliminated the impossible, whatever remains, however improbable, must be the truth" (A. Conan Doyle, *The Sign of Four*).

In cases where you are not able to come up with a hypothesis on what's wrong, you can try to introduce code that you expect to create a certain type of error, a "negative hypothesis" if you will. Sometimes you can learn a lot from the fact that it didn't create the error that was expected. Making a hypothesis does not necessarily mean you have an idea about how to make the code work; it could also be a hypothesis on how it should break.

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a working program

and make small modifications, debugging them as you go.

For example, Linux is an operating system that contains millions of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, “One of Linus’s earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux.” (*The Linux Users’ Guide Beta Version 1*).

## **Glossary**

anonymous function

A function that has no name.

Any

A special value typically found in variables that haven’t been assigned a value. It is also a special value returned by some functions that we have called “void” (because they return something generally useless such as “Any”).

argument

A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

body

The sequence of statements inside a function definition, usually in a code block delimited by braces.

composition

Using an expression as part of a larger expression, or a statement as part of a larger statement.

first-class object

Perl's subroutines are said to be higher order objects or first-class objects, because they can be passed around as other subroutines' arguments or return values, just as any other objects.

flow of execution

The order in which statements run.

frame

A box in a stack diagram that represents a subroutine call. It contains the local variables and parameters of the subroutine.

fruitful function

A function or subroutine that returns a useful value.

function

A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result. Perl comes with many built-in functions, and you can also create your own. In Perl, user-defined functions are often called subroutines.

function call

A statement that runs a function. It consists of the function name followed by an argument list, which may or may not be enclosed within parentheses.

function definition

A statement that creates a new function, specifying its name, parameters, and the statements it contains.

function factory

A function that produces other functions as return values.

function signature

The part of the definition of a function (usually between parentheses) that defines its parameters and possibly their types and other properties.

header

The first line of a function definition.

immutable parameter

A function or subroutine parameter that cannot be changed within the function body. By default, subroutine parameters are immutable.

lexical variable

A variable defined inside a subroutine or a code block. A lexical variable defined within a function can only be used inside that function.

module

A file that contains a collection of related functions and other definitions.

Nil

A special value sometimes returned by some “void” subroutines.

parameter

A name used inside a subroutine to refer to the value passed as an argument.

return value

The result of a function. If a function call is used as an expression, the return value is the value of the expression.

stack diagram



A graphical representation of a stack of subroutines, their variables, and the values they refer to.

trait

A property of a function or subroutine parameter that is defined at compile time.

use statement

A statement that reads a module file and usually imports some functions.

void function

A function or subroutine that does not return a useful value.

## Exercises

### Exercise 3-1.

Write a subroutine named `right-justify` that takes a string named `$input-string` as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.

```
> right-justify('Larry Wall')
```

Larry

```
Wall
```

Hint: use string concatenation and repetition. Also, Perl provides a built-in function called `chars` that returns the length of a string, so the value of `chars 'Larry Wall'` or `'Larry Wall'.chars` is 10. Solution: “**Exercise 3-1: Subroutine `right-justify`**”.

### Exercise 3-2.

We have seen that functions and other code objects can be passed around as values, just like any object. Functions are said to be *first-class objects*. For

example, `do-twice` is a function that takes a function as an argument and calls it twice:

```
sub do-twice($code) {  
    $code();  
    $code();  
}  
sub greet {  
    say "Hello World!";  
}  
do-twice(&greet);
```

1. Type this example into a script and test it.
2. Modify `do-twice` so that it takes two arguments, a function and a value, and calls the function twice, passing the value as an argument.
3. Copy the definition of `print-twice` from earlier in this chapter to your script.
4. Use the modified version of `do-twice` to call `print-twice` twice, passing “What’s up doc” as an argument.
5. Define a new function called `do-four` that takes a function and a value and calls the function four times, passing the value as a parameter. There should be only two statements in the body of this function, not four.

Solution: “[Exercise 3-2: Subroutine do-twice](#)”.

### **Exercise 3-3.**

Note: this exercise should be done using only the statements and other features we have learned so far.

1. Write a subroutine that draws a grid like the following:

```

+ - - - - + - - - - +
|           |           |
|           |           |
|           |           |
|           |           |
+ - - - - + - - - - +
|           |           |
|           |           |
|           |           |
|           |           |
+ - - - - + - - - - +

```

Hint: to print more than one value on a line, you can print a comma-separated sequence of values:

```
say '+', '-';
```

The `say` function prints its arguments with a newline at the end (it advances to the next line). If you don't want to go to the next line, use the `print` function instead:

```
print '+', ' ';
print '-';
```

The output of these statements is “+ -”.

A `say` statement with an empty string argument ends the current line and goes to the next line.

- Write a subroutine that draws a similar grid with four rows and four columns.

Solution: “Exercise 3-3: Subroutine print-grid”.

Credit: this exercise is based on an exercise in *Practical C Programming, 3rd Edition*, by Steve Oualline (O’Reilly, 1997).

<sup>1</sup> Complex numbers are numbers in the form “ $a + bi$ ,” where  $a$  and  $b$  are real numbers, and  $i$  an imaginary number such that  $i^2$  equals  $-1$ .

## Chapter 4. Loops, Conditionals, and Recursion

The main topic of this chapter is the `if` statement, which executes different code depending on the state of the program. But first I want to introduce two new operators: integer division and modulo.

### Integer Division and Modulo

The *integer division* operator, `div`, divides two numbers and rounds down to an integer. For example, suppose the runtime of a movie is 105 minutes. You might want to know how long that is in hours. In Perl, conventional division returns a rational number (in many languages, it returns a floating-point number, which is another kind of internal representation for noninteger numbers):

```
> my $minutes = 105;
> $minutes / 60;
1.75
```

But we don't normally write hours with decimal points. Integer division returns the integer number of hours, dropping the fraction part:

```
> my $minutes = 105;
> my $hours = $minutes div 60;
1
```

In arithmetic, integer division is sometimes called *Euclidean division*, which computes a quotient and a remainder.

To get the remainder, you could subtract off one hour in minutes:

```
> my $remainder = $minutes - $hours * 60;
45
```

An alternative is to use the *modulo operator*, `%`, which divides two numbers

and returns the remainder:

```
> my $remainder = minutes % 60;  
45
```

The modulo operator is very common in programming languages and is more useful than it seems. For example, you can check whether one number is divisible by another—if `$dividend % $divisor` is zero, then `$dividend` is divisible by `$divisor`. This is commonly used, for example, with a divisor equal to 2 in order to determine whether an integer is even or odd. We will see an example of that later in this chapter (see “[Alternative Execution](#)”).

To tell the truth, Perl 6 also has a specific operator for divisibility, `%%`. The `$dividend %% $divisor` expression returns a true value if `$dividend % $divisor` is equal to 0, that is if `$dividend` is divisible by `$divisor` (and false otherwise).

Also, you can extract the rightmost digit or digits from a number with the modulo operator. For example, `$x % 10` yields the rightmost digit of `$x` (in base 10). Similarly, `$x % 100` yields the last two digits:

```
> 642 % 100;  
42
```

## Boolean expressions

A *Boolean expression* is an expression that is either true or false. The following examples use the operator `==`, which compares two numeric operands and produces `True` if they are equal and `False` otherwise:

```
> 5 == 5;  
True
```

```
> 5 == 6;
False
```

True and False are special values that belong to the type `Bool`; they are not strings:

```
> say True.WHAT
(Bool)
> say False.WHAT
(Bool)
```

The `==` operator is one of the *numeric relational operators* and checks whether the operands are equal; the others are:

```

    $x != $y           # $x is not numerically equal to $y
    $x > $y            # $x is numerically greater than $y
    $x < $y            # $x is numerically less than $y
    $x >= $y           # $x is numerically greater than or
equal to $y
    $x <= $y           # $x is numerically less than or equal
to $y
    $x === $y          # $x and $y are truly identical
```

Although these operations are probably familiar to you, the Perl symbols are different from the mathematical symbols. A common error is to use a single equals sign (`=`) instead of a double equals sign (`==`). Remember that `=` is an assignment operator and `==` is a relational operator. There is no such thing as `=<`, and there exists a `=>` operator, but it is not a relational operator, but something completely different (it is, as we'll see later, a pair constructor).

The difference between `==` and `===` is that the former operator checks whether the values of the operands are equal and the latter checks whether the operands are truly identical. As an example, consider this:

```

say 42 == 42;           # True
say 42 == 42.0;       # True
say 42 === 42;        # True
say 42 === 42.0;      # False

```

These relational operators can only compare numeric values (numbers or variables containing numbers) or values that can be coerced to numeric values, such as, for example, the string “42” which, if used with these operators (except `===`), will be coerced to the number 42.

For comparing strings (in a lexicographic or “pseudo-alphabetic” type of comparison), you need to use the *string relational operators*:

```

        $x eq $y           # $x is string-wise equal to $y
        $x ne $y           # $x is string-wise not equal to $y
        $x gt $y           # $x is greater than $y (alphabetically
after)
        $x lt $y           # $x is less than $y (alphabetically
before)
        $x ge $y           # $x is greater than or equal to $y
        $x le $y           # $x is less than or equal to $y
        $x eqv $y         # $x is truly equivalent to $y

```

For example, you may compare (alphabetically) two former US presidents:

```

> 'FDR' eq 'JFK';
False
> 'FDR' lt 'JFK';    # alphabetical comparison
True

```

Unlike most other programming languages, Perl 6 allows you to chain relational operators transitively, just as in mathematical notation:

```

say 4 < 7 < 12;      # True
say 4 < 7 < 5;       # False

```



It may be useful to point out that numeric relational operators and string relational operators don't work the same way (and that's a good reason for having different operators), because they don't have the same idea of what is *greater than* or *less than*.

When comparing two positive integers, a number with four digits is always greater than a number with only two or three digits. For example, 1110 is greater than 886.

String comparisons, in contrast, basically follow (pseudo) alphabetical rules: "b" is greater than "aaa" because the commonly accepted rule for string comparisons is to start by comparing the first letter of each string: which string is greater is known if the two letters are different, irrespective of what character comes next; you need to proceed to comparing the second letter of each word only if comparing the first letter of each string led to a draw, and so on. Thus, any word starting with "a" is less than any word starting with "b," irrespective of the length of these words. You may think that this is nitpicking, but this becomes essential when you start sorting items: you really have to think about which type of order (numeric or alphabetical) you want to use.

There are also some so-called "three-way" relational operators, `cmp`, `<=>`, and `leg`, but we'll come back to them when we study how to sort the items in a list. Similarly, we need to learn quite a few other things about Perl before we can do justice to the incredibly powerful and expressive smart match operator, `~~`.

A final point to be noted about string comparisons is that uppercase letters are always deemed smaller than lowercase letters. So "A," "B," "BB," and "C" are *all* less than "a," "b," "bb," and "c." We will not go into the details here, but this becomes more complicated (and sometimes confusing) when the strings to be compared contain nonalphabetical characters (or non-ASCII Unicode letters).

## Logical Operators

There are three main pairs of *logical operators*:

- logical *and*: “and” and `&&`
- logical *or*: “or” and `||`
- logical *not*: “not” and `!`

The semantics (meaning) of these operators is similar to their meaning in English. For example, `$x > 0 and $x < 10` is true only if `$x` is greater than 0 *and* less than 10.

`$n % 2 == 0 and $n % 3 == 0` is true if *both* conditions are true, that is, if the number is divisible by 2 *and* by 3, i.e., is in fact divisible by 6 (which could be better written as: `$n % 6 == 0` or `$n %% 6`).

`$n % 2 == 0 or $n % 3 == 0` is true if *either or both* of the conditions is true, that is, if the number is divisible by 2 *or* by 3 (or both).

Finally, the `not` operator negates a Boolean expression, so `not (x > y)` is true if `x > y` is false, that is, if `x` is less than or equal to `y`.

The `&&`, `||`, and `!` operators have the same meanings, respectively, as *and*, *or*, and *not*, but they have a tighter precedence, which means that when they stand in an expression with some other operators, they have a higher priority of execution. We will come back to precedence later, but let’s say for the time being that, in most common cases, the *and*, *or*, and *not* operators will usually do what you want.

Strictly speaking, the operands of the logical operators should be Boolean expressions, but Perl, just like many languages partly derived from C, is not very strict on that. The numbers 0 and 0.0 are false; and any nonzero number or nonempty string is interpreted as True:

```
> 42 and True;  
True
```

This flexibility can be very useful, but there are some subtleties to it that might be confusing. You might want to avoid it unless you know what you are doing.

The `say` built-in function returns a Boolean evaluation of its argument:

```
> say so (0 and True);  
False
```

Here, the expression `(0 and True)` is false because 0 is false and the expression could be true only if both arguments of the `and` operator were true.

When several Boolean conditions are linked with some logical operator, Perl will only perform the comparisons that are strictly necessary to figure out the final result, starting with those on the left. For example, if you write:

```
> False and $number > 0;  
False
```

there is no need to evaluate the second Boolean expression to know that the overall expression will be false. In this case, Perl does not try to check whether the number is positive or even whether it is defined. It is sometimes said that these operators “short circuit” unnecessary conditions.

Similarly, in the following code, the `compute-pension` subroutine will not even be called if the person’s age is less than 65:

```
$age >= 65 and compute-pension();
```

The same goes with the `or` operator, but the other way around: if the first Boolean expression of an `or` statement is true, then the next expression will not be evaluated. The following code is thus equivalent to the previous one:

```
$age < 65 or compute-pension();
```

This *can* be a way of running the `compute-pension` subroutine conditionally, depending on the value of the age, and this is sometimes used, notably in idiomatic constructs such as:

```
do-something() or die "could not do something";
```

which aborts the program if `do-something` returns a false value, meaning that it was not able to do something so essential that it would not make sense to try to continue running it.

We will examine now clearer and much more common ways of running conditional code.

## **Conditional Execution**

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. *Conditional statements* give us this ability. The simplest form is the `if` statement:

```
if $number > 0 {  
    say '$number is positive';  
}
```

The Boolean expression after `if` is called the *condition*. If it is true, the subsequent block of code runs. If not, nothing happens. The block of code may contain any number of statements.

It is conventional and highly recommended (although not strictly mandatory from the standpoint of the compiler) to indent the statements in the block, in order to help visualize the *control flow* of the program, i.e., its structure of execution: with such indentation, we can see much better that the statements within the conditional block will run only if the condition is true.

The condition may be a compound Boolean expression:

```
if $n > 0 and $n < 20 and $n %% 2 {  
    say '$n is an even and positive number smaller than 20'  
}
```

Note that in the print statement above, the final semicolon has been omitted. When a statement is the last code line of a block, immediately before the curly brace } closing that code block, the final semicolon is optional and may be omitted, though it might be considered good form to include it.

In theory, the overall code snippet above is itself a statement and should also end with a semicolon after the closing brace. But a closing curly brace followed by a newline character implies a statement separator, so you don't need a semicolon here and it is generally omitted.

## **Alternative Execution**

A second form of the `if` statement is “alternative execution,” in which there are two possibilities and the condition determines which one runs. Given a `$number` variable containing an integer, the following code displays two different messages depending on whether the value of the integer is even or odd:

```
if $number % 2 == 0 {  
    say 'Variable $number is even'  
} else {
```

```
    say 'Variable $number is odd'  
}
```

If the remainder when `$number` is divided by 2 is 0, then we know that `$number` is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs. Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called *branches*, because they are branches in the flow of execution.

Note that if `$number` is evenly divisible by two, this code will print:

```
Variable $number is even
```

The `$number` variable value is not interpolated, because we used single quotes for the purpose of printing out the variable name rather than its value. We would have to use double quotes if we wanted to display the variable's value instead of its name.

## Chained Conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a *chained conditional*:

```
if $x < $y {  
    say 'Variable $x is less than variable $y'  
} elsif $x > $y {  
    say 'Variable $x is greater than variable $y'  
} else {  
    say 'Variables $x and $y are equal'  
}
```

The `elsif` keyword is an abbreviation of “else if” that has the advantage of avoiding nesting of blocks. Again, exactly one branch will run. There is no

limit on the number of `elsif` statements.

If there is an `else` clause, it has to be at the end, but there doesn't have to be one:

```
if $choice eq 'a' {
    draw_a()
} elsif $choice eq 'b' {
    draw_b()
} elsif $choice eq 'c' {
    draw_c()
}
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch runs and the statement ends. Even if more than one condition is true, only the first true branch runs.

## **Nested Conditionals**

One conditional can also be nested within another. We could have written the example in the previous section like this:

```
if $x == $y {
    say 'Variables $x and $y are equal'
} else {
    if $x < $y {
        say 'Variable $x is less than variable $y'
    } else {
        say 'Variable $x is greater than variable $y'
    }
}
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another `if` statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

The `if $x < $y` conditional is said to be nested within the `else` branch of the outer conditional.

Such nested conditionals show how critical it is for your own comprehension to properly indent conditional statements, as it would be very difficult here to visually grasp the structure without the help of correct indentation.

Although the indentation of the statements helps make the structure apparent, *nested conditionals* become difficult to read very quickly. It is a good idea to avoid them when you can. Logical operators often provide a way to simplify nested conditional statements. For example, consider the following code (which assumes `$x` to be an integer):

```
my Int $x;
# ... $x = ...;
if 0 < $x {
    if $x < 10 {
        say 'Value of $x is a positive single-digit number.'
    }
}
```

The `say` statement runs only if we make it past both conditionals, so we can get the same effect with the `and` Boolean operator, and the code can be rewritten using a single conditional:

```
if 0 < $x and $x < 10 {
    say '$x is a positive single-digit number.'
}
```

For this kind of condition, Perl 6 provides a more concise option using the chained relational operators described earlier:

```
if 0 < $x < 10 {
```



```
    say '$x is a positive single-digit number.'  
}
```

## if Conditionals as Statement Modifiers

There is also a form of `if` called a *statement modifier* (or sometimes “postfix conditional”) form when there is only one conditional statement. In this case, the `if` and the condition come after the code you want to run conditionally. Note that the condition is still always evaluated first:

```
say '$number is negative.' if $number < 0;
```

This is equivalent to:

```
if $number < 0 {  
    say '$number is negative.'  
}
```

This syntactic form is more concise as it takes only one code line instead of three. The advantage is that you can see more of your program code on one screen, without having to scroll up and down. However, this syntax is neat and clean only when both the condition and the statement are short and simple, so it is probably best used only in these cases.

The statement modifier form does not allow `else` and `elsif` statements.

## Unless Conditional Statement

If you don't like having to write negative conditions in a conditional `if` statement such as:

```
if not $number >= 0 {  
    say '$number is negative.'  
}
```

you may write this instead:

```
unless $number >= 0 {  
    say '$number is negative.'  
}
```

This *unless* keyword does exactly what the English says: it will display the sentence “\$number is negative.” *unless* the number is greater than or equal to 0.

You cannot use `else` or `elsif` statements with `unless`, because that would end up getting confusing.

The `unless` conditional is most commonly used in its statement modifier (or postfix notation) form:

```
say '$number is negative.' unless $number >= 0;
```

## for Loops

Suppose you need to compute and print the product of the first five positive digits (1 to 5). This product is known in mathematics as the *factorial* of 5 and is sometimes written as **5!**. You could write this program:

```
my $product = 1 * 2 * 3 * 4 * 5;  
say $product;          # prints 120
```

You could make it slightly simpler:

```
say 2 * 3 * 4 * 5;      # prints 120
```

The problem is that this syntactic construct does not scale well and becomes tedious for the product of the first 10 integers (or factorial 10).

And it becomes almost a nightmare for factorial 100. Calculating the factorial of a number is a fairly common computation in mathematics (especially in the fields of combinatorics and probability) and in computer science. We need to automatize it, and using a `for` loop is one of the most obvious ways of doing that:

```
my $product = 1;
for 1..5 {
    $product *= $_
}
say $product;           # prints 120
```

Now, if you need to compute factorial 100, you just need to replace the 5 in the code above with 100. Beware, though, the factorial function is known to grow extremely rapidly, and you'll get a truly huge number, with 158 digits (i.e., a number much larger than the estimated total number of atoms in the known universe).

In this script, `1..5` is the range operator, which is used here to generate a list of consecutive numbers between 1 and 5. The `for` keyword is used to iterate over that list, and `$_` is a special variable that takes each successive value of this list: first 1, then 2, etc. until 5. In the code block forming the body of the loop, the `$product` variable is multiplied successively by each value of `$_`. The loop ends with 5 and the result, 120, is printed on the last line.

This is a simple use of the `for` statement, but probably not the most commonly used in Perl 6; we will see more below. We will also see other types of loops. But that should be enough for now to let you write some loops. Loops are found everywhere in computer programming.

The `$_` special variable is known as the *topical variable* or simply the *topic*. It does not need to be declared and many syntactic constructs assign a value

to it without explicitly mentioning it. Also, `$_` is an implicit argument to methods called without an explicit invocant. For example, to print the first five integers, you might write:

```
for 1..5 { .say }; # prints numbers 1 to 5, each on its line
```

Here `.say` is a syntax shorthand equivalent to `$_ .say`. And since, as we saw, `$_` takes each successive value of the range introduced by the `for` keyword, this very short code line prints each number between 1 and 5, each on a different line. This is a typical example of the `$_` topical variable being used without even being explicitly mentioned. We will see many other uses of the `$_` special variable.

Sometimes, you don't use the `$_` loop variable within the loop, for example if you just want to do something five times but don't care each time through the loop at which iteration you have arrived. A subroutine that prints a message  $n$  times might look like this:

```
sub print-n-times (Int $n, Str $message) {  
    for 1..$n { say $message }  
}
```

The `for` loop also has a statement modifier or postfix form, used here to compute again the factorial of 5:

```
my $product = 1;  
$product *= $_ for 1..5;  
say $product; # prints 120
```

There is another syntax for the `for` loop, using an explicit loop variable:

```
sub factorial (Int $num) {
```

```

my $product = 1;
for 1..$num -> $x {
    $product *= $x
}
return $product
}
say factorial 10;    # 3628800

```

The for loop in this subroutine is using what is called a “pointy block” syntax. It is essentially the same idea as the previous for loops, except that, instead of using the `$_` topical variable, we now declare an explicit `$x` loop variable with the `1..$num -> $x` syntax to iterate over the range of values. Using an explicit loop variable can make your code clearer when things get more complicated, for example when you need to nest several for loops. We will see more examples of that later.

We will also see several other ways of computing the factorial of a number in this book.

## Recursion

It is legal for one function or subroutine to call another; it is also legal for a subroutine to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following subroutine:

```

sub countdown(Int $time-left) {
    if $time-left <= 0 {
        say 'Blastoff!';
    } else {
        say $time-left;
        countdown($time-left - 1);
    }
}

```

If `$n` is 0 or negative, it outputs the word “Blastoff!” Otherwise, it outputs `$time-left` and then calls a subroutine named `countdown`—itself—

passing  $n - 1$  as an argument.

What happens if we call the subroutine like this?

```
countdown(3);
```

The execution of `countdown` begins with `$time-left = 3`, and since `$time-left` is greater than 0, it outputs the value 3, and then calls itself...

*The execution of `countdown` begins with `$time-left = 2`, and since `$time-left` is greater than 0, it outputs the value 2, and then calls itself...*

*The execution of `countdown` begins with `$time-left = 1`, and since `$time-left` is greater than 0, it outputs the value 1, and then calls itself...*

*The execution of `countdown` begins with `$time-left = 0`, and since `$time-left` is not greater than 0, it outputs the word "Blastoff!" and then returns.*

*The `countdown` that got `$time-left = 1` returns.*

*The `countdown` that got `$time-left = 2` returns.*

The `countdown` that got `$time-left = 3` returns.

And then you're back in the main program. So, the total output looks like this:

```
3
2
1
Blastoff!
```

A subroutine that calls itself is *recursive*; the process of executing it is called *recursion*.

As another example, we can write a subroutine that prints a string  $\$n$  times:

```
sub print-n-times(Str $sentence, Int $n) {  
    return if $n <= 0;  
    say $sentence;  
    print-n-times($sentence, $n - 1);  
}
```

If  $\$n \leq 0$ , the *return statement* exits the subroutine. The flow of execution immediately returns to the caller, and the remaining lines of the subroutine don't run. This illustrates a feature of the `return` subroutine that we have not seen before: it is used here for flow control, i.e., to stop the execution of the subroutine and pass control back to the caller. Note also that here the `return` statement does not return any value to the caller; `print-n-times` is a void function.

The rest of the subroutine is similar to `countdown`: it displays `$sentence` and then calls itself to display `$sentence`  $\$n - 1$  additional times. So the number of lines of output is  $1 + (\$n - 1)$ , which adds up to  $\$n$ .

For simple examples like this, it may seem easier to use a `for` loop. But we will see examples later that are hard to write with a `for` loop and easy to write with recursion, so it is good to start early.

## Stack Diagrams for Recursive Subroutines

In “[Stack Diagrams](#)”, we used a stack diagram to represent the state of a program during a subroutine call. The same kind of diagram can help interpret a recursive subroutine.

Every time a subroutine gets called, Perl creates a frame to contain the subroutine's local variables and parameters. For a recursive subroutine, there might be more than one frame on the stack at the same time.

[Figure 4-1](#) shows a stack diagram for `countdown` called with  $n = 3$ .

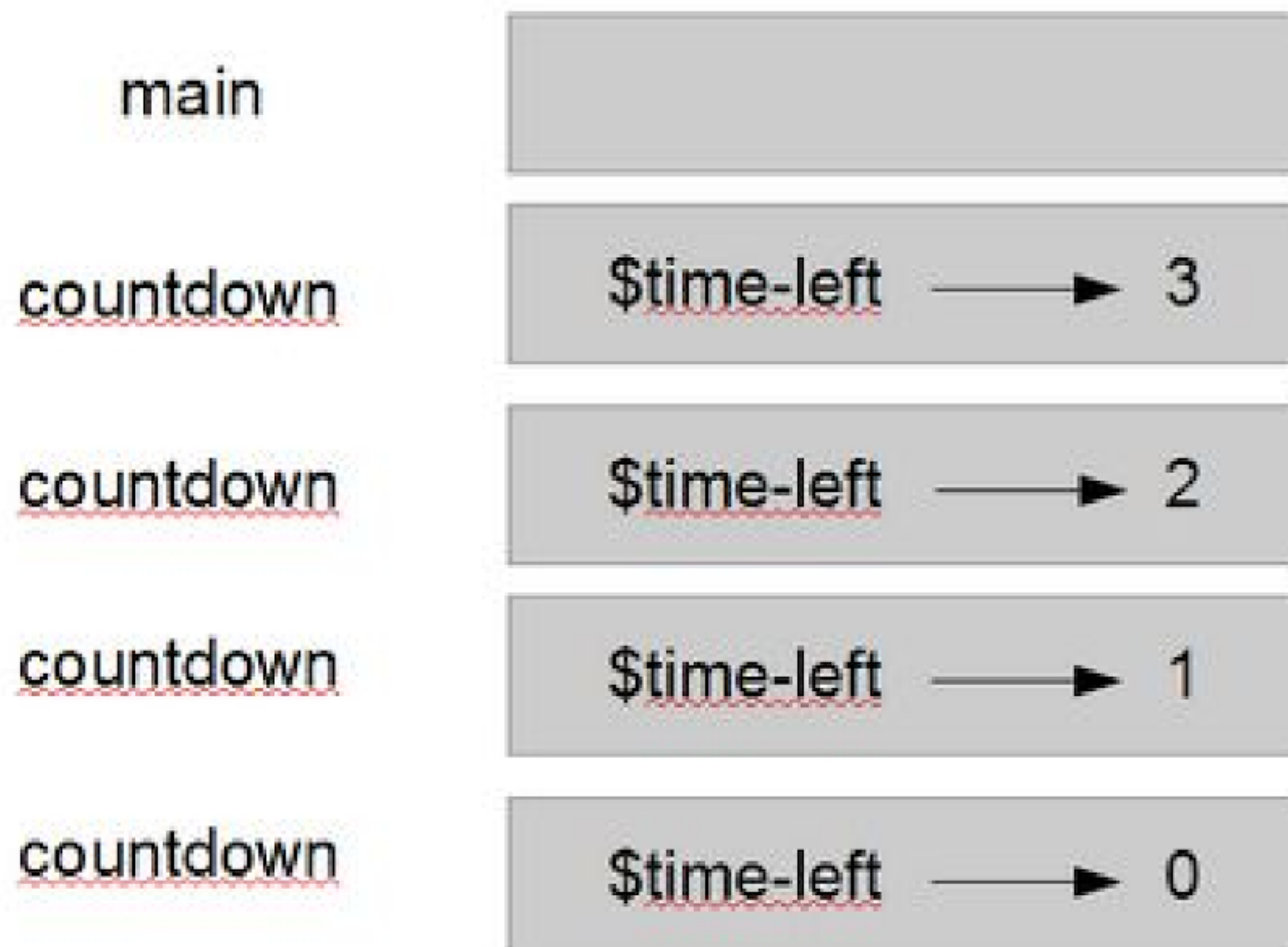


Figure 4-1. Stack diagram

As usual, the top of the stack is the frame for the main program. It is empty because we did not create any variables in it or pass any arguments to it.

The four `countdown` frames have different values for the parameter `$time-left`. The bottom of the stack, where `$time-left = 0`, is called the *base case*. It does not make a recursive call, so there are no more frames.

As an exercise, draw a stack diagram for `print-n-times` called with `$sentence = 'Hello'` and `$n = 2`. Then write a function called `do-n-times` that takes a function and a number, `$num`, as arguments, and that calls the given function `$num` times. Solution: see “Exercises of Chapter 4: Conditionals and Recursion”.

### Infinite Recursion

If a recursion never reaches a base case, it goes on making recursive calls



forever, and the program never terminates. This is known as *infinite recursion*, and it is generally not a good idea. In fact, your program will not actually execute forever but will die at some point when the computer runs out of memory or some other critical resource.

You have to be careful when writing recursive subroutines. Make sure that you have a base case, and make sure that you are guaranteed to reach it. Actually, although this is not absolutely required by the language, I would advise you to make a habit of treating the base case first.

## **Keyboard Input**

The programs we have written so far accept no input from the user. They just do the same thing every time. Perl provides built-in functions that stop the program and wait for the user to type something.

For example, the `prompt` function prompts the user with a question or an instruction. When the user presses Return or Enter, the program resumes and `prompt` returns what the user typed as a string (without the newline character corresponding to the Return key typed by the user):

```
my $user = prompt "Please type in your name: ";  
say "Hello $user";
```

This is probably one of the most common ways to obtain interactive user input, because it is usually a good idea to tell the user what is expected.

Another possibility is to use the `get` method (which reads a single line) on standard input:

```
say "Please type in your name: ";  
my $user = $*IN.get;  
say "Hello $user";
```

or the `get` function, which reads a line from standard input by default:

```
say "Please type in your name: ";  
my $user = get;  
say "Hello $user";
```

## Program Arguments and the MAIN Subroutine

There is another (and often better) way to have a program use varying input defined by the user, which is to pass command-line arguments to the program, just as we have passed arguments to our subroutines.

The easiest way to retrieve arguments passed to a program is to use a special subroutine named `MAIN`. A program that has a defined `MAIN` subroutine will usually start its execution with that subroutine and the command-line arguments supplied to the program will be passed as arguments to `MAIN`. The `MAIN` signature will thus enable you to retrieve the arguments provided in the command line and possibly also check their validity.

For example, the *greet.pl6* program might look like this:

```
sub MAIN (Str $name) {  
    say "Hello $name";  
}
```

You may call this program twice with different command-line arguments as follows:

```
$ perl6 greet.pl6 Larry  
Hello Larry
```

```
$ perl6 greet.pl6 world  
Hello world
```

It is very easy to change the argument, since all you need to do under the operating system command line is use the up arrow and edit the end of the

previous command line.

If you forget to supply the argument (or provide the wrong number of arguments, or arguments not matching the signature), the program will die and Perl 6 will nicely generate and display a usage method:

```
$ perl6 greet.pl6
Usage:
  greet.pl6 <name>
```

## Debugging

When a syntax or runtime error occurs, the error message contains a lot of information, but it can be overwhelming. The most useful parts are usually:

- What kind of error it was
- Where it occurred

Syntax errors are usually easy to find, but there are a few gotchas. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line or even many lines before.

For example, the goal of the following code was to display the multiplication tables:

```
sub multiplication-tables {
  for 1..10 -> $x {
    for 1..10 -> $y {
      say "$x x $y\t= ", $x * $y;
      say "";
    }
  }
}

multiplication-tables();
```

It failed at compilation with the following error:

```
$ perl6 mult_table.pl6
===SORRY!=== Error while compiling /home/Laurent/mult_table.pl6
Missing block (taken by some undeclared routine?)
at /home/Laurent/mult_table.pl6:9
-----> multiplication-tables();<HERE><EOL>
```

The error message reports an error on line 9 of the program (the last line of the code), at the end of the line, but the actual error is a missing closing brace after line 4 and before line 5. The reason for this is that while the programmer made the mistake on line 4, the Perl interpreter could not detect this error before it reached the end of the program. The correct program for displaying multiplication tables might be:

```
sub multiplication-tables {
  for 1..10 -> $x {
    for 1..10 -> $y {
      say "$x x $y\t= ", $x * $y;
    }
    say "";
  }
}
multiplication-tables();
```

When an error is reported on the last line of a program, it is quite commonly due to a missing closing parenthesis, bracket, brace, or quotation mark several lines earlier. An editor with syntax highlighting can sometimes help you.

The same is true of runtime errors. Consider this program aimed at computing 360 degrees divided successively by the integers between 2 and 5:

```
my ($a, $b, $c, $d) = 2, 3, 5;
```

```
my $value = 360;  
$value /= $_ for $a, $b, $c, $d;  
say $value;
```

This program compiles correctly but displays a warning and then an exception on runtime:

```
Use of uninitialized value of type Any in numeric context  
in block at product.pl6 line 3  
Attempt to divide 12 by zero using div  
in block <unit> at product.pl6 line 4
```

The error message indicates a “division by zero” exception on line 4, but there is nothing wrong with that line. The warning on line 3 might give us a clue that the script attempts to use an undefined value, but the real error is on the first line of the script, where one of the four necessary integers (4) was omitted by mistake from the list assignment.

You should take the time to read error messages carefully, but don’t assume they point to the root cause of the exception; they often point to subsequent problems.

## **Glossary**

base case

A conditional branch in a recursive function that does not make a recursive call.

Boolean expression

An expression whose value is either True or False.

branch

One of the alternative sequences of statements in a conditional statement.

chained conditional

A conditional statement with a series of alternative branches.

condition

The Boolean expression in a conditional statement that determines which branch runs.

conditional statement

A statement that controls the flow of execution depending on some condition.

infinite recursion

A recursion that doesn't have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error, for which you may not want to wait because it may take a long time.

integer division

An operation, denoted `div`, that divides two numbers and rounds down (toward zero) the result to an integer.

logical operator

One of the operators that combines Boolean expressions: `and`, `or`, and `not`. The equivalent higher-precedence operators are `&&`, `||`, and `!`.

modulo operator

An operator, denoted with a percent sign (`%`), that works on integers and returns the remainder when one number is divided by another.

nested conditional

A conditional statement that appears in one of the branches of another

conditional statement.

recursion

The process of calling the function that is currently executing.

relational operator

One of the operators that compares its operands. The most common numeric relational operators are `==`, `!=`, `>`, `<`, `>=`, and `<=`. The equivalent string relational operators are `eq`, `ne`, `gt`, `lt`, `ge`, and `le`.

return statement

A statement that causes a function to end immediately and return to the caller.

statement modifier

A postfix conditional expression, i.e., a conditional expression (using for example `if`, `unless`, or `for`) that is placed after the statement the execution of which it controls. It can also refer to a postfix looping expression.

## **Exercises**

### **Exercise 4-1.**

Using the integer division and the modulo operators:

1. Write a subroutine that computes how many days, hours, minutes, and seconds there are in the number of seconds passed as an argument to the subroutine.
2. Write a script that computes how many days, hours, minutes, and seconds there are in 240,000 seconds.
3. Change your script to compute the number of days, hours, minutes, and seconds there are in a number of seconds entered by the script user when

prompted to give a number of seconds.

Solution: “[Exercise 4-1: Days, Hours, Minutes, and Seconds](#)”.

### Exercise 4-2.

Fermat’s Last Theorem says that there are no positive integers  $a$ ,  $b$ , and  $c$  such that

$$a^n + b^n = c^n$$

for any values of  $n$  greater than 2.

1. Write a function named `check-fermat` that takes four parameters— $a$ ,  $b$ ,  $c$ , and  $n$ —and checks to see if Fermat’s theorem holds. If  $n$  is greater than 2 and

$$a^n + b^n = c^n$$

the program should print, “Holy smokes, Fermat was wrong!” Otherwise the program should print, “No, that doesn’t work.”

2. Write a function that prompts the user to input values for  $a$ ,  $b$ ,  $c$ , and  $n$ , converts them to integers, and uses `check-fermat` to check whether they violate Fermat’s theorem.

Solution: “[Exercise 4-2: Fermat’s Theorem](#)”.

### Exercise 4-3.

If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are 1 inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is



possible to form a triangle:

*If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can. (If the sum of two lengths equals the third, they form what is called a “degenerate” triangle.)*

1. Write a function named `is-triangle` that takes three positive numbers as arguments, and that prints either “Yes” or “No,” depending on whether you can form a triangle from sticks with the given lengths.
2. Write a function that prompts the user to input three stick lengths and uses `is-triangle` to check whether sticks with the given lengths can form a triangle.

Solution: “[Exercise 4-3: Is It a Triangle?](#)”.

#### **Exercise 4-4.**

The Fibonacci numbers were invented by Leonardo Fibonacci (a.k.a. Leonardo of Pisa or simply Fibonacci), an Italian mathematician of the thirteenth century.

The Fibonacci numbers are a sequence of numbers such as

**1, 1, 2, 3, 5, 8, 13, 21, 34, ...**

in which the first two numbers are equal to 1 and each subsequent number of the sequence is defined as the sum of the previous two (for example,  $5 = 2 + 3$ ,  $8 = 3 + 5$ , etc.).

In mathematical notation, the Fibonacci numbers could be defined by recurrence as follows:

$$F_1 = 1, \quad F_2 = 1, \text{ and } F_n = F_{n-1} + F_{n-2}$$

1. Write a program using a for loop that prints on screen the first 20 Fibonacci numbers.
2. Write a program which prompts the user to enter a number  $n$  and, using a for loop, computes and displays the  $n$ th Fibonacci number.

Solution: “[Exercise 4-4: The Fibonacci Numbers](#)”.

### Exercise 4-5.

What is the output of the following program? Draw a stack diagram that shows the state of the program when it prints the result.

```
sub recurse($n, $s) {  
    if ($n == 0) {  
        say $s;  
    } else {  
        recurse $n - 1, $n + $s;  
    }  
}  
recurse 3, 0;
```

1. What would happen if you called the function like this: `recurse(-1, 0)`?
2. Write a documentation comment (maybe in the form of a multiline comment) that explains everything someone would need to know in order to use this function (and nothing else).

Solution: “[Exercise 4-5: The recurse Subroutine](#)”.

## Chapter 5. Fruitful Subroutines

Most of the Perl functions we have used, such as the math functions, produce return values. But most of the subroutines we've written so far are void: they have an effect, like printing a value, but they don't have a return value. In this chapter you will learn to write fruitful functions.

### Return Values

Calling a fruitful function generates a return value, which we usually assign to a variable or use as part of an expression:

```
my $pi = 4 * atan 1;
my $height = $radius * sin $radians;
```

Many of the subroutines we have written so far are void. Speaking casually, they have no usable return value; more precisely, their return value may be `Any`, `()`, or `True`.

In this chapter, we are (finally) going to write fruitful subroutines. The first example is `area`, which returns the area of a circle with the given radius:

```
sub area($radius) {
    my $circular_area = pi * $radius**2;
    return $circular_area;
}
```

We have seen the `return` statement before, but in a fruitful function the `return` statement includes an expression. This statement means: “Return immediately from this function and use the following expression as a return value.” The expression can be arbitrarily complicated, so we could have written this function more concisely:

```
sub area($radius) {
    return pi * $radius**2;
}
```

```
}
```

On the other hand, *temporary variables* like `$circular_area` can make debugging easier. They may also help document what is going on.

Sometimes it is useful to have multiple `return` statements, for example one in each branch of a conditional:

```
sub absolute_value($num){  
    if $num < 0 {  
        return -$num;  
    } else {  
        return $num;  
    }  
}
```

Since these `return` statements are in an alternative conditional, only one runs.

This could also be written more concisely using the statement modifier syntax:

```
sub absolute_value($num){  
    return -$num if $num < 0;  
    return $num;  
}
```

Here again, only one of the `return` statements runs: if the number is negative, the first `return` statement is executed and the subroutine execution stops there; if the number is positive or zero, then only the second `return` statement is executed.

As soon as a `return` statement runs, the function terminates without executing any subsequent statements. Code that appears after an unconditional `return` statement, or any other place the flow of execution

can never reach, is called *dead code*.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a `return` statement. For example:

```
sub absolute_value($num){
    if $num < 0 {
        return -$num;
    }
    if $num > 0 {
        return $num;
    }
}
```

This subroutine is incorrect because if `$num` happens to be 0, neither condition is true, and the subroutine ends without hitting a `return` statement. If the flow of execution gets to the end of a function, the return value is `()`, which basically means “not defined” and is clearly not the absolute value of 0:

```
> absolute_value(0)
()
```

By the way, Perl provides a built-in function called `abs` that computes absolute values.

As an exercise, write a `compare` subroutine that takes two numbers, `$x` and `$y`, and returns 1 if `$x > $y`, 0 if `$x == $y`, and -1 if `$x < $y`. Solution: **“Exercise: Compare”**.

## Incremental Development

As you write larger functions, you might find yourself spending more time debugging.

To deal with increasingly complex programs, you might want to try a

process called *incremental development*. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the Cartesian or rectangular coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$ . By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a `distance` function should look like in Perl. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the inputs are two points, which you can represent using four numbers. The return value is the distance represented by a numeric value.

Immediately you can write an outline of the function:

```
sub distance($x1, $y1, $x2, $y2) {  
    return 0.0;  
}
```

Obviously, this version doesn't compute distances; it always returns zero. But it is syntactically correct, and it runs, which means that you can test it before you make it more complicated.

To test the new function, call it with sample arguments:

```
> distance(1, 2, 4, 6);  
0.0
```

I chose these values so that the horizontal distance is 3 and the vertical

distance is 4; that way, the result is 5, the hypotenuse of a 3-4-5 triangle. When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactically correct, and we can start adding code to the body. A reasonable next step is to find the differences  $x_2 - x_1$  and  $y_2 - y_1$ . The next version stores those values in temporary variables and prints them:

```
sub distance($x1, $y1, $x2, $y2) {  
  my $dx = $x2 - $x1;  
  my $dy = $y2 - $y1;  
  say '$dx is', $dx;  
  say '$dy is', $dy;  
  return 0.0;  
}
```

If the function is working, it should display `$dx is 3` and `$dy is 4` (and still return 0.0). If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of `$dx` and `$dy`:

```
sub distance($x1, $y1, $x2, $y2) {  
  my $dx = $x2 - $x1;  
  my $dy = $y2 - $y1;  
  my $dsquared = $dx**2 + $dy**2;  
  say '$dsquared is: ', $dsquared;  
  return 0.0;  
}
```

Again, you would run the program at this stage and check the output (which should be 25). Finally, you can use the `sqrt` built-in function to compute and return the result:

```
sub distance($x1, $y1, $x2, $y2) {  
  my $dx = $x2 - $x1;  
  my $dy = $y2 - $y1;  
  my $dsquared = $dx**2 + $dy**2;  
  my $result = sqrt $dsquared;  
  return $result;  
}
```

If that works correctly, you are done. Otherwise, you might want to print the value of `$result` before the `return` statement.

The final version of the subroutine doesn't display anything when it runs; it only returns a value. The print statements we wrote are useful for debugging, but once you get the function working, you should remove them. Code like that is sometimes called *scaffolding* because it is helpful for building the program but is not part of the final product.

When you start programming, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, incremental development can save you a lot of debugging time.

The key aspects of the process are:

1. Start with a working program and make small incremental changes. At any point, if there is an error, you should have a good idea where it is.
2. Use variables to hold intermediate values so you can display and check them.
3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if doing so does not make the program difficult to read.

Note that, at least for relatively simple cases, you can also use the REPL to



test expressions and even multiline statements or subroutines in interactive mode before you commit them to your program code. This is usually fast and can save you some time.

As an exercise, use incremental development to write a function called `hypotenuse` that returns the length of the hypotenuse of a right triangle given the lengths of the other two legs as arguments. Record each stage of the development process as you go. Solution: **“Exercise: Hypotenuse”**.

## Composition

As you should expect by now, you can call one function from within another. As an example, we’ll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables `$x-c` and `$y-c`, and the perimeter point is in `$x-p` and `$y-p`. The first step is to find the radius of the circle, which is the distance between the two points. We just wrote a function, `distance`, that does that:

```
my $radius = distance($x-c, $y-c, $x-p, $y-p);
```

The next step is to find the area of a circle with that radius; we just wrote that, too:

```
my $result = area($radius);
```

Encapsulating these steps in a function, we get:

```
sub circle-area($x-c, $y-c, $x-p, $y-p) {  
    my $radius = distance($x-c, $y-c, $x-p, $y-p);  
    my $result = area($radius)  
    return $result;  
}
```

The temporary variables `$radius` and `$result` are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
sub circle-area($x-c, $y-c, $x-p, $y-p) {  
    return area distance($x-c, $y-c, $x-p, $y-p);  
}
```

The last line of the previous example now works like a data pipeline from right to left: the `distance` function takes the four arguments and returns a distance (the radius) which is fed as an argument to the `area`; with this argument, `area` is now able to return the area, which is then returned by `circle-area` to the caller code. We'll come back later to this very expressive data pipeline model.

## Boolean Functions

Functions can return Boolean values, which is often convenient for hiding complicated tests inside functions. For example:

```
sub is-divisible(Int $x, Int $y) {  
    if $x % $y == 0 {  
        return True;  
    } else {  
        return False;  
    }  
}
```

It is common to give Boolean functions names that sound like yes/no questions; `is-divisible`, for instance, returns either `True` or `False` to indicate whether `x` is divisible by `y`.

Here is an example:

```
> is-divisible(6, 4);
```

```
False
> is-divisible(6, 3);
True
```

The result of the `==` operator is a Boolean value, so we can write the subroutine more concisely by returning it directly:

```
sub is-divisible(Int $x, Int $y) {
    return $x % $y == 0
}
```

If there is no `return` statement, a Perl subroutine returns the value of expression on the last code line of the subroutine (provided the last code line is an expression that gets evaluated), so that the `return` statement is not required here. In addition, since `0` is a false value and any other integer a true value, this could be further rewritten as follows:

```
sub is-divisible(Int $x, Int $y) {
    not $x % $y
}
```

The `Int` type declarations in the subroutine signatures above are not necessary. The subroutine would work without them, but they can provide some form of protection against using this subroutine with faulty arguments.

Boolean functions are often used in statement modifiers:

```
say "$x is divisible by $y" if is-divisible($x, $y);
```

It might be tempting to write something like:

```
say "$x is divisible by $y" if is-divisible($x, $y) == True;
```

But the extra comparison is unnecessary: `is-divisible` returns a Boolean value that can be interpreted directly by the `if` conditional.

As an exercise, write a function `is-between(x, y, z)` that returns `True` if

$x \leq y \leq z$  or `False` otherwise. Solution:

**“Exercise: Chained Relational Operators”.**

## A Complete Programming Language

We’ve seen in the section above several ways of writing a subroutine to check the divisibility of two integers.

In fact, Perl 6 has a “is divisible” operator, `%%`, which returns `True` if the number on the left is divisible by the one on the right:

```
> 9 %% 3
True
> 9 %% 4
False
```

So there was no need to write the `is-divisible` subroutine. But don’t worry, that’s alright if you did not know that. Speakers of natural languages are allowed to have different skill levels, to learn as they go and to put the language to good use before they know the whole language. The same is true with Perl. You (and I) don’t know all about Perl 6 yet, just as we don’t know all of English. But it is in fact “Officially Okay in Perl Culture” to use the subset of the language that you know. You are in fact encouraged to use what is sometimes called “baby Perl” to write programs, even if they are somewhat clumsy at the beginning. That’s the best way of learning Perl, just as using “baby talk” is the right way for a child to learn English.

The number of different ways of accomplishing a given task, such as checking whether one number is divisible by another, is an example of one of Perl’s mottos: *there is more than one way to do it*, oft abbreviated

TIMTOWTDI. Some ways may be more concise or more efficient than others, but, in the Perl philosophy, you are perfectly entitled to do it your way, especially if you're a beginner, provided you find the correct result.

We have only covered a small subset of Perl 6 so far, but you might be interested to know that this subset is a *complete* programming language, which means that essentially anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features you have learned so far (actually, you would need a few commands to control devices like the mouse, disks, networks, etc., but that's all).

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. For a more complete (and accurate) discussion of the Turing Thesis, I recommend Michael Sipser's book *Introduction to the Theory of Computation* (Cengage Learning).

## **More Recursion**

To give you an idea of what you can do with the tools you have learned so far, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

vorpap

An adjective used to describe something that is vorpap.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the factorial function, denoted with the symbol  $!$ , you might get something like this:

$$0! = 1$$

$$n! = n(n-1)!$$

This definition says that the factorial of 0 is 1, and the factorial of any other (positive integer) value,  $n$ , is  $n$  multiplied by the factorial of  $n - 1$ .

So  $3!$  is 3 times  $2!$ , which is 2 times  $1!$ , which is 1 times  $0!$ . Putting it all together,  $3!$  equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can write a Perl program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that `factorial` takes a number:<sup>1</sup>

```
sub factorial($n){  
}
```

If the argument happens to be 0, all we have to do is return 1:

```
sub factorial($n){  
  if $n == 0 {  
    return 1;  
  }  
}
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of  $n - 1$  and then multiply it by  $n$ :

```
sub factorial($n){  
  if $n == 0 {
```

```

        return 1;
    } else {
        my $recurse = factorial($n-1);
        my $result = $n * $recurse;
        return $result;
    }
}

```

The flow of execution for this program is similar to the flow of `countdown` in “**Recursion**”. If we call `factorial` with the value 3:

Since 3 is not 0, we take the second branch and calculate the factorial of  $n-1$ ...

*Since 2 is not 0, we take the second branch and calculate the factorial of ...*

*Since 1 is not 0, we take the second branch and calculate the factorial of ...*

*Since 0 equals 0, we take the first branch and return 1 without making any more recursive calls.*

*The return value, 1, is multiplied by  $n$ , which is 1, and the result is returned.*

*The return value, 1, is multiplied by  $n$ , which is 2, and the result is returned.*

The return value, 2, is multiplied by  $n$ , which is 3, and the result, 6, becomes the return value of the subroutine call that started the whole process.

**Figure 5-1** shows what the stack diagram looks like for this sequence of function calls.

The return values are shown being passed back up the stack. In each frame, the return value is the value of `result`, which is the product of  $n$  and

recurse.

In the last frame, the local variables `recurse` and `result` do not exist, because the branch that creates them does not run.

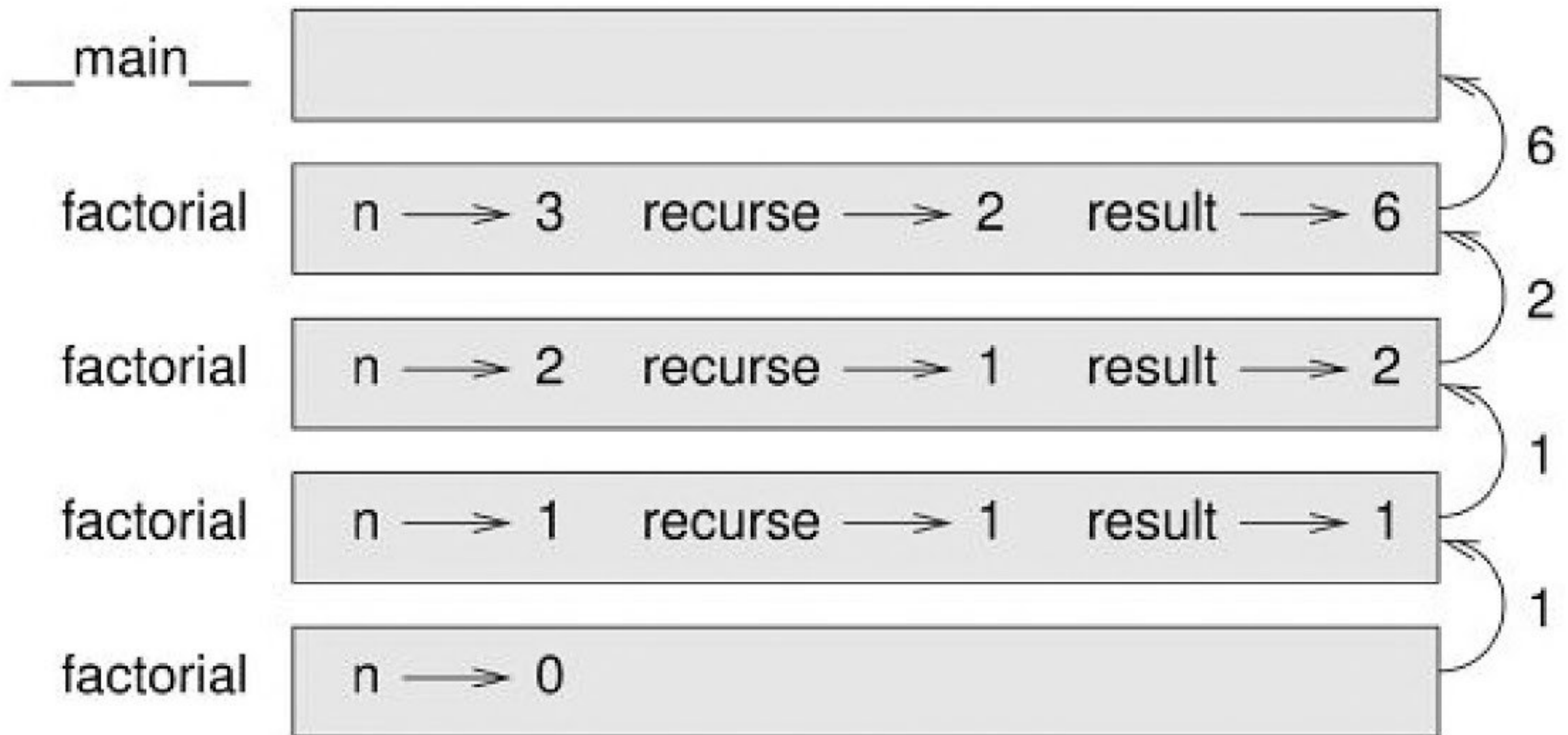


Figure 5-1. Stack diagram

A seasoned Perl programmer might write a more concise or more idiomatic subroutine:<sup>2</sup>

```
sub factorial($n){  
    return 1 if $n == 0;  
    return $n * factorial $n-1;  
}
```

This is not better than our initial version, and will probably not run significantly faster, but this is arguably clearer, at least once you get used to this type of syntax.

## Leap of Faith

Following the flow of execution is one way to read programs, but it can quickly become overwhelming. An alternative is what may be called the



“leap of faith.” When you come to a subroutine call, instead of following the flow of execution, you *assume* that the subroutine works correctly and returns the right result.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call math functions such as `cos` or `sqrt`, you don’t examine the bodies of those functions. You just assume that they work because the people who wrote the built-in functions were likely to be good programmers (and because you can safely assume that they have been thoroughly tested).

The same is true when you call one of your own subroutines. For example, in “[Boolean Functions](#)”, we wrote a subroutine called `is-divisible` that determines whether one number is divisible by another. Once we have convinced ourselves that this subroutine is correct—by examining the code and testing—we can use the subroutine without looking at the body again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (returns the correct result) and then ask yourself, “Assuming that I can find the factorial of  $n-1$ , can I compute the factorial of  $n$ ?” It is clear that you can, by multiplying by  $n$ .

Of course, it’s a bit strange to assume that the subroutine works correctly when you haven’t finished writing it, but that’s why it’s called a leap of faith!

## One More Example

After `factorial`, the most common example of a recursively defined mathematical function is `fibonacci`, which has the following definition (see also the [Wikipedia entry](#)):

$$\text{fibonacci}(0) = 1$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

In plain English, a Fibonacci sequence is a sequence of numbers such as:

1, 1, 2, 3, 5, 8, 13, 21, ...

where the two first terms are equal to 1 and any other term is the sum of the two preceding ones.

We briefly covered the Fibonacci sequence in [Exercise 4-4](#) and implemented it with a `for` loop. Let's now translate the recursive definition into Perl. It looks like this:

```
sub fibonacci ($n) {  
    return 1 if $n == 0 or $n == 1;  
    return fibonacci($n-1) + fibonacci($n-2);  
}
```

If you try to follow the flow of execution here, even for fairly small values of `$n`, your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

## Checking Types

What happens if we call `factorial` and give it 1.5 as an argument?

It seems that we get an infinite recursion. How can that be? The subroutine has a base case—when `$n == 0`. But if `$n` is not an integer, we can *miss* the base case and recurse forever.

In the first recursive call, the value of `$n` is 0.5. In the next, it is -0.5. From

there, it gets smaller (more negative), but it will never be 0.

We have two choices. We can try to generalize the `factorial` function to work with noninteger numbers, or we can make `factorial` check its argument. The first option is called the *gamma function*, and it's a little beyond the scope of this book. So we'll go for the second.

We have already seen examples of subroutines using the signature to verify the type of the argument. So we can add the `Int` type to the parameter in the signature. While we're at it, we can also make sure the argument is positive or zero.

```
sub factorial(Int $n where $n >= 0){
  return 1 if $n == 0;
  return $n * factorial $n-1;
}
```

The `Int` type checking in the signature handles nonintegers; this is not new. The `where $n >= 0` part is a parameter constraint: if the parameter is negative, the subroutine should fail. Technically, the constraint is implemented here within the signature using a syntax feature called a *trait*, a property imposed on the parameter at compile time. If the argument passed to the function is not an integer or if it is negative, the program prints an error message to indicate that something went wrong:

```
> say factorial 1.5
Type check failed in binding $n; expected Int but got Rat
  in sub factorial at <unknown file> line 1
  in block <unit> at <unknown file> line 1
```

```
> say factorial -3
Constraint type check failed for parameter '$n'
> say factorial "Fred"
Type check failed in binding $n; expected Int but got Str
  in sub factorial at <unknown file> line 1
  in block <unit> at <unknown file> line 1
```

If we get past both checks, we know that  $n$  is an integer and that it is positive or zero, so we can prove that the recursion terminates.

Another way to achieve a similar result is to define your own subset of the built-in types. For example, you can create an `Even-int` subset of integers and then use it more or less as if it were a new type for declaring your variables or typing your subroutine parameters:

```
subset Even-int of Int where { $_ %% 2 } # or : ... where { $_ % 2
== 0 }
# Even-int can now be used as a type

my Even-int $x = 2; # OK
my Even-int $y = 3; # Type mismatch error
```

Similarly, in the case of the `factorial` subroutine, we can create a *nonnegative integer* subset and use it for checking the parameter passed to the subroutine:

```
subset Non-neg-int of Int where { $_ >= 0 }
# ...

sub factorial(Non-neg-int $n){
    return 1 if $n == 0;
    return $n * factorial $n-1;
}
```

If we pass a negative integer to the subroutine, we get a similar error as before:

```
Constraint type check failed for parameter '$n'...
```

This program demonstrates a pattern sometimes called a *guardian*. The signature acts as a guardian, protecting the code that follows from values

that might cause an error. The guardians make it possible to prove the correctness of the code.

## Multi Subroutines

It is possible to write multiple versions of a subroutine with the same name but with different signatures, for example a different *arity* (a fancy word for the number of arguments) or different argument types, using the `multi` keyword. In this case, the interpreter will pick the version of the subroutine whose signature matches (or best matches) the argument list.

For example, we could rewrite the factorial function as follows:

```
multi sub fact(0) { 1 };
multi sub fact(Int $n where $n > 0) {
    $n * fact $n - 1;
}
say fact 0;    # -> 1
say fact 10;  # -> 3628800
```

Here, we don't enter into infinite recursion because, when the parameter passed to `fact` is 0, it is the first version of the multi subroutine that is called and it returns an integer value (1), and this ends the recursion.

Similarly, the Fibonacci function can be rewritten with multi subroutines:

```
multi fibonacci(0) { 0 }
multi fibonacci(1) { 1 }
multi fibonacci(Int $n where $n > 1) {
    fibonacci($n - 2) + fibonacci($n - 1)
}
say fibonacci 10;  # -> 55
```

Many built-in functions and most operators of Perl 6 are written as multi subroutines.

## Debugging

Breaking a large program into smaller functions or subroutines creates natural checkpoints for debugging. If a subroutine is not working, there are three possibilities to consider:

- There is something wrong with the arguments the subroutine is getting; a precondition is violated.
- There is something wrong with the subroutine; a postcondition is violated.
- There is something wrong with the return value or the way it is being used.

To rule out the first possibility, you can add a print statement at the beginning of the function and display the values of the parameters (and maybe their types). Or you can write code that checks the preconditions explicitly.

For the purpose of debugging, it is often useful to print the content of a variable or of a parameter within a string with surrounding characters, so that you may visualize characters that are otherwise invisible, such as spaces or newlines. For example, you think that the `$var` should contain “two,” and run the following test:

```
if $var eq "two" {  
    do-something()  
}
```

But it fails and the `do-something` subroutine is never called.

Perhaps you want to use a print statement that will ascertain the content of `$var`:

```
say "[$var]";
```

```
if $var eq "two" {  
    do-something()  
}
```

This might print:

```
[two ]
```

or:

```
[two  
]
```

Now, you know that the equality test fails because `$var` contains a trailing character (space or newline) that might otherwise be difficult to detect.

If the parameters look good, add a print statement before each `return` statement and display the return value. If possible, check the result by hand. Consider calling the function with values that make it easy to check the result (as in [“Incremental Development”](#)).

If the function seems to be working, look at the function call to make sure the return value is being used correctly (or used at all!).

Adding print statements at the beginning and end of a function can help make the flow of execution more visible. For example, here is a version of `factorial` with print statements:

```
sub factorial(Int $n) {  
    my $space = ' ' x (4 * $n);  
    say $space, 'factorial ', $n;  
    if $n == 0 {  
        say $space, 'returning 1';  
        return 1;  
    } else {
```

```

    my $result = $n * factorial $n-1;
    say $space, 'returning ', $result;
    return $result;
  }
}

```

The `$space` variable is a string of space characters that controls the indentation of the output. Here is the result of `factorial(4)` :

```

                factorial 4
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
                returning 24

```

If you are confused about the flow of execution, this kind of output can be helpful. It takes some time to develop effective scaffolding, but a bit of scaffolding can save a lot of debugging.

## **Glossary**

dead code

Part of a program that can never run, often because it appears after a return statement.

guardian

A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

incremental development

A program development plan intended to avoid debugging by adding