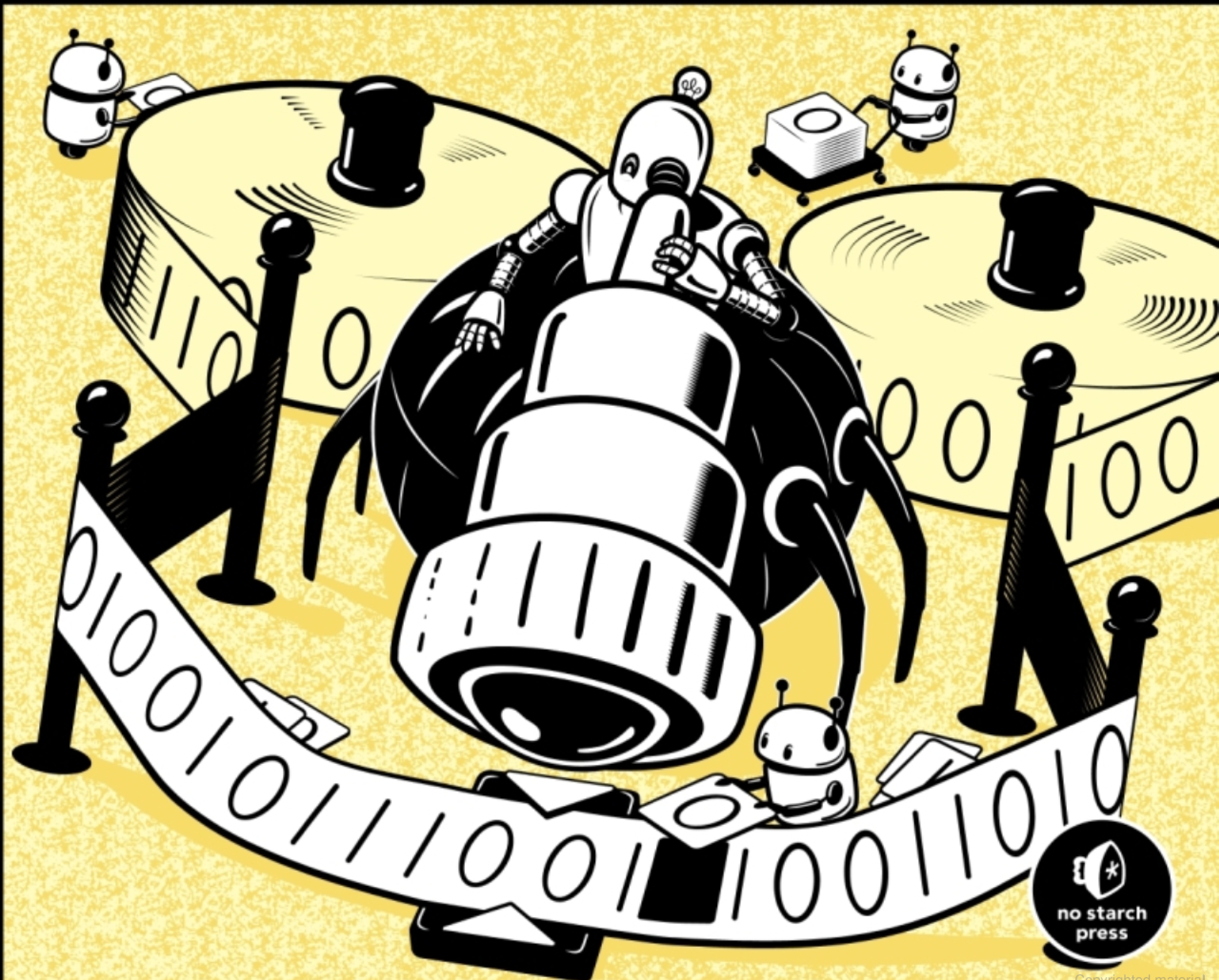
RACKET PROGRAMMING THE FUN WAY

FROM STRINGS TO TURING MACHINES

JAMES W. STELLY



Copyrighted material

RACKET PROGRAMMING THE FUN WAY

From Strings to Turing Machines

by James W. Stelly

no starch press

San Francisco

Copyrighted material

RACKET PROGRAMMING THE FUN WAY. Copyright © 2021 by James W. Stelly.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-7185-0082-2 (print)
ISBN-13: 978-1-7185-0083-9 (ebook)

Publisher: William Pollock
Executive Editor: Barbara Yien
Production Editor: Dapinder Dosanjh
Developmental Editor: Alex Freed
Interior Design: Octopod Studios
Cover Illustration: Gina Redman
Technical Reviewer: Matthew Flatt
Copyeditor: Chris Cartwright
Proofreader: Emelie Battaglia

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com
www.nostarch.com

Library of Congress Cataloging-in-Publication Data
Names: Stelly, James W., author.
Title: Racket programming the fun way: from strings to turing machines / by James W. Stelly.
Description: San Francisco : No Starch Press, [2021]. | Includes
bibliographical references and index.
Identifiers: LCCN 2020022884 (print) | LCCN 2020022885 (ebook) | ISBN
9781718500822 | ISBN 9781718500839 (ebook) | ISBN 1718500822
Subjects: LCSH: Racket (Computer program language) | LISP (Computer program
language) | Computer programming.
Classification: LCC QA76.73.R33 S 2020 (print) | LCC QA76.73.R33 (ebook)
| DDC 005.13/3-dc23
LC record available at https://lccn.loc.gov/2020022884
LC ebook record available at https://lccn.loc.gov/2020022885

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

Copyrighted material

BRIEF CONTENTS

Acknowledgments . . . xvii
Introduction . . . xix
Chapter 1: Racket Basics . . . 1
Chapter 2: Arithmetic and Other Numerical Paraphernalia . . . 27
Chapter 3: Function Fundamentals . . . 41
Chapter 4: Plotting, Drawing, and a Bit of Set Theory . . . 75
Chapter 5: GUI: Getting Users Interested . . . 117
Chapter 6: Data . . . 145
Chapter 7: Searching for Answers . . . 179
Chapter 8: Logic Programming . . . 227
Chapter 9: Computing Machines . . . 259
Chapter 10: TRAC: The Racket Algebraic Calculator . . . 275
Appendix A: Number Bases . . . 317
Appendix B: Special Symbols . . . 321
Bibliography . . . 325
Index . . . 327

Copyrighted material

Copyrighted material

CONTENTS IN DETAIL

ACKNOWLEDGMENTS xvii

INTRODUCTION xix

1
RACKET BASICS 1

Atomic Data 1
Lists 2
A First Look at Lists 2
S-Expressions 3
List Structure 4
A Few Useful List Functions 6
Defines, Assigns, and Variables 8
Symbols, Identifiers, and Keywords 10
Equality 11
Strings and Things 12
Characters 12
Useful String Functions 14
String Conversion and Formatting Functions 17
Vectors 18
Accessing Vector Elements 19
Useful Vector Functions 20
Using structs 21
Controlling Output 24
Summary 26

2
ARITHMETIC AND OTHER NUMERICAL PARAPHERNALIA 27

Booleans 27
The Numerical Tower 29
Integers 29
Rationals 30
Reals 31
Complex Numbers 32
Numeric Comparison 33
Combining Data Types 34
Built-in Functions 36

Copyrighted material

Infix Notation . . . 37
Summary . . . 39

3
FUNCTION FUNDAMENTALS 41

What Is a Function? . . . 41
Lambda Functions . . . 42
Higher-Order Functions . . . 43
Lexical Scoping . . . 45
Conditional Expressions: It's All About Choices . . . 47
I'm Feeling a Bit Loopy! . . . 48
Purity . . . 49
The Power of the Dark Side . . . 51
The for Family . . . 52
Time for Some Closure . . . 58
Applications . . . 60
I Don't Have a Queue . . . 60
The Tower of Hanoi . . . 64
Fibonacci and Friends . . . 66
The Insurance Salesman Problem . . . 72
Summary . . . 74

4
PLOTTING, DRAWING, AND A BIT OF SET THEORY 75

Plotting . . . 75
X-Y Plots . . . 76
Parametric Plots . . . 79
Getting to the Point . . . 86
Polar Plots . . . 88
Drawing . . . 92
Set Theory . . . 96
The Basics . . . 96
A Short Mathematical Detour . . . 100
Drawing Conclusions . . . 101
Are We Related? . . . 103
Applications . . . 105
Fibonacci Revisited . . . 105
Nim . . . 109
Summary . . . 115

5
GUI: GETTING USERS INTERESTED 117

Introduction to GUIs . . . 118
Animating a Cycloid . . . 121

Copyrighted material

Pick a Card . . . 124
GUI Layout . . . 127
Building the Controls . . . 128
Control Logic . . . 130
Linear Algebra Zone . . . 130
Wrapping Up the GUI . . . 131
Control Tower . . . 133
Setting Up . . . 133
Row 1 Widgets . . . 135
Row 2 Widgets . . . 137
Getting in Position . . . 138
Controlling the Animation . . . 140
Wrapping Things Up . . . 143
Summary . . . 144

6
DATA 145

I/O, I/O, It's Off to Work We Go . . . 145
File I/O Ports . . . 145
String Ports . . . 147
Computer-to-Computer Ports . . . 148
Introduction to Security . . . 149
Getting Data into Racket . . . 150
A Database Detour . . . 155
Data Visualization . . . 159
Plotting for Success . . . 161
Lumping Things Together . . . 167
A Bit of Statistics . . . 171
Standard Deviation . . . 171
Regression . . . 173
Summary . . . 178

7
SEARCHING FOR ANSWERS 179

Graph Theory . . . 180
The Basics . . . 180
Graph Search . . . 182
The *N*-Queens Problem . . . 183
A Racket Solution . . . 185
Dijkstra's Shortest Path Algorithm . . . 189
The Priority Queue . . . 191
The Implementation . . . 194

Copyrighted material

The 15 Puzzle 198
The A* Search Algorithm 200
The 8-Puzzle in Racket 204
Moving Up to the 15 Puzzle 209
Sudoku 215
Summary 225

8
LOGIC PROGRAMMING **227**

Introduction 228
The Basics 230
Knowing Your Relatives 230
Racklog Predicates 234
Racklog Utilities 241
Applications 245
SEND + MORE = MONEY 245
Fox, Goose, Beans 246
How Many Donuts? 252
Boles and Creots 254
Summary 258

9
COMPUTING MACHINES **259**

Finite-State Automata 259
The Turing Machine 263
A Racket Turing Machine 265
Pushdown Automata 267
Recognizing Zeros and Ones 268
More Zeros and Ones 269
A Racket PDA 270
More Automata Fun 272
A Few Words About Languages 272
Summary 273

10
TRAC: THE RACKET ALGEBRAIC CALCULATOR **275**

The TRAC Pipeline 276
The Lexical Analyzer 277
Regular Expressions 279
Regular Expressions in Racket 280
Regular Expressions in TRAC 284
The Lexer 286

Copyrighted material

The Parser 288
TRAC Grammar Specification 288
The TRAC Parser 291
TRAC 299
Adding a Dictionary 299
A Few Enhancements 301
Making Sure TRAC Works Properly 309
Making an Executable 314
Summary 315

A
NUMBER BASES 317

B
SPECIAL SYMBOLS 321

BIBLIOGRAPHY 325

INDEX 327

Copyrighted material

Copyrighted material

ACKNOWLEDGMENTS

First, let me thank the folks responsible for Racket. They have created a truly remarkable piece of software. The care and effort that went into producing it has to be incalculable.

I would like to extend my deepest gratitude and thanks to my editors at No Starch, Alex Freed and Athabasca Witschi, as well as my technical reviewer, Matthew Flatt. They made literally dozens of helpful suggestions for improvements as well as corrections. Any remaining flaws are entirely of my own making.

Finally, let me thank my wife for her patience. Her view of me was primarily of the back of my head while I was working on this book.

Copyrighted material

Copyrighted material

INTRODUCTION

In this book we explore using Racket (a language descended from the Scheme family of programming languages—which in turn descended from Lisp) and DrRacket, a graphical environment that allows us to make the most of all the features of Racket. One of the attractive features of this ecosystem is that it's equipped with a plethora of libraries that cover a wide range of disciplines. The developers describe Racket as a system that has "batteries included." This makes it an ideal platform for the interactive investigation of various topics in computer science and mathematics.

Given Racket's Lisp pedigree, we would be remiss to omit functional programming, so we will definitely explore it in this text. Racket is no one-trick pony though, so we will also explore imperative, object oriented, and logic programming along the way. Also on the computer science front, we will look at various abstract computing machines, data structures, and a number of search algorithms as related to solving some problems in recreational mathematics. We will finish the book by building our own calcula-

Copyrighted material

tor, which will entail lexical analysis using regular expressions, defining the grammar using extended Backus–Naur form (EBNF), and building a recursive descent parser.

Racket

Racket features extensive and well-written documentation, which includes *Quick: An Introduction to Racket with Pictures*, the introductory *Racket Guide*, and the thorough *Racket Reference*. Various other toolkits and environments also have separate documentation. Within DrRacket these items can be accessed through the Help menu.

Racket is available for a wide variety of platforms: Windows, Linux, macOS, and Unix. It can be downloaded from the Racket website via the link *https://download.racket-lang.org/*. Once downloaded, installation simply entails running the downloaded executable on Windows, *.dmg* file on macOS, or shell script on Linux. At the time of writing, the current version is 7.8. Examples in the book will run on any version 7.0 or later. They will likely run on earlier versions as well, but since the current version is freely available there is really no need to do so. When the DrRacket environment is first launched, the user will be prompted to select a Racket language variant. The examples in this book all use the first option in the pop-up dialog box (that is, the one that says "The Racket Language").

The DrRacket window provides a definitions pane (top pane in Figure 1) where variables and functions are defined and an interactions pane (bottom pane in Figure 1) where Racket code can be interactively executed. Within these panes, help is a single keypress away. Just click on any built-in function name and press F1.

The definitions window contains all the features one expects from a robust interactive development environment (IDE) such as syntax highlighting, variable renaming, and an integrated debugger.

Racket enthusiasts are affectionately known as *Racketeers* (catchy, eh?). Once you've had an opportunity to explore this wonderful environment, don't be surprised if you become a Racketeer yourself.

Copyrighted material

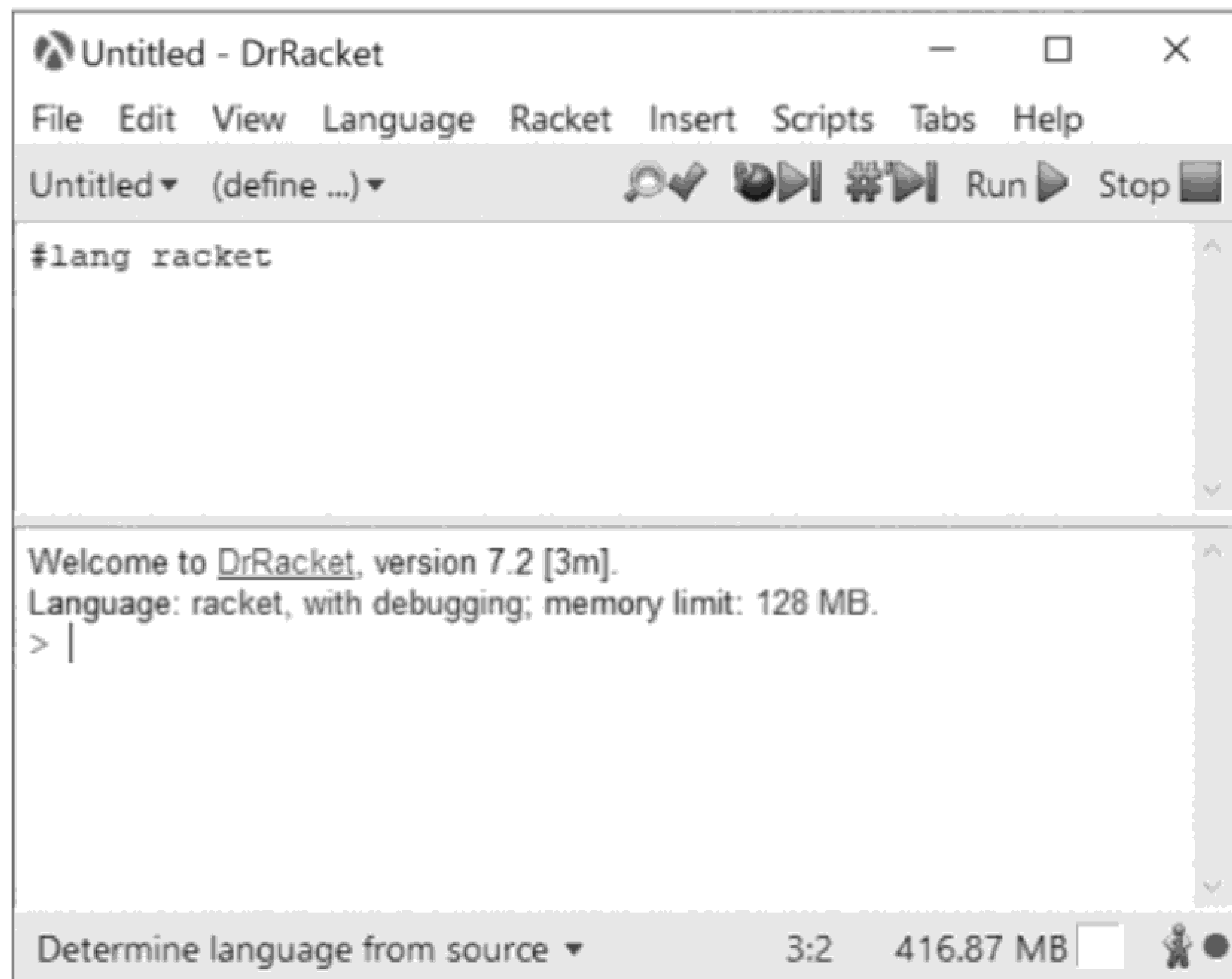


Figure 1: DrRacket IDE

Conventions Used in This Book

DrRacket supports a number of programming and learning languages. In this book we focus exclusively on the default Racket language. Thus, unless otherwise stated, all definition files should begin with the line

```
#lang racket
```

Code entered in the definitions section will be shown in a framed box as above.

Expressions entered in the interactive pane will be shown prefixed with a right angle bracket > as shown below. The angle bracket is DrRacket's input prompt. Outputs will be shown without the angle bracket. To easily differentiate inputs and outputs, inputs will be shown in bold in this book (but they are not bold in the IDE).

```
> (+ 1 2 3) ; this is an input, the following is an output
6
```


Copyrighted material

We occasionally make use of some special symbols that DrRacket supports, such as the Greek alphabet (for example, we may use θ as an identifier for an angle). These symbols are listed in Appendix B. The method used to enter these symbols is also given there. If you're typing the examples in by hand and don't want to use the special symbols, simply substitute a name of your choosing: for example use `alpha` instead of α.

An example of a program listing entered in the definitions window is shown below.

```
#lang racket

(define (piscis x y r b)
  (let* ([y (- y r)]
         [2r (* 2 r)]
         [yi (sqrt (- (sqr r) (sqr x)))] ; y-intersection
         [π pi]
       ❶ [ϕ (asin (/ yi r))]
       ❷ [θ (- π ϕ)]
       ❸ [path (new dc-path%)])
    (send dc set-brush b)
  ❹ (send path move-to 0 (- yi))
  ❺ (send path arc (- x r)     y 2r 2r  θ    (+ π  ϕ))
  ❻ (send path arc (- (- x) r) y 2r 2r (- ϕ) ϕ)
  ❼ (send dc draw-path path)))
```

We'll use Wingdings symbols such as ❶ to highlight interesting portions of the code.

Who This Book Is For

While no prior knowledge of Racket, Lisp, or Scheme is required, it wouldn't hurt to have some basic programming knowledge, but this is certainly not required. The mathematical prerequisites will vary. Some topics may be a bit challenging, but nothing more than high school algebra and trigonometry is assumed. A theorem or two may surface, but the treatment will be informal.

About This Book

If you're already familiar with the Racket language, feel free to skip (or perhaps just skim) the first couple of chapters as these just provide an introduction to the language. These early chapters are by no means a comprehensive encyclopedia of Racket functionality. The ambitious reader should consult the excellent Racket Documentation for fuller details. Here is a brief description of each chapter's content.

Chapter 1: Racket Basics Gives the novice Racket user a grounding in some of the basic Racket concepts that will be needed to progress through the rest of the book.

Copyrighted material

Chapter 2: Arithmetic and Other Numerical Paraphernalia Describes Racket's extensive set of numeric data types: integers, true rational numbers, and complex numbers (to name a few). This chapter will make the reader adept at using these entities in Racket.

Chapter 3: Function Fundamentals Introduces Racket's multi-paradigm programming capability. This chapter introduces the reader to both functional and imperative programming. The final section will look at a few fun programming applications.

Chapter 4: Plotting, Drawing, and a Bit of Set Theory Introduces interactive graphics. Most IDEs are textual only; DrRacket has extensive capability for generating graphical output in an interactive environment. This chapter will show you how it's done.

Chapter 5: GUI: Getting Users Interested Shows how to construct mini graphics applications that run in their own window.

Chapter 6: Data Explores various ways of handling data in Racket. It will discuss how to read and write data to and from files on your computer. It will also discuss ways to analyze data using statistics and data visualization.

Chapter 7: Searching for Answers Examines a number of powerful search algorithms. These algorithms will be used to solve various problems and puzzles in recreational mathematics.

Chapter 8: Logic Programming Takes a look at another powerful programming paradigm. Here we explore using Racket's Prolog-like logic programming library: Racklog.

Chapter 9: Computing Machines Takes a quick look at various abstract computing machines. These simple mechanisms are a gateway into some fairly deep concepts in computer science.

Chapter 10: TRAC: The Racket Algebraic Calculator Leverages skills developed in the previous chapters to build a stand-alone interactive command line calculator.

Copyrighted material

Copyrighted material

1

RACKET BASICS

Let's begin with an introduction to some basic concepts in Racket. In this chapter, we'll cover some of the fundamental data types that will be used throughout the book. You'll want to pay particular attention to the discussion of lists, which underpin much of Racket's functionality. We'll also cover how to assign values to variables and various ways to manipulate strings, and along the way, you'll encounter a first look at vectors and structs. The chapter wraps up with a discussion on how to produce formatted output.

Atomic Data

Atomic data is the basic building block of any programming language, and Racket is no exception. Atomic data refers to elementary data types that are typically considered to be indivisible entities; that is, numbers like `123`, strings like `"hello there"`, and identifiers such as `pi`. Numbers and strings

Copyrighted material

evaluate to themselves; if bound, identifiers evaluate to their associated value:

```
> 123
123

> "hello there"
"hello there"

> pi
3.141592653589793
```

Evaluating an unbound identifier results in an error. To prevent an unbound identifier from being evaluated, you can prefix it with an apostrophe:

```
> alpha
. . alpha: undefined;
  cannot reference an identifier before its definition

> 'alpha
'alpha
```

We can organize atomic data together using lists, which are covered next.

Lists

In Racket, lists are the primary non-atomic data structures (that is, something other than a number, string, and so on). Racket relies heavily on lists because it's a descendant of *Lisp* (short for LISt Processing). Before we get into the details, let's look at some simple representative samples.

A First Look at Lists

Here's how to make a list with some numbers:

```
> (list 1 2 3)
```

Notice the syntax. Lists typically begin with an open parenthesis, (, followed by a list of space-separated items and end with a closed parenthesis,). The first item in the list is normally an identifier that indicates how the list is to be evaluated.

Lists can also contain other lists.

```
> (list 1 (list "two" "three") 4 5)
```

which prints as

```
'(1 ("two" "three") 4 5)
```


Copyrighted material

Note the apostrophe (or tick mark) at the beginning of the last example. This is an alias for the `quote` keyword. If you want to enter a literal list (a list that is simply accepted as is), you can enter it *quoted*:

```
> (quote (1 ("two" "three") 4 5))
```

or

```
> '(1 ("two" "three") 4 5)
```

Either of which print as

```
'(1 ("two" "three") 4 5)
```

While `list` and `quote` seem like two equivalent ways to build lists, there's an important difference between them. The following sequence illustrates the difference.

```
> (quote (3 1 4 pi))
'(3 1 4 pi)

> (list 3 1 4 pi)
'(3 1 4 3.141592653589793)
```

Notice that `quote` returns the list exactly as it was entered, but when `list` was used, the identifier `pi` was evaluated and its value was substituted in its place. In general, in a non-quoted list, *all* identifiers are evaluated and replaced by their associated values. The keyword `quote` plays an important role in macros and symbolic expression evaluation, which are advanced topics that we will not cover in this text.

One criticism of the Lisp family of languages is the proliferation of parentheses. To alleviate this, Racket allows either square brackets or curly brackets to be used instead. For example, it's perfectly acceptable to write the last expression as

```
> '(1 ["two" "three"] 4 5)
```

or

```
> '(1 {"two" "three"} 4 5)
```

S-Expressions

A list is a special case of something called an *s-expression*. An s-expression (or symbolic expression) is defined as being one of two cases:

Case 1 The s-expression is an atom.

Case 2 The s-expression is expression of the form `(x . y)` where *x* and *y* are other s-expressions.

Copyrighted material

The form `(x . y)` is typically called a *pair*. This is a special syntactic form used to designate a *cons* cell, which we will have much more to say about shortly.

Let's see if we can construct a few examples of s-expressions. Ah, how about `1`? Yes, it's an atom, so it satisfies case 1. What about `"spud"`? Yep, strings are atoms, and thus `"spud"` is also an s-expression. We can combine these to make another s-expression: `(1 . "spud")`, which satisfies case 2. Since `(1 . "spud")` is an s-expression, case 2 allows us to form another s-expression as `((1 . "spud") . (1 . "spud"))`. We can see from this that s-expressions are actually tree-like structures as illustrated in Figure 1-1. (Technically s-expressions form a *binary tree*, where non-leaf nodes have exactly two child nodes).

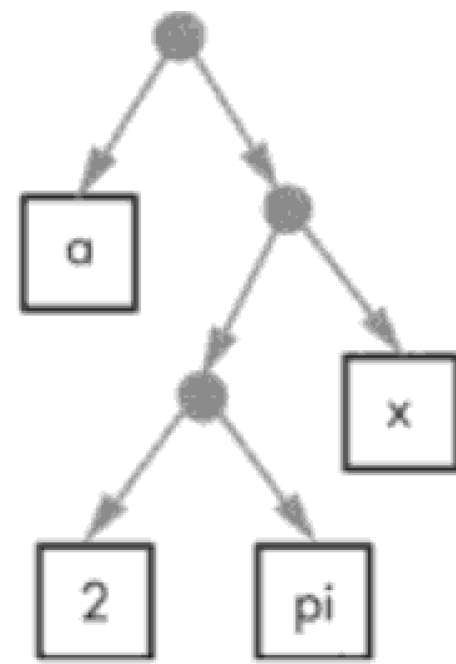


Figure 1-1: `((a . (2 . pi) . x))`

In Figure 1-1, the square boxes are leaf nodes representing atoms, and the circle nodes represent pairs. We'll see how s-expressions are used to construct lists in the next section.

List Structure

As mentioned above, a list is a special case of an s-expression. The difference is that, in a list, if we follow the rightmost elements in each pair, the final node is a special atomic node called *nil*. Figure 1-2 illustrates what the list `'(1 2 3)`—which as an s-expression is `(1 . (2 . (3 . nil)))`—looks like internally.

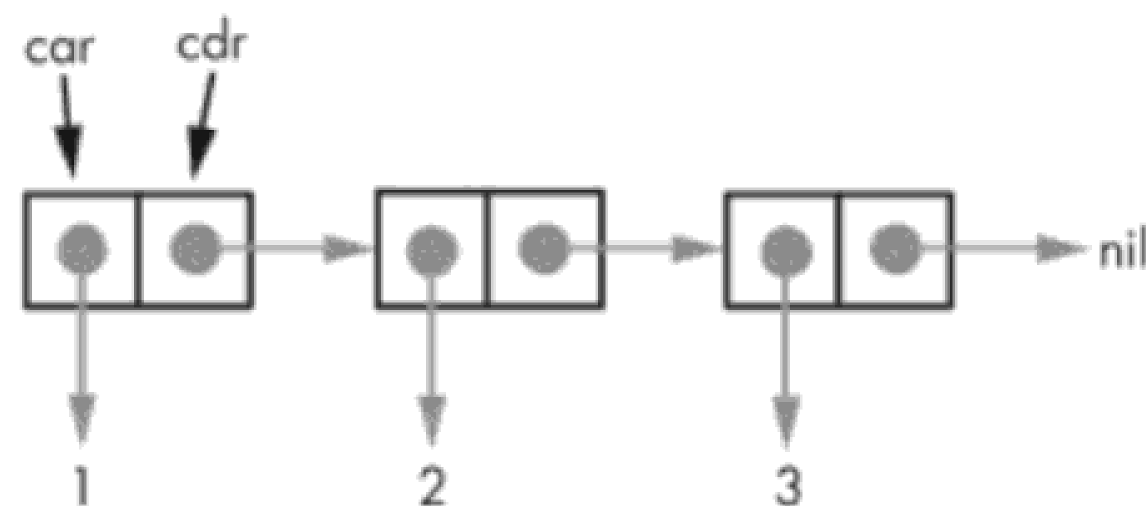


Figure 1-2: List structure

We've flattened the tree to better resemble a list. We've also expanded each pair node (aka a *cons cell*) to show that it consists of two cells, each of which contains a pointer to another node. These pointer cells, for historical reasons, are called *car* and *cdr* respectively (the names of computer registers

Copyrighted material

used in early versions of Lisp). We can see that the last cdr cell in the list is pointing to nil. Nil is indicated in Racket by an empty list: `'()` or `null`.

Cons cells can be created directly by using the `cons` function. Note that the `cons` function does not necessarily create a list. For example

```
> (cons 1 2)
'(1 . 2)
```

produces a pair but *not* a list. However, if we use an empty list as our second s-expression

```
> (cons 1 '())
'(1)
```

we produce a list with just one element.

Racket provides a couple of functions to test whether something is a list or a pair. Note in Racket `#t` means true and `#f` means false:

```
> (pair? (cons 1 2))
#t

> (list? (cons 1 2))
#f

> (pair? (cons 1 '()))
#t

> (list? (cons 1 '()))
#t
```

From this we can see that a list is always a pair, but the converse is not always true: a pair is not always a list.

Typically, `cons` is used to add an atomic value to the beginning of a list, like so:

```
> (cons 1 '(2 3))
'(1 2 3)
```

Racket provides special functions to access the components of a cons cell. The function `car` returns the item being pointed to by the car pointer, and correspondingly the `cdr` function returns the item being pointed to by the cdr pointer. In Racket the functions `first` and `rest` are similar to `car` and `cdr` but are not aliases for these functions, since they only work with lists. A few examples are given below.

```
> (car '(1 ("two" "three") 4 5))
1

> (first '(1 ("two" "three") 4 5))
1
```


Copyrighted material

```
> (cdr '(1 ("two" "three") 4 5))
'(("two" "three") 4 5)

> (rest '(1 ("two" "three") 4 5))
'(("two" "three") 4 5)
```

List elements can also be accessed with the functions `second`, `third`, . . . , `tenth`.

```
> (first '(1 2 3 4))
1

> (second '(1 2 3 4))
2

> (third '(1 2 3 4))
3
```

Finally, a value at any position can be extracted by using `list-ref`.

```
> (list-ref '(a b c) 0)
'a

> (list-ref '(a b c) 1)
'b
```

The `list-ref` function takes a list and the index of the value you want, with the list coming first. Notice that Racket uses *zero-based indexes*, meaning for any sequence of values, the first value has an index of 0, the second value has an index of 1, and so on.

A Few Useful List Functions

Let's quickly go through a number of useful list functions.

length

To get the length of a list, you can use the `length` function, like so:

```
> (length '(1 2 3 4 5))
5
```

reverse

If you need the elements in a list reversed, you can use the reverse function.

```
> (reverse '(1 2 3 4 5)) ; reverse elements of a list
'(5 4 3 2 1)
```


Copyrighted material

sort

The sort function will sort a list. You can pass in < to sort the list in ascending order:

```
> (sort '(1 3 6 5 7 9 2 4 8) <)
'(1 2 3 4 5 6 7 8 9)
```

Or, if you pass in >, it will sort the list in descending order:

```
> (sort '(1 3 6 5 7 9 2 4 8) >)
'(9 8 7 6 5 4 3 2 1)
```

append

To merge two lists together, you can use the append function:

```
> (append '(1 2 3) '(4 5 6))
'(1 2 3 4 5 6)
```

The append function can take more than two lists:

```
> (append '(1 2) '(3 4) '(5 6))
'(1 2 3 4 5 6)
```

range

The range function will create a list of numbers given some specifications. You can pass a start value and an end value, as well as a step to increment:

```
> (range 0 10 2)
'(0 2 4 6 8)
```

Or, if you just pass an end value, it will start at 0 with a step of 1:

```
> (range 10)
'(0 1 2 3 4 5 6 7 8 9)
```

make-list

Another way to make lists is using the make-list function:

```
> (make-list 10 'me)
'(me me me me me me me me me me)
```

As you can see, make-list takes a number and a value, and makes a list that contains that value repeated that number of times.

null?

To test whether a list is empty or not, you can use the null? function:

```
> (null? '()) ; test for empty list
#t
```


Copyrighted material

```
> (null? '(1 2 3))
#f
```

index-of

If you need to search a list for a value, you can use `index-of`. It'll return the index of the value if it appears:

```
> (index-of '(8 7 1 9 5 2) 9)
3
```

It'll return `#f` if it doesn't:

```
> (index-of '(8 7 1 9 5 2) 10)
#f
```

member

Another way to search lists is to use `member`, which tests whether a list contains an instance of a particular element. It returns the symbol `#f` if it does not, and returns the tail of the list starting with the first instance of the matching element if it does.

```
> (member 7 '(9 3 5 (6 2) 5 1 4))
#f

> (member 5 '(9 3 5 (6 2) 5 1 4))
'(5 (6 2) 5 1 4)

> (member 6 '(9 3 5 (6 2) 5 1 4))
#f
```

Notice that in the last instance, even though 6 is a member of a sublist of the searched list, the `member` function still returns false. However, the following does work.

```
> (member '(6 2) '(9 3 5 (6 2) 5 1 4))
'((6 2) 5 1 4)
```

You'll see later that in functional programming, you often need to determine whether an item is contained in a list. The `member` function not only finds the item (if it exists) but returns the actual value so that it can be used in further computations.

We'll have much more to say about lists in the remainder of this text.

Defines, Assigns, and Variables

Thus far, we've seen a few examples of a *function*, something that takes one or more input values and provides an output value (some form of data). The first element in a function-call expression is an identifier (the function

Copyrighted material

name). The remaining elements in a function form are the arguments to the function. These elements are each evaluated and then fed to the function, which performs some operation on its arguments and returns a value.

More specifically, a *form* or *expression* may define a function, execute a function call, or simply return a structure (normally a list), and may or may not evaluate all its arguments. Notice that `quote` is a different type of form (distinct from a function form, which evaluates its arguments) since it *does not* first evaluate its arguments. In the next section you'll meet `define`, which is yet another type of form since it does not evaluate its first argument, but it does evaluate its second argument. We will meet many other types of forms as we progress through the text.

A *variable* is a placeholder for a value. In Racket, variables are specified by *identifiers* (specific sequences of characters) associated with one thing only. (We'll have more to say about what constitutes a valid identifier shortly.) To define a variable, you use the `define` form. For example:

```
> (define a 123)
> a
123
```

Here `define` is said to *bind* the value `123` to the identifier `a`. Virtually anything can be bound to a variable. Here we'll bind a list to the identifier `b`.

```
> (define b '(1 2 3))
> b
'(1 2 3)
```

It's possible to bind several variables in parallel:

```
> (define-values (x y z) (values 1 2 3))

> x
1

> y
2

> z
3
```

Racket makes a distinction between *defining* a variable and *assigning* a value to a variable. Assignments are made with a `set!` expression. Typically any form which changes, or *mutates*, a value will end with an exclamation point. Attempting to assign to an identifier that hasn't been previously defined will result in an ugly error message:

```
> (set! ice 9)
. . set!: assignment disallowed;
  cannot set variable before its definition
  variable: ice
```


Copyrighted material

But this is okay:

```
> (define ice 9)
> ice
9
> (set! ice 32)
32
```

One way to think of this is that `define` sets up a location to store a value, and `set!` simply places a new value in a previously defined location.

When we speak of a variable *x* that is defined in Racket code, it will be typeset as `x`. If we're simply speaking of the variable in the mathematical sense, it will be typeset in italics as *x*.

Symbols, Identifiers, and Keywords

Unlike most languages, Racket allows just about any string of characters to be used as an identifier. For example we can use `2x3` as an identifier:

```
> (define 2x3 7)
> 2x3
7
```

You could conceivably define a function literally called `rags->riches` that would convert rags to riches (let me know when you get that working). All this seems quite bizarre, but it lends Racket an expressive power not found in many other computer languages. There are of course some restrictions to this, but aside from a few special characters such as parentheses, brackets, and arithmetic operators (even these are usually okay if they aren't the first character), just about anything goes. In fact it's common to see identifiers containing dashes, as in `solve-for-x`.

A *symbol* is essentially just a quoted identifier:

```
> 'this-is-a-symbol
'this-is-a-symbol
```

They are sort of a second-rate string (more on strings below). They are typically used much like an `enum` in other programming languages where they're used to stand for a specific value.

A *keyword* is an identifier prefixed with `#:`. Keywords are mainly used to identify optional arguments in function calls. Here's an example of a function (`~r`) that uses a keyword to output π as a string with two decimal places of accuracy.

```
> (~r pi #:precision 2)
"3.14"
```

Here we define the optional `precision` argument to specify that the value of `pi` should be rounded to two decimal places.

Copyrighted material

Equality

Racket defines two different kinds of equality: things that look exactly alike and things that are the same thing. Here's the difference. Suppose we make the following two definitions.

```
> (define a '(1 2 3))
> (define b '(1 2 3))
```

Identifiers a and b look exactly alike, and if we ask Racket if they are the same with the equal? predicate, it will respond that they are the same. Note a *predicate* is a function that returns a Boolean value of true or false.

```
> (equal? a b)
#t
```

But if we ask whether they are the same thing by using the eq? predicate, we get a different answer.

```
> (eq? a b)
#f
```

So when does eq? return true? Here's an example.

```
> (define x '(1 2 3))
> (define y x)
> (eq? x y)
#t
```

In this case we have bound x to the list '(1 2 3). We then bind y to the same value *location* that x is bound to, effectively making x and y be bound to the same thing. The difference is subtle, but important. In most cases equal? is what you need, but there are scenarios where eq? is used to ensure that variables are bound to the same object and not just to things that *look* the same.

One other nuance of equality that must be discussed is numeric equality. In the discussion above, we were focused on structural equality. Numbers are a different animal. We'll have much more to say about numbers in the next chapter, but we need to clarify a few things about numbers that relate to equality. Examine the following sequence:

```
> (define a  123)
> (define b  123)
> (eq? a b)
#t
```

Above we bound a and b to identical lists '(1 2 3), and in that case eq? returned false. In this case we bound a and b to the identical number 123, and eq? returned true. Numbers (technically *fixnums*, that is, small integers that fit into a fixed amount of storage—typically 32 or 64 bits, depending on your computing platform) are unique in this sense. There is only one instance of every number, no matter how many different identifiers it is bound to. In

Copyrighted material

other words, each number is stored in one and only one location. Furthermore, there's a special predicate (=) that can only be used with numbers:

```
> (= 123 123)
#t

> (= 123 456)
#f

(= '(1 2 3) '(1 2 3))
. . =: contract violation
  expected: number?
  given: '(1 2 3)
  argument position: 1st
  other arguments...:
```

In this section we only cover equality in general. We'll look at more specifics on numerical comparisons in the next chapter.

Strings and Things

In this section, we'll look at different ways of handling text values in Racket. We'll begin with the simplest kind of text value.

Characters

Individual text values, like single letters, are represented using a *character*, a special entity that corresponds to a *Unicode* value. For example, the letter A corresponds to the Unicode value 65. Unicode values are usually specified in hexadecimal, so the Unicode value for A is $65_{10} = 0041_{16}$. Character values either start with `#\` followed by a literal keyboard character or `#\u` followed by a Unicode value.

Here's a sampling of the multiple ways to write a character using character functions. Notice the use of the comment character (;), which allows comments (non-compiled text) to be added to Racket code.

```
> #\A
#\A

> #\u0041
#\A

> #\   ; this is a space character
#\space

> #\u0020  ; so is this
#\space
```


Copyrighted material

```
> (char->integer #\u0041)
65

> (integer->char 65)
#\A

> (char-alphabetic? #\a)
#t

> (char-alphabetic? #\1)
#f

> (char-numeric? #\1)
#t

> (char-numeric? #\a)
#f
```

Unicode supports a wide range of characters. Here are some examples:

```
> '(#\u2660 #\u2663 #\u2665 #\u2666)
'(#♠ #♣ #♡ #◇)

> '(#\u263A #\u2639 #\u263B)
'(\#☺ \#☹ \#☻)

> '(#\u25A1 #\u25CB #\u25C7)
'(\#□ \#○ \#◇)
```

Most Unicode characters should print fine, but this depends to some extent on the fonts available on your computer.

Strings

A *string* typically consists of a sequence of keyboard characters surrounded by double-quote characters.

```
> "This is a string."
"This is a string."
```

Unicode characters can be embedded in a string, but in this case, the leading # is left off.

```
> "Happy: \u263A."
"Happy: ☺."
```

You can also use `string-append` on two strings to create a new string.

```
> (string-append "Luke, " "I am " "your father!")
"Luke, I am your father!"
```


Copyrighted material

To access a character within a string, use string-ref:

```
> (string-ref "abcdef" 2)
#\c
```

The position of each character in a string is numbered starting from 0, so in this example using an index of 2 actually returns the third character.

The strings we have seen so far are immutable. To create a mutable string, use the string function. This allows changing characters in the string.

```
> (define wishy-washy (string #\I #\  #\a #\m #\  #\m #\u #\t #\a #\b #\l #\e)
   )
> wishy-washy
"I am mutable"

> (string-set! wishy-washy 5 #\a)
> (string-set! wishy-washy 6 #\ )

> wishy-washy
"I am a table"
```

Note that for mutable strings we have to define the string using individual characters.

Another way to create a mutable string is with string-copy:

```
> (define mstr (string-copy "I am also mutable"))
> (string-set! mstr 5 #\space)
> (string-set! mstr 6 #\space)
> mstr
"I am   so mutable"
```

You can also use make-string to do the same thing:

```
> (define exes (make-string 10 #\X))
> (string-set! exes 5 #\O)
> exes
"XXXXXOXXXX"
```

Depending on what's needed, any one of the above may be preferred. If you need to make an existing string mutable, string-copy is the obvious choice. If you only want a string of spaces, make-string is the clear winner.

Useful String Functions

There are of course a number of other useful string functions, a few of which we illustrate next.

Copyrighted material

string-length

The `string-length` function outputs the number of characters in a string (see `wishy-washy` earlier in "Strings" on page 14.)

```
> (string-length wishy-washy)
12
```

substring

The `substring` function extracts a substring from a given string.

```
> (substring wishy-washy 7 12) ; characters 7-11
"table"
```

string-titlecase

Use `string-titlecase` to capitalize the first character of each word in a string.

```
> (string-titlecase wishy-washy)
"I Am A Table"
```

string-upcase

To output a string in all caps, use `string-upcase`:

```
> (string-upcase "big")
"BIG"
```

string-downcase

Conversely, for a lowercase string, use `string-downcase`:

```
> (string-downcase "SMALL")
"small"
```

string<=?

To perform an alphabetical comparison, use the `string<=?` function:

```
> (string<=? "big" "small")  ; alphabetical comparison
#t
```

string=?

The `string=?` function tests whether two strings are equal:

```
> (string=? "big" "small")
#f
```


Copyrighted material

string-replace

The `string-replace` function replaces part of a string with another string:

```
> (define darth-quote "Luke, I am your father!")
> (string-replace darth-quote "am" "am not")
"Luke, I am not your father!"
```

string-contains?

To test whether one string is contained within another, use `string-contains?`:

```
> (string-contains? darth-quote "Luke")
#t

> (string-contains? darth-quote "Darth")
#f
```

string-split

The `string-split` function can be used to split a string into tokens:

```
> (string-split darth-quote)
'("Luke," "I" "am" "your" "father!")

> (string-split darth-quote ",")
'("Luke" " I am your father!")
```

Notice that the first example above uses the default version that splits on spaces whereas the second version explictly uses a comma (,).

string-trim

The `string-trim` function gets rid of any leading and/or trailing spaces:

```
> (string-trim "  hello   ")
"hello"

> (string-trim "  hello   " #:right? #f)
"hello   "

> (string-trim "  hello   " #:left? #f)
"  hello"
```

Notice in the last two versions, `#:left?` or `#:right?` is used to suppress trimming the corresponding side. The final `#f` argument (the default) is used to specify that only one match is removed from each side; otherwise all initial or trailing matches are trimmed.

For more advanced string functionality, see "Regular Expressions" on page 279.

Copyrighted material

String Conversion and Formatting Functions

There are a number of functions that convert values to and from strings. They all have intuitive names and are illustrated below.

```
> (symbol->string 'FBI)
"FBI"

> (string->symbol "FBI")
'FBI

> (list->string '(#\x #\y #\z))
"xyz"

> (string->list "xyz")
'(#\x #\y #\z)

> (string->keyword "string->keyword")
'#:string->keyword

> (keyword->string '#:keyword)
"keyword"
```

For a complete list of these, go to *https://docs.racket-lang.org/reference/strings.html*.

A handy function to embed other values within a string is `format`.

```
> (format "let ~a = ~a" "x" 2)
"let x = 2"
```

Within the format statement, ~a acts as a placeholder. There should be one placeholder for each additional argument. Note that the number 2 is automatically converted to a string before it's embedded in the output string.

If you want to simply convert a number to a string, use the `number->string` function:

```
> (number->string pi)
"3.141592653589793"
```

Conversely:

```
> (string->number "3.141592653589793")
3.141592653589793
```

Trying to get Racket to translate the value of words into numbers, however, will not work:

```
> (string->number "five")
#f
```


Copyrighted material

For more control we can use the ~r function, defined in the *racket/format* library, which has many options that can be used to convert a number to a string and control the precision and other output characteristics of the number. For example, to show π to four decimal places, we would use this:

```
> (~r pi #:precision 4)
"3.1416"
```

To show this right-justified, in a field 20 characters wide, and left padded with periods, we execute the following:

```
> (~r pi #:min-width 20 #:precision 4 #:pad-string ".")
"..............3.1416"
```

Additional info on ~r is available in Appendix A, which talks about number bases. There are a number of other useful tilde-prefixed string conversion functions available, such as ~a, ~v, and ~s. We won't go into detail here, but you can consult the Racket Documentation for details: *https://docs.racket-lang.org/reference/strings.html*.

Vectors

Vectors bear a superficial resemblance to lists, but they are quite different. In contrast to the internal tree structure of lists, *vectors* are a sequential array of cells (much like arrays in imperative languages) that directly contain values, as illustrated in Figure 1-3.

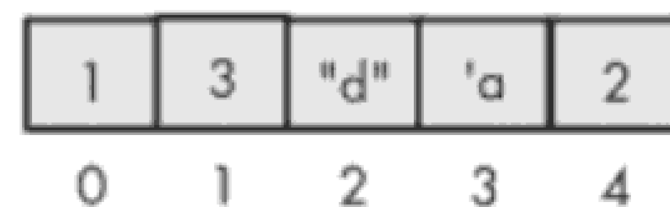


Figure 1-3: Vector structure

Vectors can be entered using the vector function.

```
> (vector 1 3 "d" 'a 2)
'#(1 3 "d" a 2)
```

Alternatively, vectors can be entered using # as follows (note that an unquoted # implies a quote):

```
> #(1 3 "d" a 2)
'#(1 3 "d" a 2)
```

It's important to note that these methods are *not* equivalent. Here's one reason why:

```
> (vector 1 2 pi)
'#(1 2 3.141592653589793)

> #(1 2 pi)
'#(1 2 pi)
```


Copyrighted material

In the first example, just as for `list`, `vector` first evaluates its arguments before forming the vector. In the last example, like `quote`, `#` does not evaluate its arguments. More importantly, `#` is an alias for `vector-immutable`, which leads to our next topic.

Accessing Vector Elements

The function `vector-ref` is an indexing operator that returns an element of a vector. This function takes a vector as its first argument and an index as its second:

```
> (define v (vector 'alpha 'beta 'gamma))
> (vector-ref v 1)
'beta

> (vector-ref v 0)
'alpha
```

To assign a value to a vector cell, `vector-set!` is used. The `vector-set!` expression takes three arguments: a vector, an index, and a value to be assigned to the indexed position in the vector.

```
> (vector-set! v 2 'foo)
> v
'#(alpha beta foo)
```

Let's try this a bit differently:

```
> (define u #(alpha beta gamma))
> (vector-set! u 2 'foo)
. . vector-set!: contract violation
  expected: (and/c vector? (not/c immutable?))
  given: '#('alpha 'beta 'gamma)
  argument position: 1st
  other arguments...:
```

Remember that `#` is an alias for `vector-immutable`. What this means is that vectors created with `#` (or `vector-immutable`) are (drum roll . . .) *immutable*: they cannot be changed or assigned new values. On the other hand, vectors created with `vector` are *mutable*, meaning that their cells can be modified.

One advantage of vectors over lists is that elements of vectors can be accessed much faster than elements of lists. This is because to access the 100th element of a list, each cell of the list must be accessed sequentially to get to the 100th element. Conversely, with vectors, the 100th element can be accessed directly, without working through earlier cells. On the other hand, lists are quite flexible and can easily be extended as well as being used to represent other data structures like trees. They are the bread and butter of Racket (and all Lisp-based languages), so much of the functionality of the language depends on the list structure. Predictably, functions are provided to easily convert from one to the other.

Copyrighted material

Useful Vector Functions

vector-length

The `vector-length` function returns the number of elements in a vector:

```
> (vector-length #(one ringy dingy))
3
```

vector-sort

The `vector-sort` function sorts the elements of a vector:

```
> (vector-sort #(9 1 3 8 2 5 4 0 7 6 ) <)
'#(0 1 2 3 4 5 6 7 8 9)

> (vector-sort #(9 1 3 8 2 5 4 0 7 6 ) >)
'#(9 8 7 6 5 4 3 2 1 0)
```

To whet your appetite for what's to come later, `vector-sort` is a typical example of functional programming. The last argument actually evaluates a function that determines the direction of the sort.

vector->list

The `vector->list` function takes a vector and returns a list:

```
>  (vector->list #(one little piggy))
'(one little piggy)
```

list->vector

Conversely `list->vector` takes a list and returns a vector:

```
> (list->vector '(two little piggies))
'#(two little piggies)
```

make-vector

To create a mutable vector, use the `make-vector` form:

```
> (make-vector 10 'piggies) ; create a mutable vector
'#(piggies piggies piggies piggies piggies piggies piggies piggies piggies
    piggies)
```

vector-append

To concatenate two vectors together, use `vector-append`:

```
> (vector-append #(ten little) #(soldier boys))
'#(ten little soldier boys)
```


Copyrighted material

vector-member

The vector-member function returns the index to where an item is located in a vector:

```
> (vector-member 'waldo (vector 'where 'is 'waldo '?) )
2
```

There are of course many other useful vector functions, and we will explore some of them in the chapters to come.

Using structs

To introduce the next Racket feature, let's build an example program. Instead of keeping your checkbook transactions in a paper bankbook, you could create an electronic version using Racket. Typically such transactions have the following components:

- Transaction date
- Payee
- Check number
- Amount

One way to keep track of these disparate pieces of information is in a Racket structure called a struct. A Racket struct is conceptually similar to a struct in languages such as C or C++. It's a composite data structure that has a set of predefined fields. Before you can use a struct, you have to tell Racket what it looks like. For our bank transaction example, such a definition might look like this:

```
> (struct transaction (date payee check-number amount))
```

Each of the components of a structure (date, payee, etc.) is called a *field*. Once we've defined our transaction struct, we can create one like this:

```
> (define trans (transaction 20170907 "John Doe" 1012 100.10))
```

Racket automatically creates an *accessor method* for each of the fields in the structure. An accessor method returns the value of the field. They always begin with the name of the structure (in this case transaction), a hyphen, and then the name of the field.

```
> (transaction-date trans)
20170907

> (transaction-payee trans)
"John Doe"

> (transaction-check-number trans)
1012
```


Copyrighted material

```
> (transaction-amount trans)
100.1
```

Suppose, however, that you made a mistake and determined that the check to John Doe should have been for $100.12 instead of $100.10 and try to correct it via `set-transaction-amount!`. Note the exclamation point: this is a signal that `set-transaction-amount!` is a *mutator*, that is, a method that modifies a field value). These mutators are generated when the struct is defined and typically start with set and end with !.

```
> (set-transaction-amount! trans 100.12)
. . set-transaction-amount!: undefined;
  cannot reference an identifier before its definition
```

Oops . . . Fields in a structure are immutable by default and hence do not export *mutators*. The way around this is to include the `#:mutable` keyword in the structure definition for any field that may need to be modified.

```
> (struct transaction
    (date payee check-number [amount #:mutable]))
> (define trans (transaction 20170907 "John Doe" 1012 100.10))
> (set-transaction-amount! trans 100.12)
> (transaction-amount trans)
100.12
{
```

If all the fields should be mutable, adding the `#:mutable` keyword after the field list will do the trick.

```
> (struct transaction
    (date payee check-number amount) #:mutable)
> (define trans (transaction 20170907 "John Doe" 1012 100.10))
> (set-transaction-check-number! trans 1013)
> (transaction-check-number trans)
1013
```

While the accessor methods are sufficient for getting the value of a single field, they are a bit cumbersome for seeing all the values at once. Just entering the structure name does not yield much information.

```
> trans
#<transaction>
```

To make your structure more transparent, include the `#:transparent` option in the struct definition.

```
> (struct transaction
    (date payee check-number amount) #:mutable #:transparent)
> (define trans (transaction 20170907 "John Doe" 1012 100.10))
> trans
(transaction 20170907 "John Doe" 1012 100.1)
```


Copyrighted material

There are additional useful options that can be applied when defining structures, but one that is of particular interest is `#:guard`. `#:guard` provides a mechanism to validate the fields when a structure is constructed. For instance, to ensure that negative check numbers are not used, we could do the following.

```
> (struct transaction
    (date payee check-number amount)
    #:mutable #:transparent
    #:guard (λ (date payee num amt name)
        (unless (> num 0)
        (error "Not a valid check number"))
        (values date payee num amt)))

> (transaction 20170907 "John Doe" -1012 100.10)
Not a valid check number

> (transaction 20170907 "John Doe" 1012 100.10)
(transaction 20170907 "John Doe" 1012 100.1)
```

Don't panic. We haven't covered that funny-looking symbol (λ, or *lambda*) yet, but you should be able to get the gist of what's going on. The `#:guard` expression is a function that takes one parameter for each field and one additional parameter that contains the structure name. In this case we're only testing whether the check number is greater than zero. The `#:guard` expression must return the same number of values as the number of fields in the `struct`.

In the previous example we simply returned the same values that were entered, but suppose we had a variable that contained the last check number called `last-check`. In this case, we could enter a 0 for the check number and use the `#:guard` expression to plug in the next available number as shown here.

```
> (define last-check 1000)

> (struct transaction
    (date payee check-number amount)
    #:mutable #:transparent
    #:guard (λ (date payee num amt name)
              (cond
                [(< num 0)
                   (error "Not a valid check number")]
                [(= num 0)
                   (let ([next-num (add1 last-check)])
                     (set! last-check next-num)
                     (values date payee next-num amt))]
                [else
                   (set! last-check num)
                   (values date payee num amt)])))
```


Copyrighted material

```
> (transaction 20170907 "John Doe" 0 100.10)
(transaction 20170907 "John Doe" 1001 100.1)

> (transaction 20170907 "Jane Smith" 1013 65.25)
(transaction 20170907 "Jane Smith" 1013 65.25)

> (transaction 20170907 "Acme Hardware" 0 39.99)
(transaction 20170907 "Acme Hardware" 1014 39.99)
```

As you can see, non-zero check numbers are stored as the last check number, but if a zero is entered for the check number, the `struct` value gets generated with the next available number, which becomes the current value for `last-check`. The `cond` statement will be explained in more detail a bit later in the book, but its use here should be fairly clear: it's a way to check multiple cases.

Controlling Output

In the interactions pane, DrRacket immediately displays the output resulting from evaluating any expression. It's often desirable to have some control over how the output is presented. This is especially important when the output is being generated by some function or method. Racket provides a number of mechanisms for generating formatted output. The main forms are `write`, `print`, and `display`. Each of these works in a slightly different way. The best way to illustrate this is with examples.

write

The `write` expression outputs in such a way that the output value forms a valid value that can be used in the input:

```
> (write "show me the money")
"show me the money"

> (write '(show me the money))
(show me the money)

> (write #\A)
#\A

> (write 1.23)
1.23

> (write 1/2)
1/2

> (write #(a b c))
#(a b c)
```


Copyrighted material

display

The display expression is similar to write, but strings and character data types are written as raw strings and characters without any adornments such as quotation or tick marks:

```
> (display "show me the money")
show me the money

> (display '(show me the money))
(show me the money)

> (display #\A)
A

> (display 1.23)
1.23

> (display 1/2)
1/2

> (display #(a b c))
#(a b c)
```

print

The print expression is also similar to write, but adds a bit more formatting to the output. The intent of print is to show an expression that would evaluate to the same value as the printed one:

```
> (print "show me the money")
"show me the money"

> (print '(show me the money))
'(show me the money)

> (print #\A)
#\A

> (print 1.23)
1.23

> (print 1/2)
1
-
2

> (print #(a b c))
'#(a b c)
```


Copyrighted material

Notice how the rational value 1/2 is printed (more on rationals in the next chapter).

Each of these comes in a form that ends with ln. The only difference is that the ones that end with ln automatically print a new line at the end. Here are a couple of examples to highlight the difference.

```
> (print "show me ") (print "the money")
"show me ""the money"

> (display "show me ") (display "the money")
show me the money

> (println "show me ") (println "the money")
"show me "
"the money"

> (displayln "show me ") (displayln "the money")
show me
the money
```

One very useful form is printf. The printf expression works much like the format function: it takes a format string as its first argument and any number of other values as its other argument. The format string uses ~a as a placeholder. There must be one placeholder for each of the arguments after the format string. The format string is printed exactly as entered, with the exception that for each placeholder the corresponding argument is substituted. Here's printf in action.

```
> (printf "~a + ~a = ~a" 1 2 (+ 1 2))
1 + 2 = 3

> (printf "~a, can you hear ~a?" "Watson" "me")
Watson, can you hear me?

> (printf "~a, can you hear ~a?" "Jeeves" "the bell")
Jeeves, can you hear the bell?
```

There are additional format specifiers (see the Racket Documentation for details), but we'll mostly be using print since it gives a better visual indication of the data type of the value being output.

Summary

In this chapter, we laid the groundwork for what's to come. Most of the core data types have been introduced along with what are hopefully some helpful examples. By now you should be comfortable with basic Racket syntax and have a pretty good understanding of the structure of lists and how to manipulate them. The next chapter will take a detailed look at the various numeric data types provided by Racket.

Copyrighted material

2

ARITHMETIC AND OTHER NUMERICAL PARAPHERNALIA

In this chapter, we'll take a look at the rich set of numerical data types that Racket provides. We'll discover the expected integer and floating-point values, but we'll also learn that Racket supports rational (or fractional) values along with complex numbers (don't worry if you don't know what complex numbers are; they are not heavily used in this text, but we take a brief look for those that may be interested).

Booleans

Booleans are true and false values, and while they aren't strictly numbers, they behave a bit like numbers in that they can be combined by various operators to produce other Boolean values. The discipline governing these operations is known as *Boolean algebra*. In Racket, Booleans are represented by the values `#t` and `#f`, true and false respectively. It's also possible to use `#true` (or `true`) and `#false` (or `false`) as aliases for `#t` and `#f` respectively.

Copyrighted material

Before we introduce specific Boolean operators, one important observation about Racket Boolean operators in general is that they typically treat any value that's not literally #f as true. You'll see some examples of this behavior below.

The first operator we'll look at is not, which simply converts #t to #f and vice versa.

```
> (not #t)
#f

> (not #f)
#t

> (not 5)
#f
```

Notice that 5 was converted to #f, meaning that it was originally treated as #t.

The next Boolean operator we'll look at is and, which returns true if all its arguments are true. Let's look at some examples:

```
> (and #t #t)
#t

> (and #t #f)
#f

> (and 'apples #t)
#t

> (and (equal? 5 5) #f)
#f

> (and (equal? 5 5) #t)
#t

> (and (equal? 5 5) #t 23)
23
```

You may be a bit puzzled by the last example (and rightfully so). Remember that Racket considers all non-false values as true, so 23 is in fact a valid return value. More important though is how and evaluates its arguments. What happens in reality is that and sequentially evaluates its arguments until it hits a #f value. If no #f value is encountered, it returns the value of its last argument, 23 in the example above. While this behavior seems a bit odd, it is consistent with how the or operator works, where, as we'll see shortly, it can be quite useful in certain circumstances.

Copyrighted material

The last Boolean operator we'll look at is the `or` operator, which will return true if any of its arguments are true and `#f` otherwise. Here are some examples:

```
> (or #f #f)
#f

> (or #f #t)
#t

> (or #f 45 (= 1 3))
45
```

Much like `and`, `or` sequentially evaluates its arguments. But in `or`'s case, the first *true* value is returned. In the example above, 45 is treated as true, so that's the value returned. This behavior can be quite useful when one wants the first value that's not `#f`.

Other less frequently used Boolean operators are `nand`, `nor`, and `xor`. Consult the Racket Documentation for details on these operators.

The Numerical Tower

In mathematics there's a hierarchy of number types. *Integers* are a subset of rational (or fractional) numbers. *Rational numbers* are a subset of real numbers (or floating-point values as they are approximated by computers). And *real numbers* are a subset of complex numbers. This hierarchy is known as the *numerical tower* in Racket.

Integers

In mathematics the set of integers is represented by the symbol $\mathbb{Z}$. Racket integers consist of a sequence of digits from 0 to 9, optionally preceded by a plus or minus sign. Integers in Racket are said to be *exact*. What this means is that applying arithmetical operations to exact numbers will always produce an exact numerical result (in this case a number that's still an integer). In many computer languages, once an operation produces a number of a certain size, the result will either be incorrect or it will be converted to an approximate value represented by a floating- point number. With Racket, numbers can get bigger and bigger until your computer literally runs out of memory and explodes. Here are some examples.

```
> (+ 1 1)
2

> (define int 1234567890987654321)
> (* int int int int)
2323057235416375647706123102514602108949250692331618011140356079618623681

> (- int)
```


Copyrighted material

```
-1234567890987654321

> (- 5 -7)
12

> (/ 4 8)
1/2

> (/ 5)
1/5
```

Note that in the last examples, division doesn't result in a floating-point number but rather returns an *exact* value: a rational number (discussed in the next section).

It's possible to enter integers in number bases other than 10. Racket understands *binary numbers* (integers prefixed by `#b`), *octal* numbers (integers prefixed by `#o`), and *hexadecimal* numbers (integers prefixed by `#x`):

```
> #b1011
11

> #b-10101
-21

> #o666
438

> #xadded
712173
```

Non-decimal bases have somewhat specialized use cases, but one example is that HTML web pages typically express color values as hexadecimal numbers. Also, binary numbers are how computers store all values internally, so they can be useful for individuals studying basic computer science. Octal and hexadecimal values have a further advantage: binary numbers can easily be converted to octal since three binary digits equates to a single octal value and four binary digits equates to a single hexadecimal digit.

Rationals

Next up on the mathematical food chain are the rational numbers (or fractions), expressed by the mathematical symbol $\mathbb{Q}$. Fractions in Racket consist of two positive integer values separated by a forward slash (no spaces allowed), optionally preceded by a plus or minus sign. Rational numbers are also an exact numeric type, and all operations permitted for integers are also valid for rational numbers.

Copyrighted material

```
> -2/4
-1/2

> 4/6
2/3

> (+ 1/2 4/8)
1

> (- 1/2 2/4 4/8 8/16)
-1

> (* 1/2 2/3)
1/3

> (/ 2 2/3)
3
```

The numerator and denominator of a rational number can be obtained with the `numerator` and `denominator` functions.

```
> (numerator 2/3)
2

> (denominator 2/3)
3
```

Reals

A *real* number is a mathematical concept (specified by the symbol $\mathbb{R}$) that, in reality, does not exist in the world of computers. Real numbers such as π have an infinite decimal expansion that can only be approximated in a computer. Thus, we reach our first class of *inexact* numbers: floating-point numbers. Floating-point numbers in Racket are entered in the same way as they are in most programming languages and calculators. Here are some (unfortunately boring) examples:

```
> -3.14159
-3.14159

> 3.14e159
3.14e+159

> pi
3.141592653589793

> 2.718281828459045
```


Copyrighted material

```
2.718281828459045

> -20e-2
-0.2
```

It's important to keep in mind that there are some subtle distinctions in the mathematical concept of certain number types and what they mean in a computing environment. For example a number entered as 1/10 is, as mentioned above, treated as an exact rational number since it can be represented as such in a computer (internally it's stored as two binary integer values), but the value 0.1 is treated as an inexact floating-point value, an approximation of the real number value, since it cannot be represented internally a single binary value (at least not without using an infinite number of binary digits).

Complex Numbers

When we use the term *complex* number it does not mean we are speaking of a *complicated* number, but rather a special type of number. If you're not already familiar with this concept, there's no harm in moving on to the next section, since complex numbers aren't used in the remainder of the book (although I would encourage you to read up on this fascinating subject). This section is included as a reference for the brave souls who may make use of this information in their own projects.

Complex numbers are entered almost exactly as they appear in any mathematical text, but there are some points to note. First, if the real component is omitted, the imaginary part must be preceded by a plus or minus sign. Second, there can be no spaces in the string used to define the number. And third, complex numbers must end in `i`. Examples:

```
> +1i ; our friend, the imaginary number
0+1i

> 1i ; this will give an error
. . 1i: undefined;
  cannot reference an identifier before its definition

> +i ; it is even possible to leave off the 1
0+1i

> -1-234i
-1-234i

> -1.23+4.56i
-1.23+4.56i

> 1e10-2e10i
10000000000.0-20000000000.0i
```


Copyrighted material

Note that complex numbers can be exact or inexact. We can test exactness using the exact? operator:

```
> (exact? 1/2+8/3i)
#t

> (exact? 0.5+8/3i)
#f
```

To get at the components of a complex number, use real-part and imag-part:

```
> (real-part 1+2i)
1

> (imag-part 1+2i)
2
```

This concludes our look at the numerical tower and basic arithmetical operations on the various number types. In the next few sections we'll look at comparison operators, what happens when different number types are added together (for example adding an integer to a floating-point number), and some useful mathematical functions.

Numeric Comparison

Racket supports the usual complement of numeric comparison operators. We can test if numbers are equal:

```
> (= 1 1.0)
#t

> (= 1 2)
#f

> (= 0.5 1/2)
#t
```

and compare their sizes:

```
> (< 1 2)
#t

> (<= 1 2)
#t

> (>= 2 1.9)
#t
```


Copyrighted material

You can also use these operators on multiple arguments, and Racket will ensure that the elements pair-wise satisfy the comparison operator. In the example below, this means that 1 < 2, 2 < 3, and 3 < 4.

```
> (< 1 2 3 4)
#t

> (< 1 2 4 3)
#f
```

But there's no *not equals* operator, so to test if two numbers are not equal to each other, you would have to do something like the following:

```
> (not (= 1 2))
#t
```

Combining Data Types

As you saw above, you can compare numbers of different types. But notice that we only performed arithmetic on exact numbers with exact numbers and vice versa. Here we'll discuss the implications of mixing exact and inexact numbers. Mixing exact and inexact numbers won't result in mass chaos (think *Ghostbusters* stream-crossing), but there are some fine points you should be aware of.

First and foremost, when it comes to arithmetic operators (addition, subtraction, and so on), the rules are fairly simple:

Mixing exact with exact will give an exact result.

Mixing inexact with inexact will give an inexact result.

Mixing exact with inexact (or vice versa) will give an inexact result.

No surprises here, but there are some nuanced exceptions to these rules, such as multiplying anything by zero gives exactly zero.

Trigonometric functions will generally always return an inexact result (but again, there are some reasonable exceptions; for example `exp 0` gives an exact 1). You'll see some of these functions later in the chapter. The square function, `sqr`, will return an exact result if given an exact number. Its square root counterpart, `sqrt`, will return an exact result if it's given an exact number *and* the result is an exact number; otherwise, it will return an inexact number:

```
> (sqrt 25)
5

> (sqrt 24)
4.898979485566356

> (sqr 1/4)
1/16
```


Copyrighted material

```
> (sqr 0.25)
0.0625

> (sqrt 1/4)
1/2

> (sqrt -1)
0+1i
```

There are a couple of functions available to test exactness. Earlier you saw the function `exact?`, which returns `#t` if its argument is an exact number; otherwise it returns `#f`. Its counterpart is `inexact?`. It's also possible to force an exact number to be inexact and vice versa using two built-in functions:

```
> (exact->inexact 1/3)
0.3333333333333333

> (inexact->exact pi)
3 39854788871587/281474976710656
>
```

There's a predicate to test for each of the numeric data types we have mentioned in this section, but they may not work exactly as you expect.

```
> (integer? 70)
#t

> (real? 70.0)
#t

> (complex? 70)
#t

> (integer? 70.0)
#t

> (integer? 1.5)
#f

> (rational? 1.5)
#t

> (rational? 1+5i)
#f

> (real? 2)
#t
```


Copyrighted material

```
> (complex? 1+2i)
#t
```

These predicates return a result that honors the mathematical meaning of the predicate. You may have expected `(complex? 70)` to return `#f`, but integers are complex numbers, just with a zero real component. Likewise, you may have expected `(integer? 70.0)` to return `#f` since it's a floating-point number, but since the fractional part is 0, the number (while also real) is in fact an integer (but not an exact number). The number 1.5 is equivalent to 3/2, so Racket considers this to be a rational number (but again, inexact). The number type predicates (`integer?`, `rational?`, `real?`, and `complex?`) are aligned with the mathematical hierarchy (or numerical tower) as mentioned at the beginning of the section.

Built-in Functions

Aside from the normal arithmetical operators illustrated above, Racket provides the usual complement of mathematical functions that are standard fare in any programming language. A generous litany of examples follows.

```
> (abs -5)
5

> (ceiling 1.5)
2.0

> (ceiling 3/2)
2

> (floor 1.5)
1.0

> (tan (/ pi 4))
0.9999999999999999

> (atan 1/2)
0.4636476090008061

> (cos (* 2 pi))
1.0

> (sqrt 81)
9

> (sqr 4)
16

> (log 100) ; natural logarithm
```


Copyrighted material

```
4.605170185988092

> (log 100 10) ; base 10 logarithm
2.0

> (exp 1) ; e^1
2.718281828459045

> (expt 2 8) ; 2^8
256
```

Note that when possible, a function that has an exact argument will return an exact result.

There are of course many other functions available. Consult the Racket Documentation for details.

Infix Notation

As we've seen, in Racket, mathematical operators are given before the operands: `(+ 1 2)`. Typical mathematical notation has the operator between the operands: $1 + 2$. This is called *infix notation*. Racket natively allows a form of infix notation by using a period operator. Here are some examples.

```
> (1 . >= . 2)
#f

> (1 . < . 2)
#t

> (1 . + . 2)
3

> (2 . / . 4)
1/2

> (2 . * . 3)
6
```

This can be useful when we want to make explicit the relationship between certain operators, but it's unwieldy for complex expressions.

For complex mathematical expressions, Racket provides the `infix` package. This package can be imported with the following code:

```
#lang at-exp racket
(require infix)
```

The `#lang` keyword allows us to define language extensions (in this case the `at-exp` allows us to use @-expressions, which we will see shortly). The `require infix` expression states that we want to use the *infix* library.

Copyrighted material

Unfortunately, the infix package is not installed by default and must be installed from the Racket package manager (the package manager can be accessed through the DrRacket File menu) or the raco command line tool (if the
executable for raco is not in your execution path, it can be launched directly from the Racket install folder). To install using raco, execute the following on the command line:

```
> raco pkg install infix
```

Also note that we're using the language extension at-exp, which, while not entirely necessary, provides a nicer syntax to enter infix expressions. For example without at-exp, to compute $1 + 2 * 3$, we would enter the following:

```
> ($ "1+2*3")
7
```

With the at-exp extension, we could enter this:

```
> @${1+2*3}
7
```

While this only saves a couple of keystrokes, it removes the annoying string delimiters and just looks a bit more natural.

Function calls are handled in a familiar way by using square brackets. For example

```
> @${1 + 2*sin[pi/2]}
3.0
```

There is even a special form for lists:

```
> @${{1, 2, 1+2}}
'(1 2 3)
```

And there's one for variable assignments (which use :=, equivalent to set!, so the variable must be bound first):

```
> (define a 5)
> @${a^2}
25
> @${a := 6}
> @${2*a + 7}
19
```

To further illustrate the capabilities of the infix package, below is a complete program containing a function called quad, which returns a list containing the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

Copyrighted material

As you'll recall from your algebra class (you *do* remember, don't you), these roots are given by

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
#lang at-exp racket
(require infix)

(define (quad a b c)
  (let ([d 0])
    @${d := sqrt[b^2 - 4 * a * c];
         {(-b + d)/(2*a), (-b - d)/(2*a)}}))
```

After compiling this, we can solve $2x^2 - 8x + 6 = 0$ for x, by entering

```
> @${quad[2, -8, 6]}
'(3 1)
```

or equivalently . . .

```
> (quad 2 -8 6)
'(3 1)
```

Summary

With these first two chapters under your belt, you should be thoroughly familiar with Racket's basic data types. You should also be comfortable performing mathematical operations in Racket's rich numerical environment. This should prepare you for the somewhat more interesting topics to follow where we will explore number theory, data analysis, logic programming, and more. But, next up is functional programming, where we get down to the nitty-gritty of actually creating programs.

Copyrighted material

Copyrighted material

3

FUNCTION FUNDAMENTALS

In the last chapter, we introduced you to Racket's basic numerical operations. In this chapter, we'll explore the core ideas that form the subject of functional programming.

What Is a Function?

A *function* can be thought of as a box with the following characteristics: if you push an object in one side, an object (possibly the same, or not) comes out the other side; and for any given input item, the same output item comes out. This last characteristic means that if you put a triangle in one side and a star comes out the other, the next time you put a triangle in, you will also get a star out (see Figure 3-1). Unfortunately, Racket doesn't have any built-in functions that take geometric shapes as input, so we'll need to settle for more-mundane objects like numbers or strings.

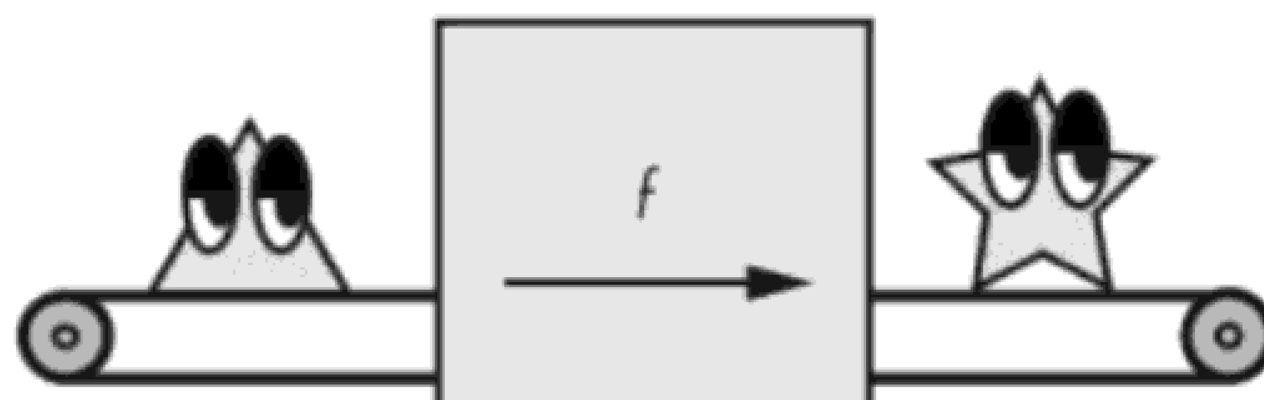


Figure 3-1: How a function works

Copyrighted material

Lambda Functions

In its most basic form, a function in Racket is something produced by a *lambda expression*, designated by the Greek letter λ. This comes from a mathematical discipline called lambda calculus, an arcane world we won't explore here. Instead, we'll focus on practical applications of lambda expressions. Lambda functions are intended for short simple functions that are immediately applied, and hence, don't need a name (they're anonymous). For example, Racket has a built-in function called `add1` that simply adds 1 to its argument. A Racket lambda expression that does the same thing looks like this:

```
(lambda (x) (+ 1 x))
```

Racket lets you abbreviate `lambda` with the Greek symbol λ, and we'll frequently designate it this way. You can enter λ in DrRacket by selecting it from the Insert menu or using the keyboard shortcut CTRL-\. We could rewrite the code above to look like this:

```
(λ (x) (+ 1 x))
```

To see a lambda expression in action, enter the following in the interactions pane:

```
> ((λ (x y) (+ (* 2 x) y)) 4 5)
13
```

Notice that instead of a function name as the first element of the list, we have the actual function. Here 4 and 5 get passed to the lambda function for evaluation.

An equivalent way of performing the above computation is with a `let` form.

```
> (let ([x 4]
        [y 5])
    (+ (* 2 x) y))
13
```

This form makes the assignment to variables `x` and `y` more obvious.

We can use lambda expressions in a more conventional way by assigning them to an identifier (a named function).

```
> (define foo (λ (x y) (+ (* 2 x) y)))
> (foo 4 5)
13
```

Racket also allows you to define functions using this shortcut:

```
> (define (foo x y) (+ (* 2 x) y))
> (foo 4 5)
13
```

These two forms of function definition are entirely equivalent.

Copyrighted material

Higher-Order Functions

Racket is a functional programming language. *Functional programming* is a programming paradigm that emphasizes a declarative style of programming without side effects. A *side effect* is something that changes the state of the programming environment, like assigning a value to a global variable.

Lambda values are especially powerful because they can be passed as values to other functions. Functions that take other functions as values (or return a function as a value) are known as *higher-order functions*. In this section, we'll explore some of the most commonly used higher-order functions.

The map Function

One of the most straightforward higher-order functions is the `map` function, which takes a function as its first argument and a list as its second argument, and then applies the function to each element of the list. Here's an example of the `map` function:

```
> (map (λ (x) (+ 1 x)) '(1 2 3))
'(2 3 4)
```

You can also pass a named function into `map`:

```
> (define my-add1 (λ (x) (+ 1 x)))
> (map my-add1 '(1 2 3)) ; this works too
'(2 3 4)
```

In the first example above, we take our increment function and pass it into `map` as a value. The `map` function then applies it to each element in the list `'(1 2 3)`.

It turns out that `map` is quite versatile. It can take as many lists as the function will accept as arguments. The effect is sort of like a zipper, where the list arguments are fed to the function in parallel, and the resulting values is a single list, formed by applying the function to the elements from each list. The example below shows `map` being used to add the corresponding elements of two equally sized lists together:

```
> (map + '(1 2 3) '(2 3 4))
'(3 5 7)
```

As you can see, the two lists were combined by adding the corresponding elements together.

The apply Function

The `map` function lets you apply a function to each item in a list individually. But sometimes, we want to apply all the elements of a list as arguments in a single function call. For example, Racket arithmetical operators can take multiple numeric arguments:

```
> (+ 1 2 3 4)
10
```


Copyrighted material

But if we try to pass in a list as an argument, we'll get an error:

```
> (+ '(1 2 3 4))
. . +: contract violation
  expected: number?
  given: '(1 2 3 4)
```

The + operator is only expecting numeric arguments. But not to worry. There's a simple solution: the `apply` function:

```
> (apply + '(1 2 3 4))
10
```

The `apply` function takes a function and a list as its arguments. It then *applies* the function to values in the list as if they were arguments to the function.

The foldr and foldl Functions

Yet another way to add the elements of a list together is with the `foldr` function. The `foldr` function takes a function, an initial argument, and a list:

```
> (foldr + 0 '(1 2 3 4))
10
```

Even though `foldr` produced the same result as `apply` here, behind the scenes it worked very differently. This is how `foldr` added the list together: $1 + (2 + (3 + (4 + 0)))$. The function "folds" the list together by performing its operation in a right-associative fashion (hence the `r` in `foldr`).

Closely associated with `foldr` is `foldl`. The action of `foldl` is slightly different from what you might expect. Observe the following:

```
> (foldl cons '() '(1 2 3 4))
'(4 3 2 1)

> (foldr cons '() '(1 2 3 4))
'(1 2 3 4)
```

One might have expected `foldl` to produce `'(1 2 3 4)`, but actually `foldl` performs the computation `(cons 4 (cons 3 (cons 2 (cons 1 '()))))`. The list arguments are processed from left to right, but the two arguments fed to `cons` are reversed—for example, we have `(cons 1 '())` and not `(cons '() 1)`.

The compose Function

Functions can be combined together, or *composed*, by passing the output of one function to the input of another. In math, if we have $f(x)$ and $g(x)$, they can be composed to make $h(x) = f(g(x))$ (in mathematics text this is sometimes designated with a special composition operator as $h(x) = (f \circ g)(x)$. We can do this in Racket using the `compose` function, which takes two or more functions and returns a new composed function. This new function works a bit like a pipeline. For example, if we want to increment a number by 1 and

Copyrighted material

square the result (that is, for any n compute $(n + 1)^2$), we could use following function:

```
(define (n+1_squared n) (sqr (add1 n)))
```

But compose allows this to be expressed a bit more succinctly:

```
> (define n+1_squared (compose sqr add1))
> (n+1_squared 4)
25
```

Even simpler . . .

```
> ((compose sqr add1) 4)
25
```

Please note that add1 is performed first and then sqr. Functions are composed from right to left—that is, the rightmost function is applied first.

The filter Function

Our final example is filter. This function takes a predicate (a function that returns a Boolean value) and a list. The returned value is a list such that only elements of the original list that satisfy the predicate are included. Here's how we'd use filter to return the even elements of a list:

```
> (filter even? '(1 2 3 4 5 6))
'(2 4 6)
```

The filter function allows you to filter out items in the original list that won't be needed.

As you've seen throughout this section, our description of a function as a box is apt since it is in reality a value that can be passed to other functions just like a number, a string, or a list.

Lexical Scoping

Racket is a lexically scoped language. The Racket Documentation provides the following definition for *lexical scoping*:

> Racket is a lexically scoped language, which means that whenever an identifier is used as an expression, something in the textual environment of the expression determines the identifier's binding.

What's important about this definition is the term *textual environment*. A textual environment is one of two things: the *global environment*, or forms where identifiers are bound. As we've already seen, identifiers are bound in the global environment (sometimes referred to as the top level) with define. For example

```
> (define ten 10)
> ten
10
```


Copyrighted material

The values of identifiers bound in the global environment are available everywhere. For this reason, they should be used sparingly. Global definitions should normally be reserved for function definitions and constant values. This, however, is not an edict, as there are other legitimate uses for global variables.

Identifiers bound within a form will *normally* not be defined outside of the form environment (but see "Time for Some Closure" on page 58 for an intriguing exception to this rule).

Let's look at a few examples.

Previously we explored the lambda expression `((λ (x y) (+ (* 2 x) y)) 4 5)`. Within this expression, the identifiers `x` and `y` are bound to 4 and 5. Once the lambda expression has returned a value, the identifiers are no longer defined.

Here again is the equivalent `let` expression.

```
(let ([x 4]
      [y 5])
  (+ (* 2 x) y))
```

You might imagine that the following would work as well:

```
(let ([x 4]
      [y 5]
      [z (* 2 x)])
  (+ z y))
```

But this fails to work. From a syntactic standpoint there's no way to convert this back to an equivalent lambda expression. And although the identifier `x` is bound in the list of binding expressions, the value of `x` is only available inside the body of the `let` expression.

There is, however, an alternative definition of `let` called `let*`. In this case the following would work.

```
> (let* ([x 4]
         [y 5]
         [z (* 2 x)])
    (+ z y))
13
```

The difference is that with `let*` the value of an identifier is available immediately after it's bound, whereas with `let` the identifier values are only available after *all* the identifiers are bound.

Here's another slight variation where `let` *does* work.

```
> (let ([x 4]
        [y 5])
    (let ([z (* 2 x)])
      (+ z y)))
13
```


Copyrighted material

In this case the second `let` is within the lexical environment of the first `let` (but as we've seen, `let*` more efficiently encodes this type of nested construct). Hence `x` is available for use in the expression `(* 2 x)`.

Conditional Expressions: It's All About Choices

The ability of a computer to alter its execution path based on an input is an essential component of its architecture. Without this a computer cannot compute. In most programming languages this capability takes the form of something called a *conditional expression*, and in Racket it's expressed (in its most general form) as a `cond` expression.

Suppose you're given the task to write a function that returns a value that indicates whether a number is divisible by 3 only, divisible by 5 only, or divisible by both. One way to accomplish this is with the following code.

```
(define (div-3-5 n)
  (let ([div3 (= 0 (remainder n 3))]
        [div5 (= 0 (remainder n 5))])
    (cond [(and div3 div5) 'div-by-both]
          [div3 'div-by-3]
          [div5 'div-by-5]
          [else 'div-by-neither])))
```

The `cond` form contains a list of expressions. For each of these expressions, the first element contains some type of test, which if it evaluates to true, evaluates the second element and returns its value. Note that in this example the test for divisibility by 3 and 5 must come first. Here are trial runs:

```
> (div-3-5 10)
'div-by-5

> (div-3-5 6)
'div-by-3

> (div-3-5 15)
'div-by-both

> (div-3-5 11)
'div-by-neither
```

A simplified version of `cond` is the `if` form. This form consists of a single test (the first subexpression) that returns its second argument (after it's evaluated) if the test evaluates to true; otherwise it evaluates and returns the third argument. This example simply tests whether a number is even or odd.

```
(define (parity n)
  (if (= 0 (remainder n 2)) 'even 'odd))
```


Copyrighted material

If we run some tests:

```
> (parity 5)
'odd
> (parity 4)
'even
```

Both `cond` and `if` are expressions that return values. There are occasions where one simply wants to conditionally execute some sequence of steps if a condition is true or false. This usually involves cases where some side effect like printing a value is desired and returning a result is not required. For this purpose, Racket provides `when` and `unless`. If the conditional expression evaluates to true, `when` evaluates all the expressions in its body; otherwise it does nothing.

```
> (when (> 5 4)
    (displayln 'a)
    (displayln 'b))
a
b

> (when (< 5 4) ; doesn't generate output
    (displayln 'a)
    (displayln 'b))
```

The `unless` form behaves in exactly the same way as `when`; the difference is that `unless` evaluates its body if the conditional expression is not true.

```
> (unless (> 5 4) ; doesn't generate output
    (displayln 'a)
    (displayln 'b))

> (unless (< 5 4)
    (displayln 'a)
    (displayln 'b))
a
b
```

I'm Feeling a Bit Loopy!

Loops (or iteration) are the bread and butter of any programming language. With the discussion of loops, invariably the topic of *mutability* comes up. Mutability of course implies change. Examples of mutability are assigning values to variables (or worse, changing a value embedded in a data structure such as a vector). A function is said to be *pure* if no mutations (or side effects, like printing out a value or writing to a file—also forms of mutation) occur within the body of a function. Mutations are generally to be avoided if possible. Some languages, such as Haskell, go out of their way to avoid

Copyrighted material

this type of mischief. A Haskell programmer would rather walk barefoot through a bed of glowing, hot coals than write an impure function.

There are many good reasons to prefer pure functions, such as something called referential transparency (this mouthful simply means the ability to reason about the behavior of your program). We won't be quite so persnickety and will make judicious use of mutation and impure functions where necessary.

Suppose you're given the task of defining a function to add the first *n* positive integers. If you're familiar with a language like Python (an excellent language in its own right), you might implement it as follows.

```
def sum(n):
    s = 0
    while n > 0:
     ❶ s = s + n
     ❷ n = n - 1
    return s
```

This is a perfectly good function (and a fairly benign example of using mutable variables) to generate the desired sum, but notice both the variables `s` and `n` are modified ❶ ❷. While there's nothing inherently wrong with this, these assignments make the implementation of the function `sum` impure.

Purity

Before we get down and dirty, let's begin by seeing how we can implement looping using only pure functions. *Recursion* is the custom when it comes to looping or iteration in Racket (and all functional programming languages). A recursive function is just a function defined in terms of itself. Here's a pure (and simple) recursive program to return the sum of the first *n* positive integers.

```
(define (sum n)
❶ (if (= 0 n) 0
   ❷ (+ n (sum (- n 1)))))
```

As you can see, we first test whether `n` has reached 0 ❶, and if so we simply return the value 0. Otherwise, we take the current value of `n` and *recursively* add to it the `sum` of all the numbers less than `n` ❷. For the mathematically inclined, this is somewhat reminiscent of how a proof by mathematical induction works where we have a base case ❶ and the inductive part of the proof ❷.

Let's test it out.

```
> (sum 100)
5050
```

There's a potential problem with the example we have just seen. The problem is that every time a recursive call is made, Racket must keep track of

Copyrighted material

where it is in the code so that it can return to the proper place. Let's take a deeper look at this function.

```
(define (sum n)
  (if (= 0 n) 0
   ❶ (+ n (sum (- n 1)))))
```

When the recursive call to `sum` is made ❶, there's still an addition remaining to be done after the recursive call returns. The system must then remember where it was when the recursive call was made so that it can pick up where it left off when the recursive call returns. This isn't a problem for functions that don't have to nest very deeply, but for large depths of recursion, the computer can run out of space and fail in a dramatic fashion.

Racket (and virtually all Scheme variants) implement something called *tail call optimization* (the Racket community says this is simply the proper way to handle tail calls rather than an optimization, but *tail call optimization* is generally used elsewhere). What this means is that if a recursive call is the very last call being made, there's no need to remember where to return to since there are no further computations to be made within the function. Such functions in effect behave as a simple iterative loop. This is a basic paradigm for performing looping computations in the Lisp family of languages. You do, however, have to construct your functions in such a way as to take advantage of this feature. We can rewrite the summing function as follows.

```
(define (sum n)
  (define (s n acc)
 ❶ (if (= 0 n) acc
     ❷ (s (- n 1) (+ acc n))))
  (s n 0))
```

Notice that `sum` now has a local function called `s` that takes an additional argument called `acc`. Also notice that `s` calls itself recursively ❷, but it's the last call in the local function; hence tail call optimization takes place. This all works because `acc` accumulates the sum and passes it along as it goes. When it reaches the final nested call ❶, the accumulated value is returned.

Another way to do this is with a named `let` form as shown here.

```
(define (sum n)
  (let loop ([n n] [acc 0])
    (if (= 0 n) acc
        (loop (- n 1) (+ acc n)))))
```

The named `let` form, similar to the normal `let`, has a section where local variables are initialized. The expression `[n n]` may at first appear puzzling, but what it means is that the first `n`, which is local to the `let`, is initialized with the `n` that the `sum` function is called with. Unlike `define`, which simply binds an identifier with a function body, the named `let` binds the identifier (in this case `loop`), evaluates the body, and returns the value resulting from calling the function with the initialized parameter list. In this example the

Copyrighted material

function is called recursively (which is the normal use case for a named `let`) as indicated by the last line in the code. This is a simple illustration of a side-effect-free looping construct favored by the Lisp community.

The Power of the Dark Side

Purity is good, as far as it goes. The problem is that staying pure takes a lot of work (especially in real life). It's time to take a closer look at the dreaded `set!` form. Note that an exclamation point at the end of any built-in Racket identifier is likely there as a warning that it's going to do something impure, like modify the program state in some fashion. A programming style that uses statements to change a program's state is said to use *imperative programming*. In any case, `set!` reassigns a value to a previously bound identifier. Let's revisit the Python `sum` function we saw a bit earlier. The equivalent Racket version is given below.

```
(define (sum n)
  (let ([s 0])   ; initialize s to zero
    (do ()       ; an optional initializer statement can go here
      ((< n 1))  ; do until this becomes true
      (set! s (+ s n))
      (set! n (- n 1)))
    s))
```

Racket doesn't actually have a `while` statement (this has to do with the expectation within the Lisp community that recursion *should* be the go-to method for recursion). The Racket `do` form functions as a `do-until`.

If you're familiar with the C family of programming languages, then you will see that the full form of the `do` statement actually functions much like the C `for` statement. One way to sum the first *n* integers in C would be as follows:

```
int sum(int n)
{
  int s = 0;
  for (i=1; i<= n; i++) // initialize i=1, set i = i+1 at each iteration
                        // do while i<= n
  {
     s = s + i;
  }
  return s;             // return s
}
```

Here's the Racket equivalent:

```
(define (sum n)
❶ (let ([s 0])
  ❷ (do ([i 1 (add1 i)])   ; initialize i=1, set i = i+1 at each iteration
```


Copyrighted material

```
❸ ((> i n) s)          ; do until i>n, then return s
❹ (set! s (+ s i)))))
```

In the above code we first initialize the local variable s (which holds our sum) to 0 ❶. The first argument to do ❷ initializes i (i is local to the do form) to 1 and specifies that i is to be incremented by 1 at each iteration of the loop. The second argument ❸ tests whether i has reached the target value and if so returns the current value of s. The last line ❹ is where the sum is actually computed by increasing the value of s with the current value of i via the set! statement.

The value of forms such as do with the set! statement is that many algorithms are naturally stated in a step-by-step fashion with variables mutated by equivalents to the set! statement. This helps to avoid the mental gymnastics needed to convert such constructs to pure recursive functions.

In the next section, we examine the for family of looping variants. Here we will see that Racket's for form provides a great deal of flexibility in how to manage loops.

The for Family

Racket provides the for form along with a large family of for variants that should satisfy most of your iteration needs.

A Stream of Values

Before we dive into for, let's take a look at a couple of Racket forms that are often used in conjunction with for: in-range and in-naturals. These functions return something we haven't seen before called a *stream*. A stream is an object that's sort of like a list, but whereas a list returns all its values at once, a stream only returns a value when requested. This is basically a form of *lazy evaluation*, where a value is not provided until asked for. For example, (in-range 10) will return a stream of 10 values starting with 0 and ending with 9. Here are some examples of in-range in action.

```
> (define digits (in-range 10))
> (stream-first digits)
0

> (stream-first (stream-rest digits))
1

> (stream-ref digits 5)
5
```

In the code above, (in-range 10) defines a sequence of values 0, 1, . . . , 9, but digits doesn't actually contain these digits. It basically just contains a specification that will allow it to return the numbers at some later time. When (stream-first digits) is executed, digits gives the first available value, which in this case is the number 0. Then (stream-rest digits) returns the

Copyrighted material

stream containing the digits after the first, so that (stream-first (stream-rest digits)) returns the number 1. Finally, stream-ref returns the *i*-th value in the stream, which in this case is 5.

The function in-naturals works like in-range, but instead of returning a specific number of values, in-naturals returns an infinite number of values.

```
> (define naturals (in-naturals))
> (stream-first naturals)
0

> (stream-first (stream-rest naturals))
1

> (stream-ref naturals 1000)
1000
```

How the stream concept is useful will become clearer as we see it used within some for examples. We'll also met some useful additional arguments for in-range.

for in the Flesh

Here's an example of for in its most basic form. The goal is to print each character of the string "Hello" on a separate line.

```
> (let* ([h "Hello"]
       ❶ [l (string-length h)])
    ❷ (for ([i (in-range l)])
      ❸ (display (string-ref h i))
       (newline)))
H
e
l
l
o
```

We capture the string-length ❶ and use this length with the in-range function ❷. for then uses the resulting stream of values to populate the identifier i, which is used in the body of the for form to extract and display the characters ❸. In the prior section it was pointed out that in-range produces a sequence of values, but it turns out that in the context of a for statement, a positive integer can also produce a stream as the following example illustrates.

```
> (for ([i 5]) (display i))
01234
```

The for form is quite forgiving when it comes to the type of arguments that it accepts. It turns out that there's a much simpler way to achieve our goal.

Copyrighted material

```
> (for ([c "Hello"])
    (display c)
    (newline))
H
e
l
l
o
```

Instead of a stream of indexes, we have simply provided the string itself. As we'll see, `for` will accept many built-in data types that consist of multiple values, like lists, vectors, and sets. These data types can also be converted to streams (for example, by `in-list`, `in-vector`, and so on), which in some cases can provide better performance when used with `for`. All expressions that provide values to the identifier that `for` uses to iterate over are called *sequence expressions*.

It's time to see how we can make use of the mysterious `in-naturals` form introduced above.

```
> (define (list-chars str)
    (for ([c str]
          [i (in-naturals)])
      (printf "~a: ~a\n" i c)))

> (list-chars "Hello")
0: H
1: e
2: l
3: l
4: o
```

The `for` form inside the `list-chars` function now has *two* sequence expressions. Such sequence expressions are evaluated in parallel until one of the expressions runs out of values. That is why the `for` expression eventually terminates, even though `in-naturals` provides an infinite number of values.

There is, in fact, a version of `for` that *does not* evaluate its sequence expressions in parallel: it's called `for*`. This version of `for` evaluates its sequence expressions in a nested fashion as the following example illustrates.

```
> (for* ([i (in-range 2 7 4)]
         [j (in-range 1 4)])
    (display (list i j (* i j)))
    (newline))
(2 1 2)
(2 2 4)
(2 3 6)
(6 1 6)
```


Copyrighted material

```
(6 2 12)
(6 3 18)
```

In this example we also illustrate the additional optional arguments that `in-range` can take. The sequence expression `(in-range 2 7 4)` will result in a stream that starts with the number 2, and increment that value by 4 with each iteration. The iteration will stop once the streamed value reaches one less than 7. So in this expression, `i` is bound to 2 and 6. The expression `(in-range 1 4)` does not specify a step value, so the default step size of 1 is used. This results in `j` being bound to 1, 2, and 3.

Ultimately, `for*` takes every possible combination of `i` values and `j` values to form the output shown.

Can You Comprehend This?

There is a type of notation in mathematics called set-builder notation. An example of set-builder notation is the expression $\left\{x^2 \mid x \in \mathbb{N}, x \leq 10\right\}$. This is just the set of squares of all the natural numbers between 0 and 10. Racket provides a natural (pun intended) extension of this idea in the form of something called a *list comprehension*. A direct translation of that mathematical expression in Racket would appear as follows.

```
> (for/list ([x (in-naturals)] #:break (> x 10)) (sqr x))
'(0 1 4 9 16 25 36 49 64 81 100)
```

The `#:break` keyword is used to terminate the stream generated by `in-naturals` once all the desired values have been produced. Another way to do this, without having to resort to using `#:break`, would be with `in-range`.

```
> (for/list ([x (in-range 11)]) (sqr x))
'(0 1 4 9 16 25 36 49 64 81 100)
```

If you only wanted the squares of even numbers, you could do it this way:

```
> (for/list ([x (in-range 11)] #:when (even? x)) (sqr x))
'(0 4 16 36 64 100)
```

This time the `#:when` keyword was brought into play to provide a condition to filter the values used to generate the list.

An important difference of `for/list` over `for` is that `for/list` does not produce any side effects and is therefore a pure form, whereas `for` is expressly for the purpose of producing side effects.

More Fun with for

Both `for` and `for/list` share the same keyword parameters. Suppose we wanted to print a list of squares, but don't particularly like the number 5. Here's how it could be done.

Copyrighted material

```
> (for ([n (in-range 1 10)] #:unless (= n 5))
    (printf "~a: ~a\n" n (sqr n)))
1: 1
2: 4
3: 9
4: 16
6: 36
7: 49
8: 64
9: 81
```

By using `#:unless` we've produced an output for all values, $1 \leq n < 10$, unless $n = 5$.

Sometimes it's desirable to test a list of values to see if they all meet some particular criteria. Mathematicians use a fancy notation to designate this called the universal quantifier, which looks like this $\forall$ and means "for all." An example is the expression $\forall x \in \{2,4,6\}, x \bmod 2 = 0$, which is literally interpreted as "for all x in the set $\{2,4,6\}$, the remainder of x after dividing by 2 is 0." This just says that the numbers 2, 4, and 6 are even. The Racket version of "for all" is `for/and`.

Feed the `for/and` form a list of values and a Boolean expression to evaluate the values. If each value evaluates to true, the entire `for/and` expression returns true; otherwise it returns false. Let's have a go at it.

```
> (for/and ([x '(2 4 6)]) (even? x))
#t

> (for/and ([x '(2 4 5 6)]) (even? x))
#f
```

Like `for`, `for/and` can handle multiple sequence expressions. In this case, the values in each sequence are compared in parallel.

```
> (for/and ([x '(2 4 5 6)]
            [y #(3 5 9 8)])
    (< x y))
#t
> (for/and ([x '(2 6 5 6)]
            [y #(3 5 9 8)])
    (< x y))
#f
```

Closely related to `for/and` is `for/or`. Not to be outdone, mathematicians have a notation for this as well: it's called the existential quantifier, $\exists$. For example, they express the fact that there *exists* a number in the set $\{2,7,4,6\}$ greater than 5 with the expression $\exists x \in \{2,7,4,6\}, x > 5$.

Copyrighted material

```
> (for/or ([x '(2 7 4 6)]) (> x 5))
#t

> (for/or ([x '(2 1 4 5)]) (> x 5))
#f
```

Suppose now that you not only want to know whether a list contains a value that meets a certain criterion, but you want to extract the first value that meets the criterion. This is a job for `for/first`:

```
> (for/first ([x '(2 1 4 6 7 1)] #:when (> x 5)) x)
6

> (for/first ([x '(2 1 4 5 2)] #:when (> x 5)) x)
#f
```

The last example demonstrates that if there is no value that meets the criterion, `for/first` returns false.

Correspondingly, if you want the last value, you can use `for/last`:

```
> (for/last ([x '(2 1 4 6 7 1)] #:when (> x 5)) x)
7
```

The `for` family of functions is fertile ground for exploring parallels between mathematical notation and Racket forms. Here is yet another example. To indicate the sum of the squares of the integers from 1 to 10, the following notation would be employed:

$$S = \sum_{i=1}^{10} i^2$$

The equivalent Racket expression is:

```
> (for/sum ([i (in-range 1 11)]) (sqr i))
385
```

The equivalent mathematical expression for products is

$$p = \prod_{i=1}^{10} i^2$$

which in Racket becomes

```
> (for/product ([i (in-range 1 11)]) (sqr i))
13168189440000
```

Most of the `for` forms discussed above come in a starred version (for example `for*/list`, `for*/and`, `for*/or`, and so on). Each of these works by evaluating their sequence expressions in a nested fashion as described for `for*`.

Copyrighted material

Time for Some Closure

Suppose you had $100 in the bank and wanted to explore the effects of compounding with various interest rates. If you're not familiar with how compound interest works (and you very well should be), it works as follows: if you have n_0 in a bank account that pays i periodic interest, at the end of the period you would have this:

$$n_1 = n_0 + n_0 i = n_0(1 + i)$$

Using your $100 deposit as an example, if your bank pays 4 percent (i = 0.04) interest per period (good luck getting that rate at a bank nowadays), you would have the following at the end of the period:

$$100 + 100 \cdot 4\% = 100(1 + 0.04) = 104$$

One way to do this is to create a function that automatically updates the balance after applying the interest rate. A clever way to compute this in Racket is with something called a *closure*, which we use in the following function:

```
(define (make-comp bal int)
  (let ([rate (add1 (/ int 100.0))])
 ❶ (λ () (set! bal (* bal rate))  (round bal))))
```

Notice that this function actually returns another function—the lambda expression (λ . . .) ❶—and that the lambda expression contains variables from the defining scope. We shall explain how this works shortly.

In the code above, we've defined a function called `make-comp` which takes two arguments: the starting balance and the interest rate percentage. The `rate` variable is initialized to $(1 + i)$. Rather than return a number, this function actually returns another function. The returned function is designed in such a way that every time it's called (without arguments) it updates the balance by applying the interest and returns the new balance. You might think that once `make-comp` returns the lambda expression, the variables `bal` and `rate` would be undefined, but not so with closures. The lambda expression is said to *capture* the variables `bal` and `rate`, which are available within the lexical environment where the lambda expression is defined. The fact that the returned function contains the variables `bal` and `rate` (which are defined outside of the function) is what makes it a closure.

Let's try this out and see what happens.

```
> (define bal (make-comp 100 4))

> (bal)
104.0

> (bal)
108.0
```


Copyrighted material