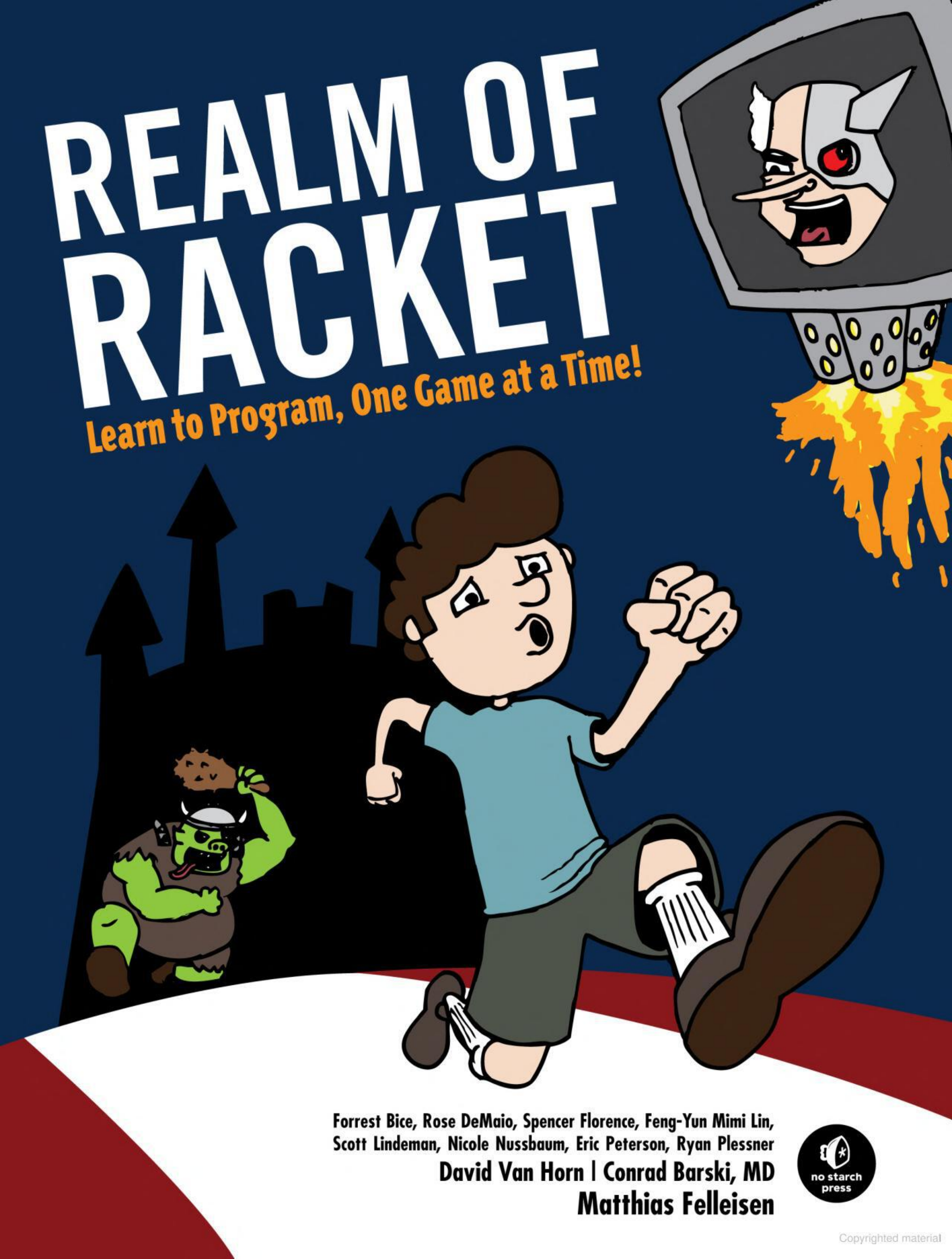
REALM OF RACKET

Learn to Program, One Game at a Time!



Forrest Bice, Rose DeMaio, Spencer Florence, Feng-Yun Mimi Lin,
Scott Lindeman, Nicole Nussbaum, Eric Peterson, Ryan Plessner

David Van Horn | Conrad Barski, MD

Matthias Felleisen

no starch press

REALM OF RACKET

Learn to Program, One Game at a Time!

```
(list

 Forrest Bice
 Rose DeMaio
 Spencer Florence
 Feng-Yun Mimi Lin
 Scott Lindeman
 Nicole Nussbaum
 Eric Peterson
 Ryan Plessner

 David Van Horn
 Matthias Felleisen

 Conrad Barski, MD)
```

no starch press

San Francisco

Printed in USA

First printing

17 16 15 14 13   1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-491-2
ISBN-13: 978-1-59327-491-7

Publisher: William Pollock
Production Editor: Alison Law
Cover Illustration: Eric Peterson
Illustrations: Feng-Yun Mimi Lin and Eric Peterson
Interior Design and Composition: Rose DeMaio and Nicole Nussbaum
Developmental Editors: Keith Fancher and William Pollock
Copyeditor: Marilyn Smith
Proofreader: Paula L. Fleming

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
38 Ringold 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:

```
Felleisen, Matthias.
  Realm of Racket : learn to program, one game at a time! / by Matthias Felleisen, Conrad
Barski, Forrest Bice, Rose DeMaio, Spencer Florence, Feng-yun Mimi Lin, Scott Lindeman,
Nicole Nussbaum, Eric Peterson, Ryan Plessner, and David Van Horn.
       pages cm
  Includes index.
  ISBN-13: 978-1-59327-491-7
  ISBN-10: 1-59327-491-2
 1.  Racket (Computer program language) 2.  LISP (Computer program language) 3.  Computer
programming.  I. Title.
  QA76.73.R33F45 2013
  005.1--dc23
                                                                  2013002308
```

BRIEF CONTENTS

CONTENTS IN DETAIL

Acknowledgments

We gratefully acknowledge the generous support of Brian Wenzinger, an alumnus of Northeastern University's College of Computer and Information Science.

The College's dean, Larry Finkelstein, and his staff—especially Kirsten Anderson, Doreen Hodgkin, and Jennifer Wong—provided great help along the way. Thank you.

Without Matthew Flatt (Utah), Robby Findler (Northwestern), and the rest of the PLT folks, there would be no Racket, no DrRacket, and no *Realm of Racket*.

Along the way we got a lot of support from various people at NoStarch Press: Alison Law cheerfully assisted with various aspects of the manuscript production; Marilyn Smith meticulously copyedited our draft; Serena Yang patiently worked with us on the cover design; and Jessica Miller got the marketing going. A big thank you also goes to Keith Fancher, our acquisitions editor, and Bill Pollock, the publisher, who trusted a team of eight freshman students to produce a full-fledged book, cover to cover and on time. Thank you for your trust.

Finally, cheers to our co-author, Conrad Barski, for getting us launched on this adventure through Racket land. We also owe thanks to Eric Chin, Pranav Gandhi, James Grammatikos, Cole Levi, Jack Noble, Alex Schwartz, and Brendan Wilson, who participated in the initial explorations for this project. In addition, Forrest would like to acknowledge and express his humble gratitude for the diverse and unwavering support his mom and dad, Sarah and Steve, have lovingly provided and continue to provide. Rose would like to thank her family and friends for being incredibly supportive of the project and for understanding when "the book" was the only explanation given for her frequent absences from activities in the past few months. She also wishes to thank all her co-authors for three semesters of forever quotable Thursday evenings. Mimi would like to thank her parents for giving her learning opportunities that exceeded their own. Furthermore, she is grateful to Matthias and David for opening up this rare educational experience. Scott says thank you to his little brother Steve, as retribution for constantly giving him a hard time. Nicole thanks her parents for always supporting her. She also thanks her co-authors for giving her Thursday nights purpose and joy. Thanks to MF and DVH, who could make her laugh even after fourteen straight hours of edits. Eric would like to thank his friends and family for always being so supportive of his doodles, and also David and Matthias for giving him the opportunity to use his doodles for good. David would like to thank Marisa, for everything. Matthias acknowledges his wife, Helga, for patiently waiting many Thursday evenings and his advisor, Dan Friedman, for teaching him Lisp and English.

;; Preface
(Hello World)

```
#|
Step into Realm of Racket, a book that takes you on a unique
journey into the land of computer programming. In the style
of Conrad Barski's Land of Lisp, this book teaches you how to
program in Racket by creating a series of games. Racket is a
friendly mutation of Lisp that's perfect for all, from those who
want to launch their education in computer science to those look-
ing to expand their knowledge and experience in programming.
|#
```

Why Would I Want to Learn About Racket?

You've certainly heard of JavaScript, Perl, Python, and Ruby. But what about Racket? Just because it's not the most mainstream programming language doesn't mean you should discount its capabilities. Racket allows functional programming and other different paradigms that even hard-core programmers have never seen before. Get ready for the excursion. Even after you get through *Realm of Racket*, there is a lot to explore.

Who Should Read This Book?

Our mantra is "by freshmen, for freshmen," but that doesn't mean you should drop this book if you are a sophomore or an industry professional. True, we were freshmen when we started writing this book, but our mantra means only that this book was written for you by peers who have a special interest in programming and want to explore it in a new, fun way. So you see why our mantra is what it is—it would have been a bit of a mouthful to say, "By people who have a special interest in programming and want to explore it in a new, fun way, for people who have a special interest in programming and want to explore it in a new, fun way." And our recent expedition into the realm of Racket has enabled us to write this book with genuine empathy for a novice learner.

Regardless of your programming background, many of the topics in this book will be new to you, and much of what you've learned before will appear in a new light. This book is written for those who are truly inquisitive and interested in exploring a unique world of programming, so really we are all "freshmen" in this context.

What Teaching Approach Is Used?

It won't take you long to realize that this is not your typical programming textbook. We decided to present the material in a way that is engaging and really sticks—with games and comics.

In this book, we will teach you various topics through coding games, including a text-based game, some old-school games like Snake, and others that we invented ourselves. Along the way, you will need to use your programming skills to help a character named Chad navigate the dungeons of DrRacket.

Can I Skip Chapters?

You might think you can skip ahead and save Chad right away, but we highly recommend that you read this book from front to back. Each chapter depends on the knowledge you learned from the previous one, and we don't want you to miss out on any of Chad's adventures.

Anything Else I Should Know?

The source code of our games is available with the code base of Racket. Once you download Racket, navigate to the Racket installation and look for the **collects/realm/** folder. All the game code is there for you to explore, modify, and experiment with.

Finally, the book comes with its own website. Visit `realmofracket.com` and keep visiting—you never know what you'll find there. Onward!

;; Introduction (Open Paren)

#|
So you think you know how to program because you took an introductory course or two. Or perhaps you read a book that taught you programming in 13 days. And then you picked up this book, which is full of parentheses and comics. Doesn't it look different from what you have seen in the past?

The programs you see here look like those that we encountered in our first programming courses. You might be wondering why anyone would teach such a weird-looking programming language and why we find it so exciting that we would write a whole book about it.

Or maybe you've heard others rave about the Lisp language and thought, "Boy, Lisp sure looks different from other languages that people talk about. Maybe I should pick up a Lisp book." Either way, you're now holding a book about a programming language in the Lisp family. And that whole family is very cool and unusual and fun. You won't regret it.
|#

(.1 What Makes Lisp So Cool and Unusual?

Lisp is a highly expressive language. With Lisp, you take the most complicated problems and express their solutions in a clear and appropriate way. If Lisp doesn't have the means to do so, you change Lisp.

Lisp will change your mind, too. Eric Raymond, a well-known "hacker"—in the original, positive sense of the word—once wrote that "Lisp is worth learning for a different reason—the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot."

Here is what we did with Lisp in the very first class of our very first course: we launched a rocket. Well, we didn't launch a real rocket, but the animation looked cool, and it was just a few lines of code. After a couple of weeks into the course, we wrote our first interactive graphical game. Yes, our program used mouse clicks, clock ticks, and keyboard events to control a nifty little Snake game. Then we wrote an interpreter for the language that we used in the course. Did you get to write an interpreter for your language in your course? And before we knew it, we wrote distributed games. In case you don't know what "distributed" means, our games ran on many computers, and at each computer, some person interacted with the game or some program played the game, and all these computers exchanged messages to make everything work together. And can you believe that some of us had never programmed before?

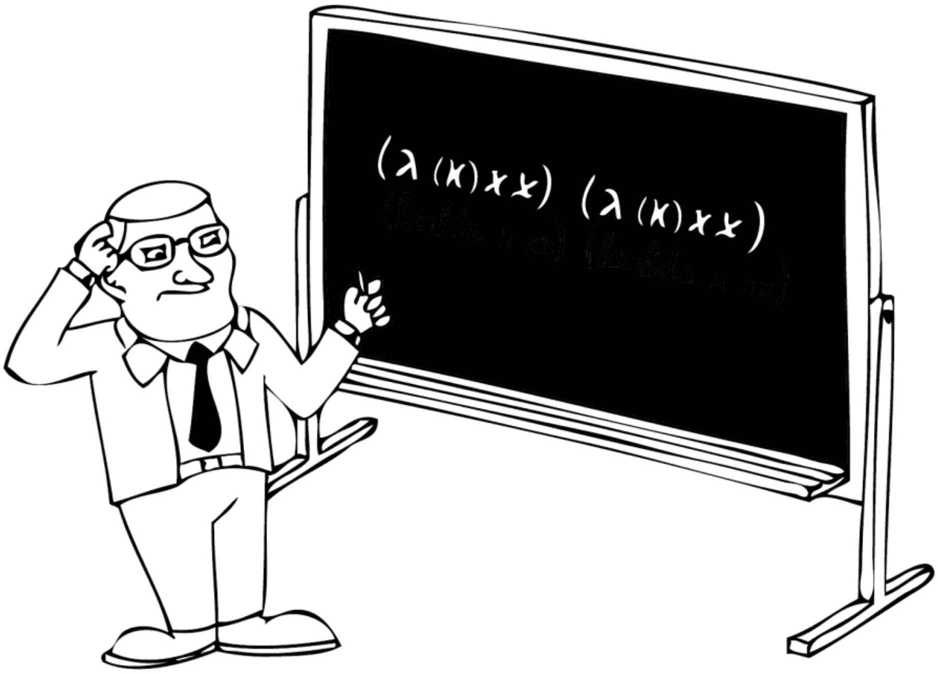
So Lisp is different. And the Lisp we use is Racket, with which even children can quickly feel at home. Once you have experienced this kind of programming, it will become a part of you, and you will dream about it. You will always strive to mimic this style in the languages you must use to earn a living. You'll say to yourself, "That's kind of how I'd do it in Lisp." That's the power only Lisp can give you.

(.2 Where Did Lisp Come From?

At this point, you may start wondering why your instructor didn't tell you about Lisp. You may think that it may be something new that he hadn't heard of yet. Sadly though, it's the opposite. The idea of Lisp is truly ancient, and yet, in some ways, most existing flavors of Lisp are more advanced than any other kind of language. But Lisp's history is very different from that of other languages, and that could be why people overlook it.

Here is what we know from the anecdotes our professors told us. Way back in the 1930s, the first True Wonk of Programming walked the Earth. His name was Al—though his birth certificate said Alonzo Church—and he invented a new kind of calculus: **lambda calculus**. In this calculus, everything was a function, and every function that counted was something. All in all, it was functional and it functioned, although all this programming happened with paper and pencil because people had just figured out electricity.

Soon enough, the Earth was aflame in a great war with serious consequences on the world of programming. Governments sponsored science projects, and several of these projects created real computers. All computers, and especially the very first ones, are plain, dumb pieces of electrical hardware. As a matter of fact, the early computers—with names such as ENIAC and Zuse's Z3—were so primitive that "programming" them involved flipping switches or patching cables to physically encode each operation. Those dark days saw a lot of experimentation with different computer architectures and an explosion of different ways of "programming" them.



Naturally, people invented the idea of a programming language because it is the only way to make dumb computers truly useful. However, "language" meant nothing but a fairly thin veil over the underlying hardware. With these languages, programmers gave machines instructions. It made programming easier than flipping switches and patching cables, but it didn't make it easy. Programmers had to think about the machine when they programmed, and their programs remained tied to specific computers.

Programming languages needed to evolve to survive beyond the confines of a specific machine. Thus, the 1950s saw the arrival of new types of software, including the most important: compilers and interpreters. A compiler can take something that looks like plain arithmetic and convert it automatically into a format that the computer can execute. An interpreter is similar, although it performs the actions described in a human-written program directly; there is no intermediate step that converts it all the way down to a computer format. Best of all, compilers and interpreters are software. This means that a competent programmer can change an existing compiler or interpreter a bit to make it work on a different computer, without changing the language that programmers use to interact with the compiler or interpreter. As a result, computer programmers could suddenly write programs in a notation that was mostly independent of a specific computer.

Nevertheless, programming remained a task of giving machines instructions. Take FORTRAN, the earliest language with a compiler. Its designers created it to help scientists program, and scientists still use FORTRAN today. Just typing a FORTRAN program makes you shuffle bits and bytes in the machine, and once you're finished, you can barely see the mathematical problem you wanted to solve.

Another early language in the same mold as FORTRAN is ALGOL 60. While FORTRAN was made in America, ALGOL emerged in Europe, designed by a committee of computer scientists from all over the world. Together, FORTRAN and ALGOL spawned a long series of programming languages, called the ALGOL family. All members of this family are made from more or less the same material and for the same kind of programmers—those who love the machine more than the problem. You may have heard of some of these languages, such as C, Pascal, C++, and Java. To this day, most college courses use the ALGOL family to teach the first course on programming. Your instructors may even have told you that all languages are so similar that once you have seen one of them, you have seen all of them. Although this statement might be true for the ALGOL family of languages, it isn't true for Lisp, a language nearly as old as FORTRAN and ALGOL.

Lisp's beginnings are quite humble. Also in the late 1950s, John McCarthy—a computer scientist who worked at MIT, the best college in East Cambridge—came across Al's old papers on lambda calculus. The papers were difficult to read because they had been written in the days before people had real computers. They sent John daydreaming. When he woke up, he knew that he was sick of programming computers in the dumb ways of machine language or FORTRAN. He wanted an intelligent way to go about it. So one day he gathered together his researchers and challenged them to build a programming language that wasn't about the bits and bytes in a computer. Instead, he wanted to create a language that would help programmers solve problems without forcing them to think about the elements of a machine.

John's first example was about lists: lists of ideas, lists of tasks, lists of insights, and even lists of programs. To deal with lists, a program should provide lists and functions for processing lists—never mind how the computer really deals with all of this inside. Better still, a language should be able to "talk" about itself and about programming. In short, he wanted Lisp, a language so elegant and powerful that even writing an interpreter for Lisp in Lisp itself would take only about 50 lines of computer code.

Before John knew it, his people made Lisp real. The language was indeed small, and it was really possible to write a Lisp interpreter in Lisp in a few dozen lines of code. Because it was so easy to write a Lisp interpreter, many people wrote one. As it turned out, everyone tinkered with the original small Lisp interpreter. Soon enough, there wasn't just one Lisp but a lot of Lisps. Fortunately, all these Lisps shared the essential traits of John's original ideas so that Lisp programmers could easily exchange ideas. That's why Lisp is a family of programming languages, not just a single programming language.

(.3 What Does Lisp Look Like?

Now you know that Lisp is cool, old, and an entire family of languages. You also know that we are totally excited about one member of this family: Racket. Before you become impatient with us, let us show you some Lisp code so that this introduction doesn't become all talk and no action.

At some level, Lisp code isn't as different as it may seem at first. Here are some valid Lisp expressions, and we're sure you can guess what they compute:

```
(* 1 1)
(- 8 (* 2 3))
(sqrt 9)
```

If you answered 1, 2, and 3, then you've already figured out how to read Lisp code. It looks like arithmetic, except that the functions—addition, subtraction, multiplication, and square root—come before their operands, and expressions are surrounded by parentheses.

You may wonder why in the world Lisp breaks with the centuries-old tradition of infix notation, but take a look at this:

```
(+ 1 2 3 4 5 6 7 8 9 0)
```

For many functions, it makes a lot of sense to supply a lot of arguments at once, and with prefix notation, doing so is easy. But programming is also about defining your own operators and writing expressions that use these and the ones that come with the language. So in C++ programs, you may see things like this:

```
(foo<bar>)*g++.baz(!&qux::zip->ding());
```

Can you explain the order in which the subexpressions of this complex beast are evaluated? No one can. Everyone must dissect the expression first to understand it. Dissecting means putting in more parentheses, either in your head or on paper, after looking up which operators have the highest precedence. Do you recall your struggles with precedence in grade school? When you read Lisp code, this confusion never happens:

```
(sqrt (+ (sqr 3) (sqr 4)))
```

Here, you see a deeply nested expression, and yet the parentheses immediately tell you in which order things are evaluated. The parentheses also tell you that the operators—`sqrt`, +, and `sqr`—are involved, because you know that every left parenthesis—called "open" when Lispers speak—is followed by an operator. And guess what the right parentheses are called in Lisp-speak? You got it: "close." That's how easy it is to read Lisp. Once you're used to it, which may take a day or two, you will see its advantages.

However, Lisp isn't just about numbers or arithmetic; it's also about list processing. So you should be curious about what lists look like in Lisp. Here are some examples:

```
(list 1 2 3 4 5 6 7 8 9 0)
(list (list 1 3 5 7 9) (list 2 4 6 8 0))
(list (list 'hello 'world)
      (list (list 'it 'is) 2063)
      (list 'and 'we 'love 'Racket))
```

The first list consists of 10 digits; the second one groups these digits into two lists, which then become the elements of another list; and the third example shows that while lists can be deeply nested, Lisp comes with symbolic values, too.

Of course, a language that is about lists has even better ways to write lists:

```
'(1 2 3 4 5 6 7 8 9 0)
'((1 3 5 7 9) (2 4 6 8 0))
'((hello world)
  ((it is) 2063)
  (and we love Racket))
```

These first three examples should look familiar; they are abbreviations for the preceding list examples. See how little you need to write for a list of lists with symbolic information inside? One last example should inspire some awe:

```
'(sqrt (+ (sqr 3) (sqr 4)))
```

By adding a quote to the left of a piece of Lisp code, we turn it into a piece of data. Because the two expressions are equivalent, we could have written this:

```
(list 'sqrt
      (list '+
            (list 'sqr 3)
            (list 'sqr 4)))
```

Both create lists that contain a list, which contains a symbolic representation of addition and two more lists. The quoted expression creates a piece of data that captures all the structure that is in the program expression itself: its nesting structure, its numbers, and its symbols. If you had used string quotes to turn the expression into a piece of data, all you would have is a string. Before you could recover the expression's structure and organization, you would need to complete your entire computer science undergraduate major.

In Lisp, all it takes is one keystroke on your keyboard—one character on the screen. You cannot do anything like this without a Lisp. Now that's cool. And it's powerful.

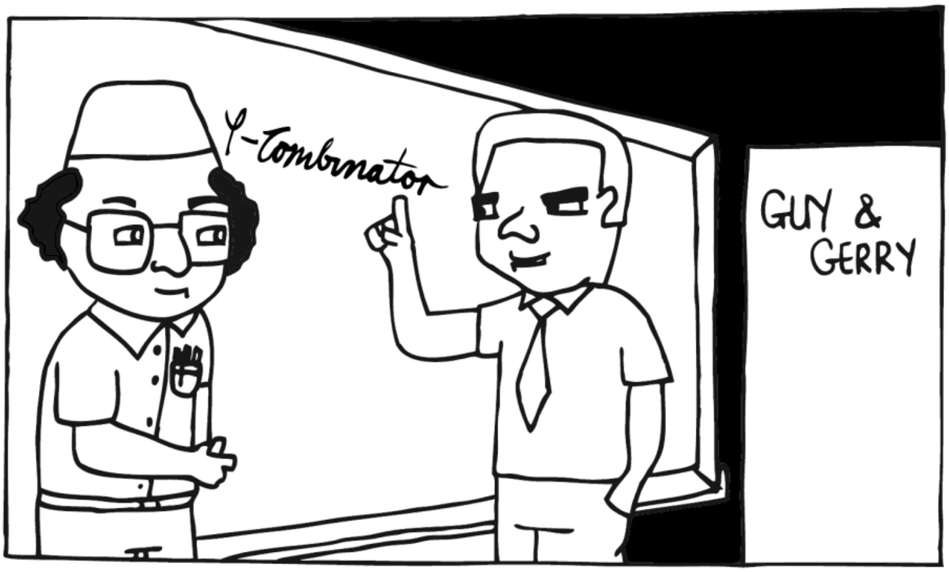
(.4 Where Does Racket Come From?

Now back to 1972. Object-oriented programming had made an appearance. What we call the Smalltalk and Simula programming languages were up and running, and they suggested the object-oriented way of programming. People at John's old place began to study these ideas. Guy Steele and Gerry Sussman were two of these programmers. Here is how it all went down when they took this topic into their own hands:

> *What's an object?*
>
> An object computes something if you send it different kinds of messages.
>
> *Do we really need to understand how objects work with many different messages?*
>
> No way. A scientist just needs to know how objects deal with one message, and that shows how all messages work.
>
> *So what are objects that understand one kind of message?*
>
> Functions! Objects that understand only one message—"run your code on the following arguments"—are functions.
>
> *How about all these loops?*
>
> Al already knew in the 1930s that loops are just abbreviations for recursive functions.
>
> *You mean loops are like frosting on the cake when an object can send a message to itself?*
>
> Absolutely, loops are like adding sugar to soda.
>
> *Are conditionals fundamental?*
>
> Nope, not even they are. Al showed that such things are just functions and compositions of functions.

Guy and Gerry went back and forth for a while in this fashion. When they were done, they had extended Al's programming language of functions with just two ideas: assignment statements and jumps in control flow. Everything else was **syntactic sugar**, that is, shorthand for a combination of concepts from this tiny core language. The language drew its power from Guy and Gerry's efforts to make everything simple and regular.

The two also understood that this collection of ideas was close to John's idea of Lisp and that Lisp was really good for prototyping new languages. And so they prototyped their language in Lisp and called it Scheme. Because of that, people started thinking of Scheme as just another form of Lisp.

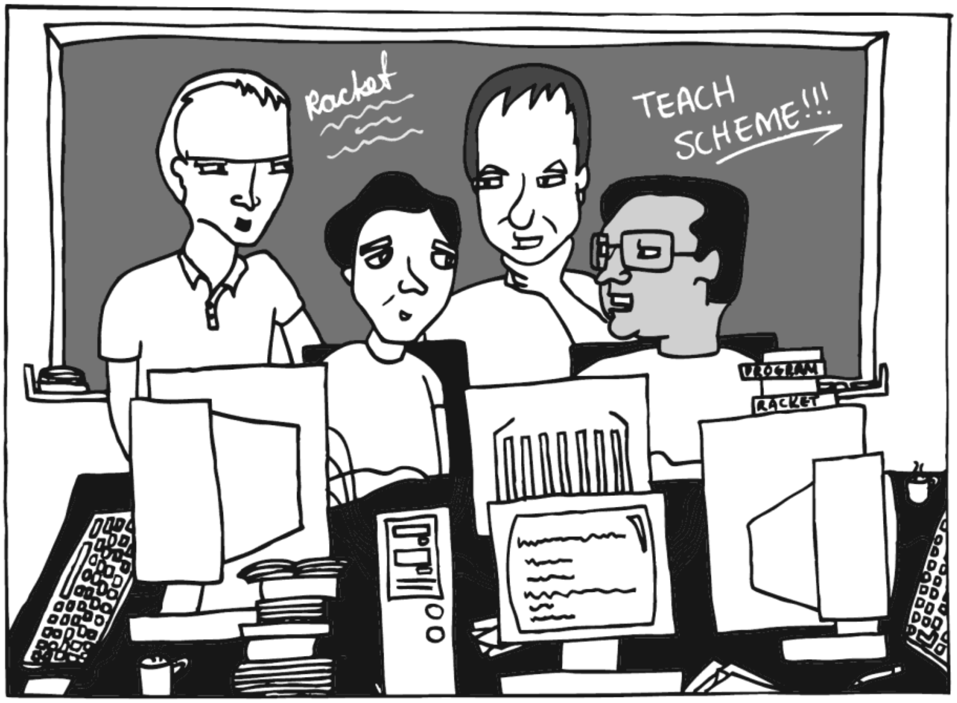


From Guy and Gerry's MIT AI Lab, Scheme quickly spread to several other places. Dan Friedman and Mitch Wand picked it up at Indiana University, and their research group built a version called Scheme84. They used Scheme to conduct programming language research. Because Scheme is so small and regular, it is easy to study the effects of adding one more idea or whether a new idea is just syntactic sugar for Scheme. A team of Yale researchers, under Paul Hudak's leadership, created another flavor, dubbed T. They looked at T as a compiler project, figuring out how to compile an expressive language into fast machine code. On top of that, individuals all over the world constructed their own variants of Scheme. Soon enough, dozens of Scheme implementations existed, all with their own small mixins and little extensions and tiny add-ons.

One of these Scheme implementations emerged at Rice University in Houston, Texas. Matthias Felleisen, Robby Findler, Matthew Flatt, and Shriram Krishnamurthi wanted to use Scheme to teach children math in a creative way. Children in middle school and high school could write computer games in plain arithmetic and algebra, which is easily expressed in a Scheme-like language. The four researchers also wanted to build all the necessary software in Scheme, and then they quickly realized that Scheme was too small to build real systems. They added structures, a class system, exceptions, fancy loops like you've never seen before, modules, custodians, eventspaces, libraries for building graphical-user interfaces, and many other things. Yes, some of these additions can be understood as syntactic sugar, but others introduced fundamentally new ideas into the world of programming languages.

Eventually, Matthias and Robby and Matthew and Shriram and everyone else who used this flavor of Scheme decided that their language was quite different from the original Scheme. After a lot of loud and wild arguing, they decided to give it a new name. This is where Racket comes from.

But don't worry—just because Racket is a large, useful language doesn't mean it is difficult to learn. Remember that Matthias, Robby, Matthew, and Shriram had middle school students in mind when they launched the Racket project. Because of this, learning Racket is like walking up a gentle slope; you should never feel like you have to climb a straight wall—unlike in your class, perhaps.



(.5 What Is This Book About?

So this book is *for* freshmen and it is written *by* freshmen, with a little help from some sophomores and professors. We assume that you have programmed somewhere, somehow—most likely in a freshman course at college. Our goal is to show you our own world of programming, one game at a time. We hope that this will open your mind about programming and that it will get you in touch with your inner Racketeer. If you like it, this book is a platform from which you can easily work your way into the rest of Racket and a whole new world of understanding programs.

`Halt`—Chapter Checkpoint

This introduction acquainted you with some historic background about programming and the Lisp family of programming languages:

- Computers are dumb pieces of hardware.
- Programmers use programming languages to turn computers into useful and entertaining gadgets.
- One of the oldest high-level languages is Lisp, which is more than 50 years old.
- There are many related flavors of Lisp, and we call Lisp a family of languages.
- To this day, Lisp offers programmers a way to experience programming as poetry.
- Racket is our chosen flavor of Lisp. Racket is relatively new, and it is especially well suited for novice programmers who want to ramp up gradually.

THIS IS CHAD.

Chad looks sad.

Maybe that's because he feels lost.

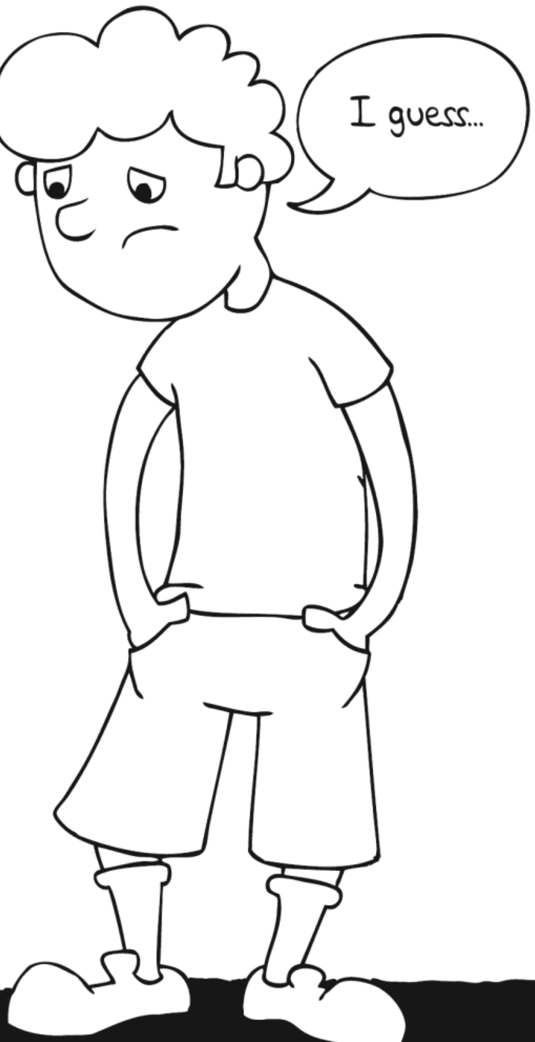
After his first year in college, he still feels unsure about his future.

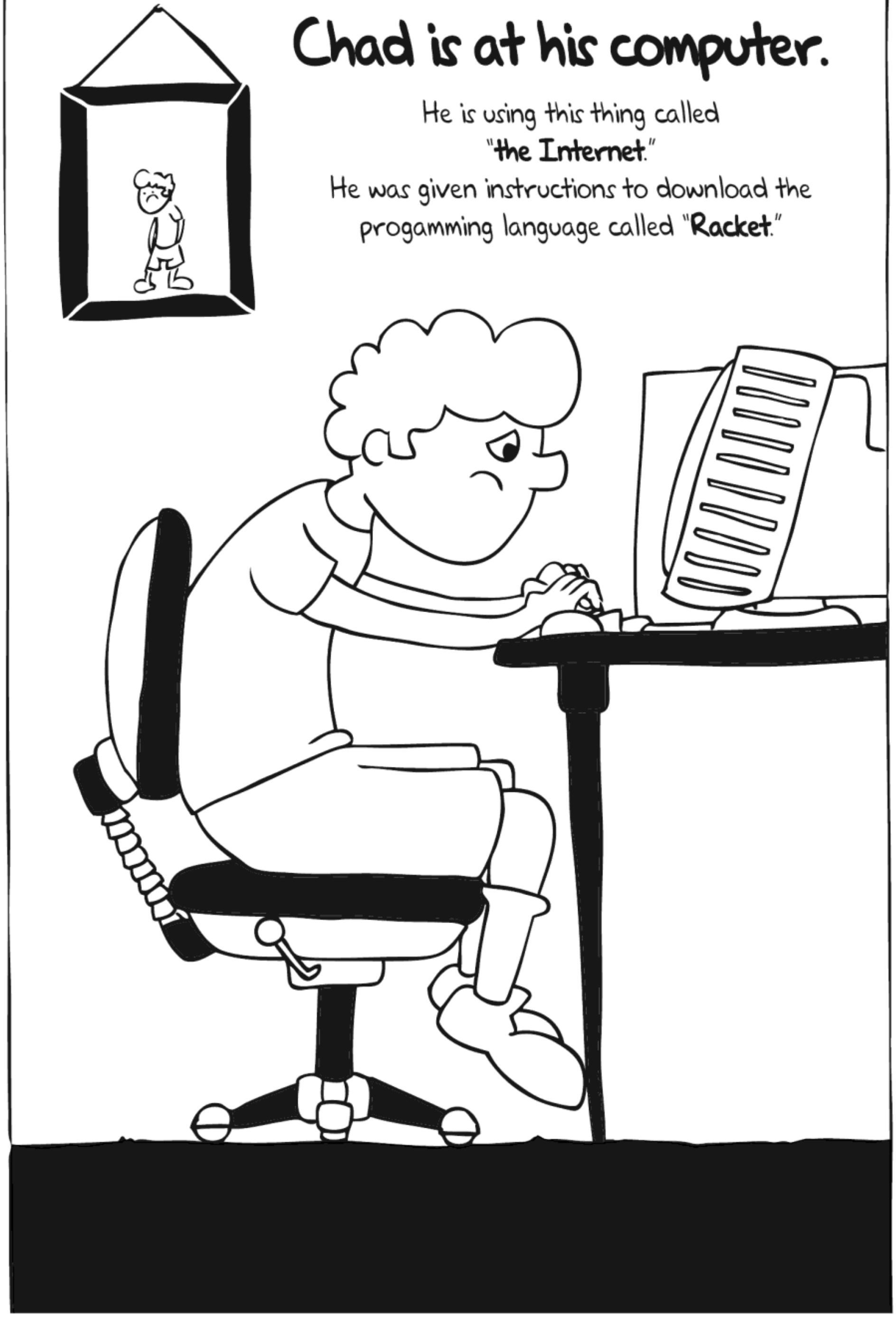
He has not declared a major yet and didn't find any of his first-year courses exciting.

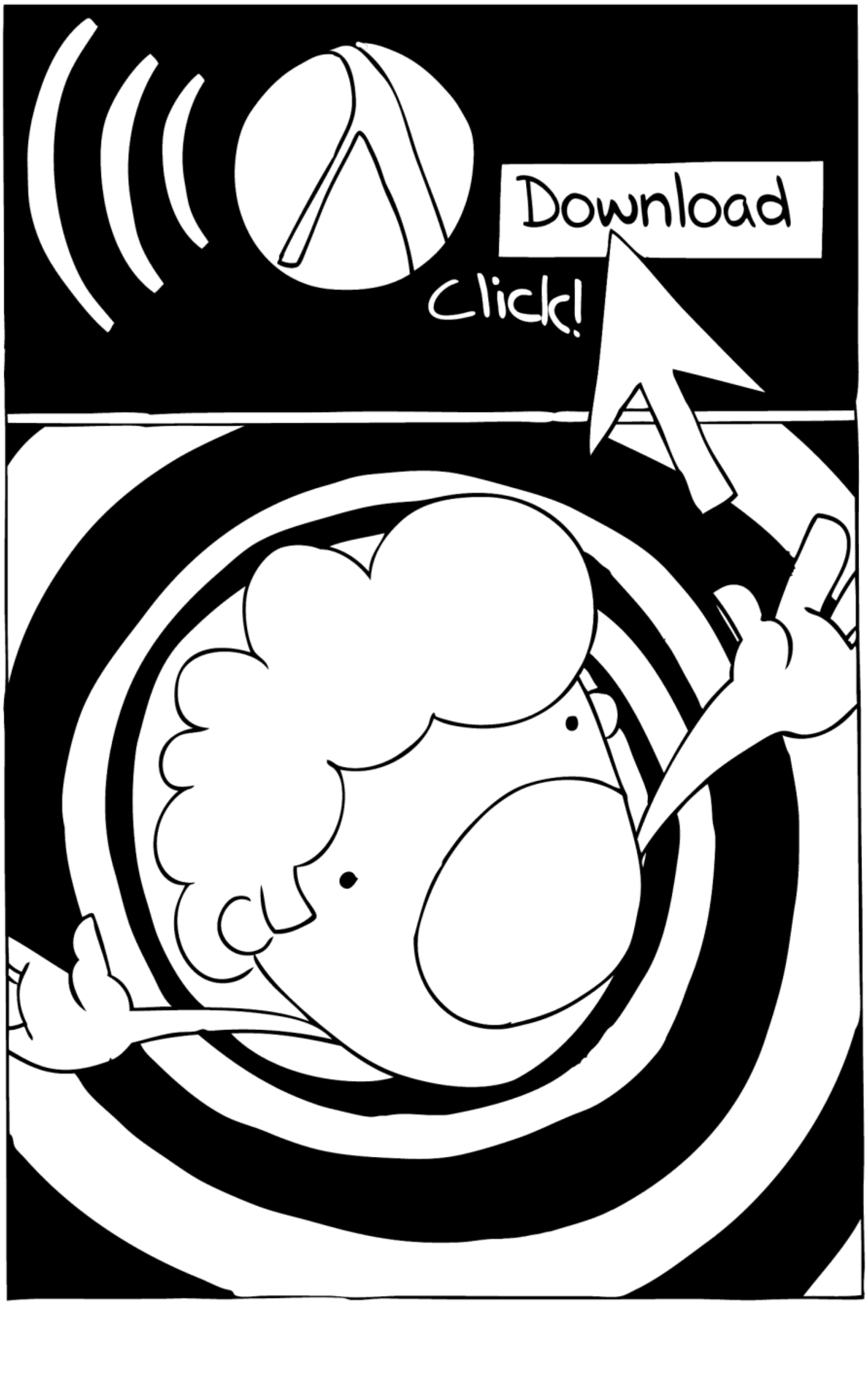
His good friends, Matt and Dave, suggested that he should check out programming, but Chad can't understand why.

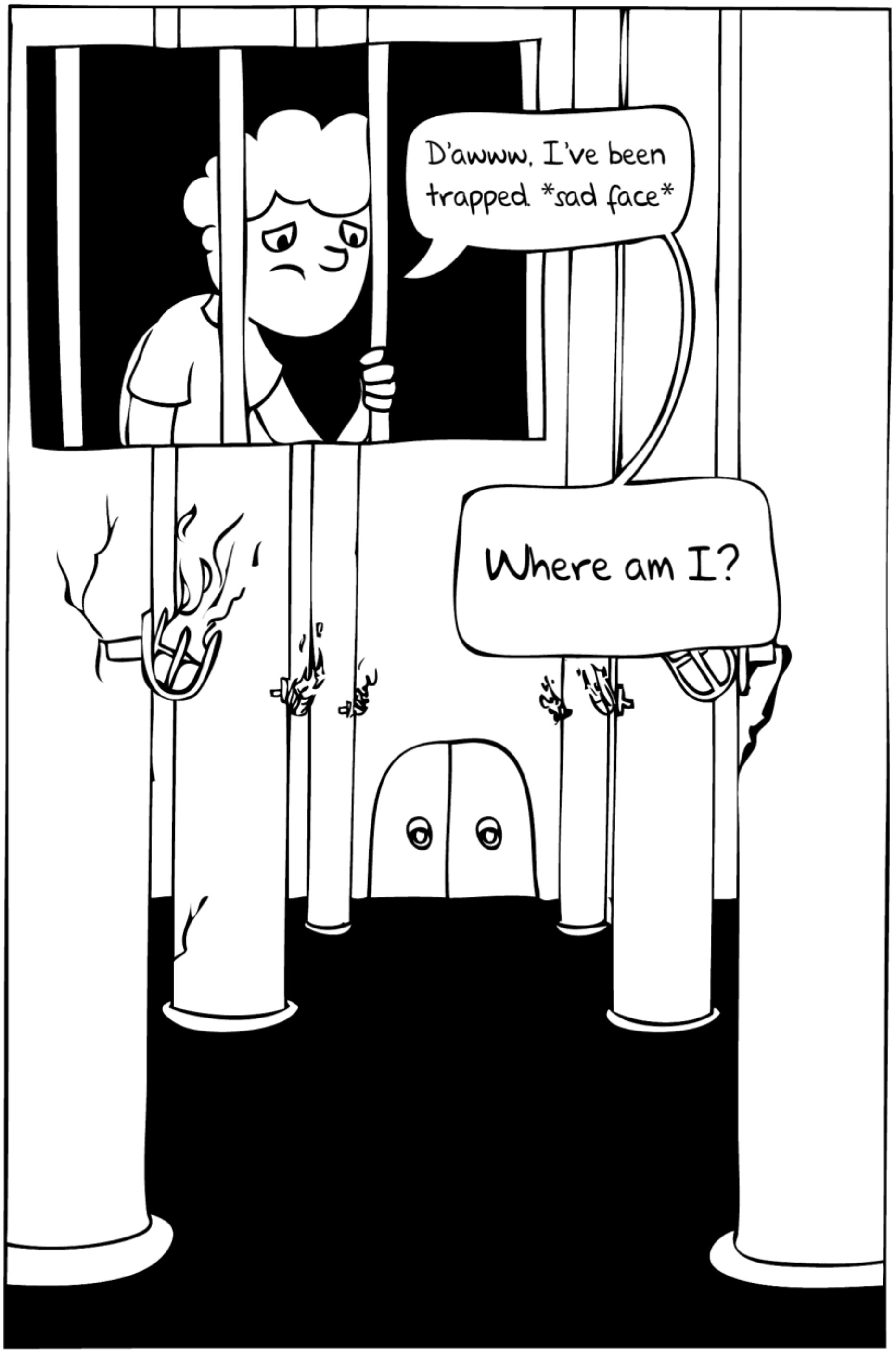
Chad is going to do some research.

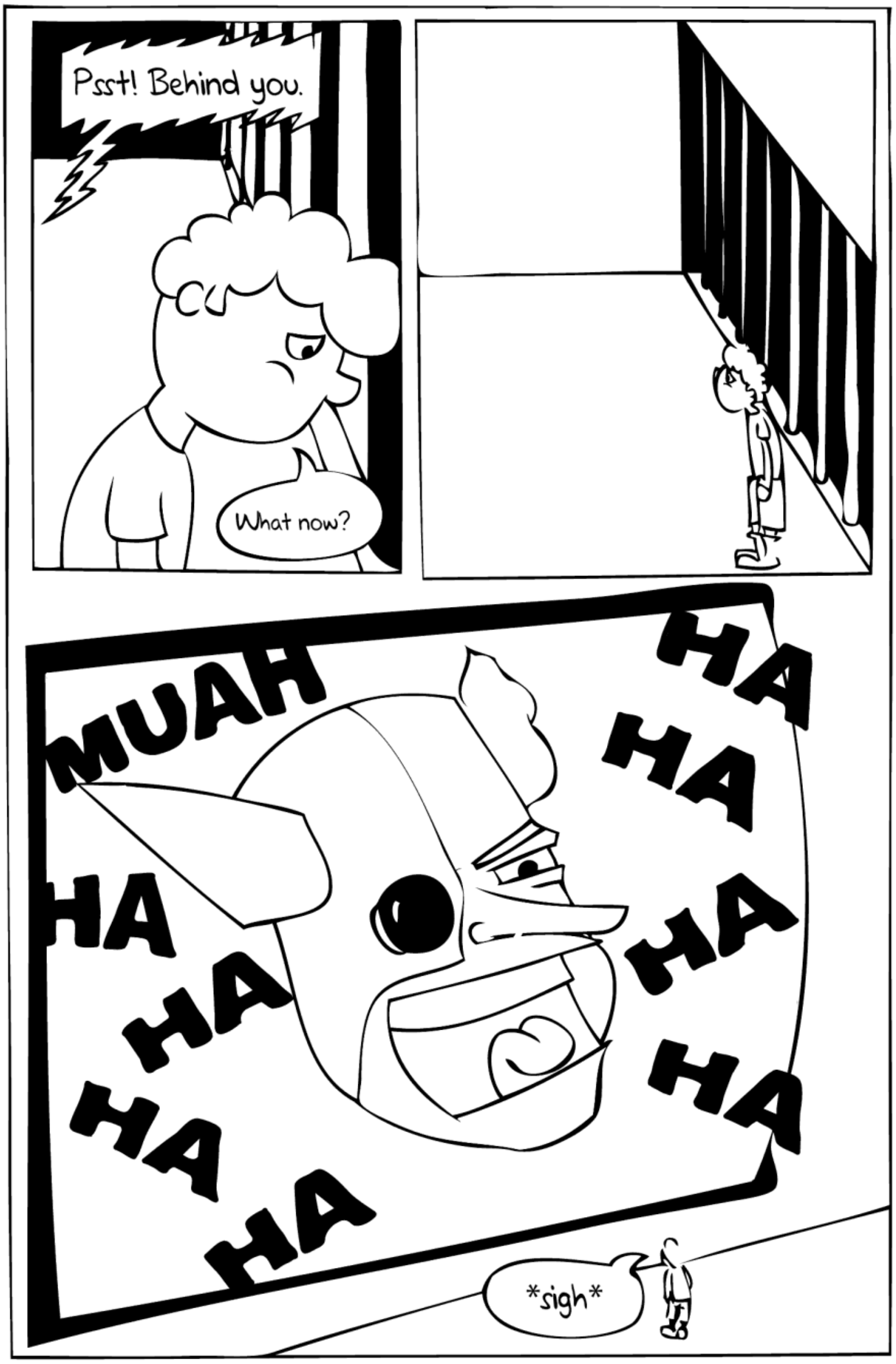
How exciting can programming really be?




Chad is at his computer.
He is using this thing called
"**the Internet**."
He was given instructions to download the
progamming language called "**Racket**."


Download
Click!


D'awww, I've been trapped *sad face*
Where am I?


Psst! Behind you.
What now?
MUAH HA HA HA HA HA HA HA HA HA
sigh

Me?

OPEN PARENTHESIS!

define me

"I am DrRacket! Now that you have entered my dungeons, you shall be my minion!

The only way you can escape is by using your mind to defeat my traps and puzzles. Your life now depends on your thinking and creativity.

Abandon all hope, ye who...who...just...uh...um...argh, nevermind.

It's going to be really difficult from now on!

MUAHAHAHAHA!"

CLOSE PARENTHESIS!

;; Chapter 1
(Getting Started)

```
#|
You need a Racket before you can launch a rocket, so the first
thing you need to learn is how to download and install DrRacket.
Once you have it on your computer, you will learn how to experi-
ment. Racket is all about experimenting with expressions and
creating your programs from these experiments. After some quick
demonstrations, you'll be ready to write a game!
|#
```

1.1 Readying Racket

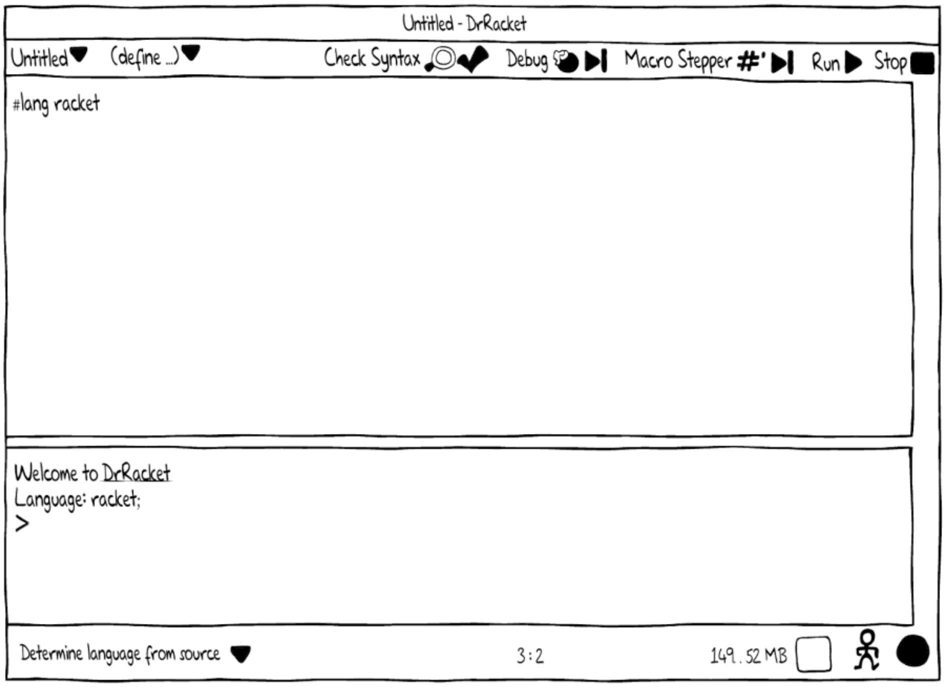
Racket is a programming language. In principle, the Racket compiler is all you need to write Racket programs. You could—and die-hard Lispers would—launch the Racket compiler in an interactive mode, type in the program, and voilà, you'd have a running program. Or you'd get an error message saying that something isn't quite right, and you'd have to retype a part of your program. After a while, this gets bothersome. Just as you need a comfortable seat to work, you need a convenient "software seat" to develop your programs, to experiment with pieces, to run your test suites, and to explore the partially finished game. We call this place a **program development environment** (**PDE**); others refer to it as an **interactive development environment** (**IDE**).

DrRacket—pronounced "Doctor Racket"—is the PDE for Racket, and it is bundled with the Racket programming language. The original Racketeers—the people who created and use DrRacket—wanted to have a PDE where everyone—young children, old Lispers, and regular programmers—could quickly feel comfortable. Therefore, DrRacket is designed in such a way that you can immediately experiment with expressions and program fragments. You can edit programs and run them. You can write tests for your program and check them. And using DrRacket is quite easy.

So let's get DrRacket. Point your browser to `racket-lang.org`, and near the top-right area of the page, you will see a download icon. Click the icon, choose your platform, and download Racket from any of the sites that show up.

Racket can run on Windows, Mac, and *nix systems. For Windows and Macs, a software installer takes care of everything that needs to happen. If you're on a *nix box, you are already a hero and don't need instructions. In the end, you will have a folder with several applications. Launch DrRacket as appropriate for your platform: by clicking, from a shell, or whatever.

#|
NOTE: In this folder, you can see another folder called **collects** and within that folder, you can find **realm.** There, you can access all the source code for all the games in this book. We encourage you to open these files in DrRacket and to experiment with the code.
|#



```
#lang racket
```

```
Welcome to DrRacket
Language: racket;
>
```

Among other things, Racket is a programming language for making programming languages. Because of that, Racket comes in many flavors. Some flavors are for beginners; they are called **teaching languages**. Others are for writing small shell scripts or large applications. A third kind is for old people who wish to program in long-gone languages. And there are many more flavors beyond this short list. As far as this book is concerned, however, we will show you just one flavor: plain Racket.

You choose which Racket flavor you want to use with these four steps:

- Select the "Language" menu of DrRacket.
- Select the "Choose Language..." menu.
- Enable the option "The Racket Language."
- Click "OK" and then click "Run," the little green go icon.

Now take a closer look at DrRacket. You should see these primary pieces:

- Some buttons, including "Run" and "Stop"
- A **definitions panel** where you see the text `#lang racket`, which means you are using the Racket programming language
- An **interactions panel** labeled "Welcome to DrRacket" with another line of text that says your chosen language is "racket" and a third line with just one symbol: ">"

1.2 Interacting with Racket

DrRacket displays the ">" prompt in the interactions panel. The prompt signals that DrRacket expects you to enter an expression, say `(+ 1 1)`. DrRacket reads this expression, evaluates it, prints the result, and then displays its prompt again. Old Lispers call this mechanism a **read-eval-print loop**, but we are sticking with the Racketeer term **interactions panel**.

Go ahead. Enter an expression. Use `(+ 1 1)` if you can't think of anything better. Here is what happens:

```
> (+ 1 1)
2
```

As soon as you hit the "Enter" key, DrRacket becomes active and prints the result. Let's try another expression:

```
> (+ 3 (* 2 4))
11
```

The interactions panel works! Hooray! You type expressions and Racket evaluates them. Experiment some more—that's what the interactions panel is for. You could, for example, play with expressions like these:

```
> (sqrt 9)
3
> (sqrt -9)
0+3i
```

Yes, Racket knows about complex numbers. How about this:

```
> (+ 1 2 3 4 5 6 7 8 9 0)
45
```

Calling + on many numbers actually works. And so do nested expressions:

```
> (sqrt (+ (sqr 3) (sqr 4)))
5
```

Lists are a pleasure, too:

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

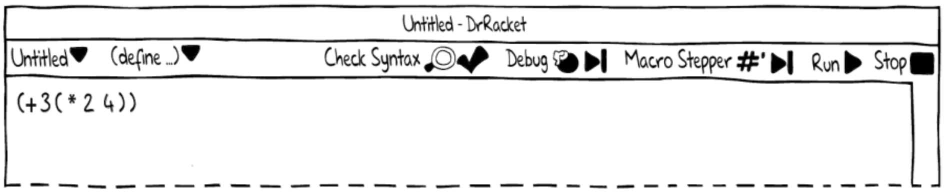
They come back out just as you put them in, because there is nothing to evaluate with lists.

How about expressions as nested lists?

```
> '(sqrt (+ (sqr 3) (sqr 4)))
'(sqrt (+ (sqr 3) (sqr 4)))
```

Okay, enough of that. We need to move on to the definitions panel.

Take a look at the sketch below. We entered `(+ 3 (* 2 4))` into the definitions panel, hit "Enter," and nothing happened. While the interactions panel is for experimenting with expressions, the definitions panel is where you record your code.



```
(+3(* 2 4))
```

To make something happen, click the "Run" button. DrRacket will evaluate the expression and then print the result `11` in the interactions panel. You could also enter this in the definitions panel:

```
'(hello world)
```

Click "Run," and you'll see `'(hello world)` in the interactions panel. DrRacket does all the printing for you; there's no need for you to specify such mundane things. Also, note that your cursor is back in the interactions panel. You can now enter expressions there again and experiment some more.

Before we forget, click the "Save" button—the one with the disk icon. DrRacket will ask you to select a file name and folder on your computer where it will save the current contents of the definitions panel. Racketeers use file names that end in **.rkt** and you should do so, too:

my-first-program.rkt

You could run the program in this file in stand-alone mode now, or you could open the file in DrRacket and run things there.

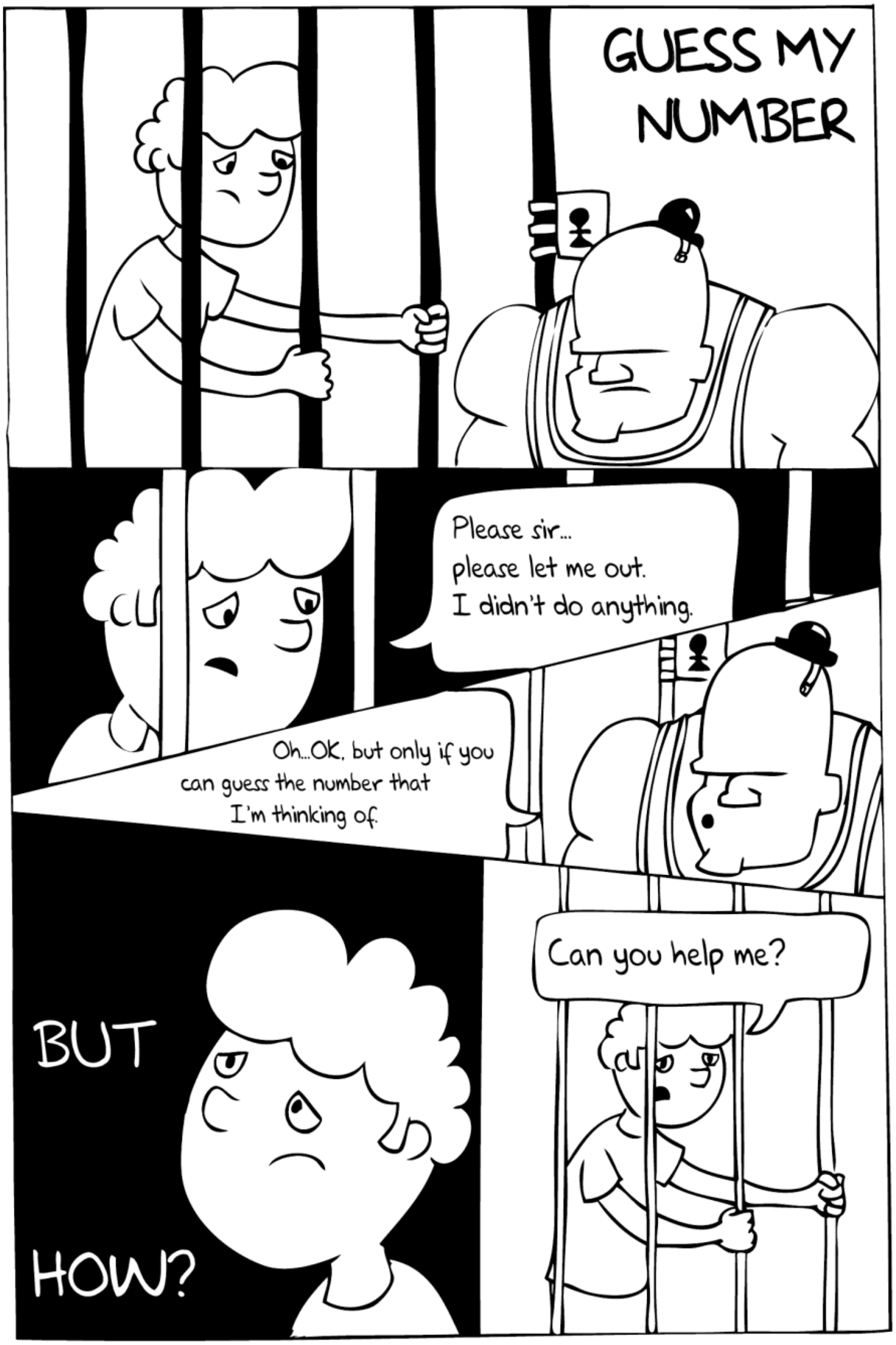
Say you closed DrRacket and, before you know it, panic strikes. You just discovered that your first program wasn't supposed to add `3` to the product of `2` and `4`. It was supposed to say `'(hello world)` and nothing else. Launch DrRacket again. Use the "File" menu and choose "Open Recent" from the available options. When you mouse-over this option, you can see **my-first-program.rkt**, and you should click it. The program is back in the definitions panel, and you are now free to change it to `'(hello world)`. And that's why we place everything we wish to keep in the definitions panel.

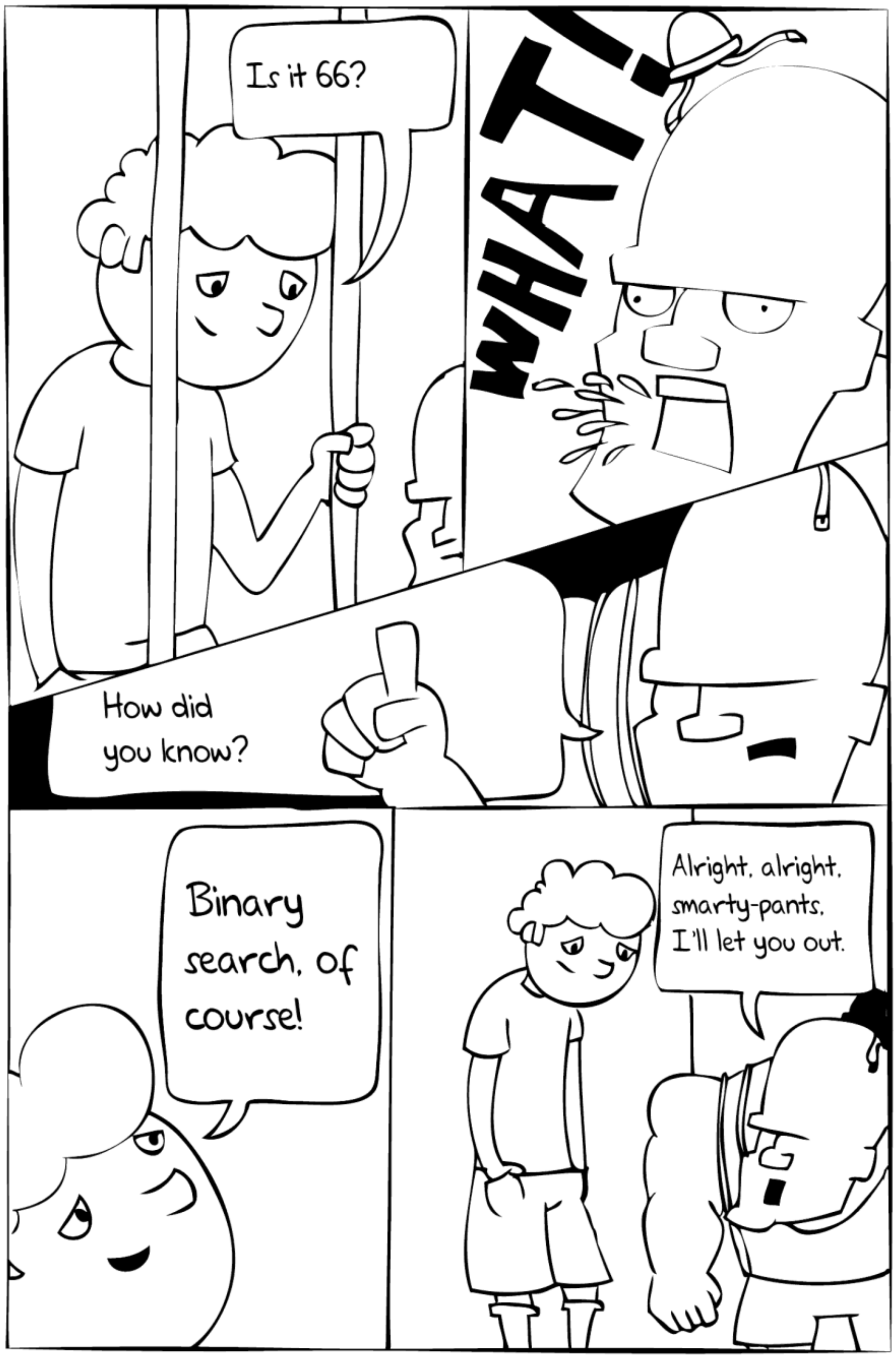
Any questions? Oh yes, it's called the definitions panel because you actually write down definitions for variables and functions. But the best way to show you how to do this is by writing our first simple game together. Ready?

`Raise`—Chapter Checkpoint

In this chapter, you learned a few basics about Racket and DrRacket:

- DrRacket is the PDE for Racket. It runs on most computing platforms. Both Racket and DrRacket are available for free online at `racket-lang.org` as one package.
- You can enter expressions and evaluate them in the interactions panel.
- You can edit programs in the definitions panel and run them, and the interactions panel shows what they compute.


GUESS MY NUMBER
Please sir... please let me out. I didn't do anything.
Oh...OK, but only if you can guess the number that I'm thinking of.
BUT HOW?
Can you help me?


Is it 66?
WHAT!
How did you know?
Binary search, of course!
Alright, alright, smarty-pants. I'll let you out.

;; Chapter 2
(A First Racket Program)

```
#|
You've installed DrRacket, learned that programs go into the
definitions panel, and experimented with Racket in the interac-
tions panel. You are now ready to write your first real program,
a simple game for guessing numbers.
|#
```

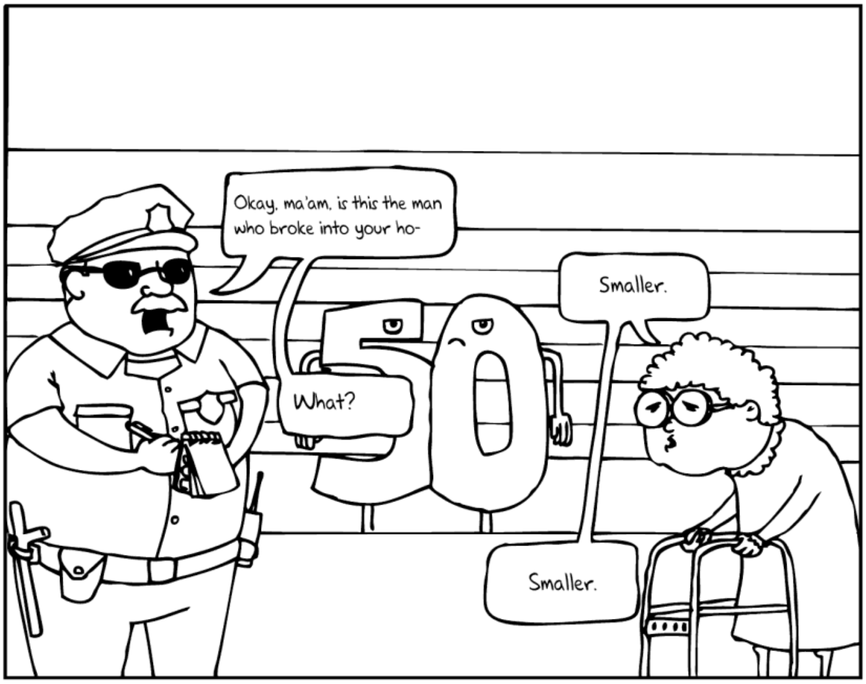
2.1 The Guess My Number Game

The first game we will write is one of the simplest and oldest games around. It's the classic Guess My Number game. In this game, the player thinks of a number between `1` and `100`. Our program will then figure out that number by repeatedly making guesses and asking the player if her number is bigger or smaller than the current guess.

The following piece shows what a game might look like in the interactions panel if the player chose `18`:

```
> (guess)
50
> (smaller)
25
> (smaller)
12
> (bigger)
18
```

The above interactions involve three different kinds of expressions: `(guess)`, `(smaller)`, and `(bigger)`. You can probably imagine what they mean. The first one tells the program to start guessing; the second one says that the guess is too high, so a smaller number needs to be guessed; and the third one commands the program to look for a bigger number.



Everything to the immediate right of "(" is a function, which means that we are dealing with three functions here: `guess`, `smaller`, and `bigger`. All you need to do is define these functions, and you have programmed yourself a first game. The player calls these functions in the interactions panel, starting with the `guess` function.

Now, let's think about the strategy behind this simple game. The basic steps are as follows:

- Determine or set the upper and lower limits of the player's number.
- Guess a number halfway between those two numbers.
- If the player says the number is smaller, lower the upper limit.
- If the player says the number is bigger, raise the lower limit.

By cutting the range of possible numbers in half with every guess, the program can quickly home in on the player's number. Cutting the number of possibilities in half at each step is called a **binary search**. As you may know, binary search is frequently used in programming because it is remarkably effective at finding answers quickly. Even if we played the game with numbers between 1 and 1,000,000, a binary strategy could guess the right number in about 10 guesses.

At this point, you know almost everything there is to know about the game. We just need to introduce a few more Racket mechanics, and you will have Guess My Number going.

2.2 Defining Variables

As the player calls the functions that make up our game, the program will need to update the lower and upper limits at each call. One way to do so is to store the limits in variables and to change the values of the variables during the game. For Guess My Number, we'll need to create two new variables called `lower` and `upper`.

The way to create a new variable is with `define`:

```
> (define lower 1)
> lower
1
> (define upper 100)
> upper
100
```

We weren't completely honest with you. In addition to functions, it is also possible that "(" is followed by a **keyword**, such as `define`. Expressions that start with a keyword work in their own special way, depending on the particular keyword. You'll just need to remember the rules for each keyword. Fortunately, there is only a very small number of them.

The `define` keyword is quite important for understanding Racket programs, as it is used to define variables and functions. Here we are using it to define variables. The first part of the `define` expression is the name of the variable, and the second is an expression that produces the value we want the variable to have.

What may surprise you is that a definition does not evaluate to anything. Don't worry. We will explain. Oh, and do place those two definitions where they belong—in the definitions panel.

2.3 Basic Racket Etiquette

Racket ignores spaces and line breaks when it reads code. This means you could format your code in any crazy way but still get the same result:

```
> (                define
             lower 1)
> lower
1
```

Because Racket code can be formatted in such flexible ways, Racketeers have conventions for formatting programs, including when to use multiple lines and indentation. We'll follow common conventions when writing examples in this book, and you're better off mimicking them. However, we're more interested in writing games than discussing source code indentation rules, so we won't spend too much time on coding style in this book.

```
#|
NOTE: Pressing the "Tab" key in DrRacket automatically indents your
code to follow common convention. You can auto-indent a chunk of code
by highlighting it and then pressing the "Tab" key. You can auto-
indent an entire Racket program just by pressing Command+i on a Mac
or Ctrl+i on Windows and *nix.
|#
```

2.4 Defining Functions in Racket

Our Guess My Number program defines `guess` to start the game and responds to requests for either `smaller` or `bigger` guesses. In addition to these three functions, we also define a function called `start` that starts the game for a different range of numbers.

Like variables, functions are defined with `define`:

```
(define (function-name argument-name ...)
  function-body-expression
  function-body-expression
  ...)
```

First, we specify the name of the function and the names of its arguments and put all of them in a pair of parentheses. Second, we follow it up with the expressions that comprise the function's logic.

The dots mean that the preceding entity occurs an arbitrary number of times: zero times, one time, two times, and so on. It is thus possible that a function may have zero arguments, but it must have at least one expression in its body.

A Function for Guessing

The first function we'll define is `guess`. This function uses the values of the `lower` and `upper` variables to generate a guess of the player's number. In our definitions panel, its definition looks like this:

```
(define (guess)
  (quotient (+ lower upper) 2))
```

To indicate that the function does not take any arguments, we place a closing parenthesis directly after the function name `guess`.

```
#|
NOTE: Although you don't need to worry about indentation or line
breaks when entering code snippets, you must be sure to place paren-
theses correctly. If you forget a parenthesis or put one in the wrong
place, you'll most likely get an error; if you don't, you're in trou-
ble. But there's no need to worry: as you have probably noticed by
now, DrRacket helps you with this task.
|#
```

Guess what this function does. As discussed earlier, the computer's best guess in this game is a number halfway between the two limits. To accomplish this, we choose the average of the two limits. If the average number ends up being a fraction, we choose the nearest whole number.

We implement this in the `guess` function by first adding the numbers that represent the upper and lower limits. The expression `(+ lower upper)` adds together the value of those two variables. We then use the `quotient` function to divide the sum in half to get an integer.

Let's see what happens when we call our new function after clicking "Run":

```
> (guess)
50
```

Since this is the program's first guess, the output we see when calling this function tells us that everything is working as planned. The program picked the number 50, right in between 1 and 100.

When programming in Racket, you'll almost always write functions that won't explicitly print values on the screen. Instead, they'll simply return the value calculated in the body of the function, and DrRacket prints it for you. For instance, let's say we wanted a function that just returns the number 5. Here's how we could write the `return-five` function:

```
(define (return-five)
  5)
```

Because the value calculated in the body of the function evaluates to `5`, calling `(return-five)` in the interactions panel returns `5`, and DrRacket prints that:

```
> (return-five)
5
```

This is how `guess` is designed; we see this calculated result on the screen not because the function displayed the number, but because this is a feature of DrRacket's interactions panel.

```
#|
NOTE: If you've used other programming languages before, you may
remember needing to write something like "return" to cause a value
to be returned. This is not necessary in Racket. The final value
calculated in the body of the function is returned automatically.
|#
```

Functions for Closing In

Now we'll write our `smaller` and `bigger` functions, which update the `upper` and `lower` variables when necessary. Like `guess`, these functions are defined with the `define` form. Let's start with `smaller`:

```
(define (smaller)
  (set! upper (max lower (sub1 (guess))))
  (guess))
```

First, we use `define` to start the definition of any new function. Because `smaller` takes no parameters, the parentheses are wrapped tightly around `smaller`. Second, the function body consists of two expressions, one per line.

Third, the function uses a `set!` expression to change the value of a variable. In general, a `set!` expression has the following shape:

```
(set! variable expression)
```

The purpose of `set!`, pronounced "set bang," is to evaluate the *`expression`* and set the *`variable`* to the resulting value. With this in mind, we can see that the `set!` expression in the definition of `smaller` first computes the new maximum number, and then it assigns that number to `upper`, giving us the new upper bound.

Since we know the maximum number must be smaller than the last guess, the biggest it can be is one less than that guess. The code `(sub1 (guess))` calculates this value. It calls our `guess` function to get the most recent guess, and then it uses the function `sub1` to subtract `1` from the result. By taking the `max` of `lower` and `(sub1 (guess))`, we ensure that `bigger` is never smaller than `lower`.

Finally, we want our `smaller` function to show the player a new guess. We do so by putting a call to `guess` as the final expression in the function body. This time, `guess` calculates the new guess using the updated value of `upper`.

The `bigger` function works in the same manner as `smaller`, except that it raises the `lower` value instead:

```
(define (bigger)
  (set! lower (min upper (add1 (guess))))
  (guess))
```

After all, if the player calls the `bigger` function, she is saying that her number is bigger than the previous guess, so the smallest it can now be is one more than the previous guess. The function `add1` simply adds `1` to the value returned by `guess`.

Here we see our functions in action, with the number 56 as the player's number:

```
> (guess)
50
> (bigger)
75
> (smaller)
62
> (smaller)
56
```

The Main Function

It is practical to have one **main function** that starts—or restarts—the whole application. Placing the definition of the main function at the top of the definitions panel also helps readers understand the purpose of the program.

For Guess My Number, this is a simple feat:

```
(define (start n m)
  (set! lower (min n m))
  (set! upper (max n m))
  (guess))
```

As you understand by now, `start` takes two arguments, which are the numbers we want to set as the `lower` and `upper` bounds. By using the `max` and `min` functions, we cut down on the instructions that we need to give any player. It suffices for a player to put in any two numbers in any order, and the function can determine the `lower` and `upper` bounds. For example, you could start the game using a small range of numbers, like this:

```
> (start 1 30)
15
```

```
#|
NOTE: As we continue to more challenging games, you will see how
a main function makes our games much more user friendly.
|#
```

The other functions continue to work as advertised:

```
> (bigger)
23
> (smaller)
19
```

Now go ahead and do some guessing yourself.

`Resume`—Chapter Checkpoint

In this chapter, we discussed some basic Racket forms. Along the way, you learned how to do the following:

- Use `define` to define a variable or function.
- Use `set!` to change the value of a variable.
- Use the interactions panel for experimentation.
- Copy and paste successful experiments to the definitions panel.

;; Chapter 3
(Basics of Racket)

```
#|
You've written your first program. It consisted of a few functions
that dealt with numbers. You've seen the basics of expressions
and definitions. You know there are a lot of parentheses.
  Now it's time to bring some order to chaos. In this chapter,
we'll show you other kinds of data, as well as the general struc-
ture and meaning of Racket programs.
|#
```

3.1 Syntax and Semantics

To understand any language—be it a human language or a language for programming—requires two concepts from the field of linguistics. Computer scientists refer to them as "syntax" and "semantics." You should know that these are just fancy words for "grammar" and "meaning."

Here is a typical sentence in the English language:

```
My dog ate my homework.
```

This sentence uses correct English syntax. Syntax is the collection of rules that a phrase must follow to qualify as a valid sentence. Here are some of the rules of sentences in the English language that this text obeys: the sentence ends in a punctuation mark, contains a subject and a verb, and is made up of letters in the English alphabet.

However, there is more to a sentence than just its syntax. We also care about what the sentence actually means. For instance, here are three sentences that, roughly, have the same semantics:

My dog ate my homework.

The canine, which I possess, has consumed my school assignment.

我的狗吃了我的家庭作業。

The first two are just different ways of saying the same thing in English. The third sentence is in Chinese, but it still has the same meaning as the first two.

The same distinction between syntax and semantics exists in programming languages. For instance, here is a valid line of code written in C++:

```
((foo<bar>)*(g++)).baz(!&qux::zip->ding());
```

This line of code obeys the rules of C++ syntax. To make the point, we put in a lot of syntax that is unique to C++. If you were to place this line of code in a Python program, it would cause a **syntax error**.

Of course, if we were to put this line of code in a C++ program in the proper context, it would cause the computer to do something. The actions that a computer performs in response to a program make up its semantics. It is usually possible to express the same semantics with distinct programs written in different programming languages; that is, the programs will perform the same actions independent of the chosen language.

Most programming languages have similar semantic powers. In fact, this is something that Al and his student Alan Turing first discovered in the 1930s. On the other hand, syntax differs among languages. Racket has a very simple syntax compared to other programming languages. Having a simple syntax is a defining feature of the Lisp family of languages, including Racket.

3.2 The Building Blocks of Racket Syntax

From the crazy line of C++ code in the previous section, you can get the hint that C++ has a lot of weird syntax—for indicating namespaces, dereferencing pointers, performing casts, referencing member functions, performing Boolean operations, and so on.

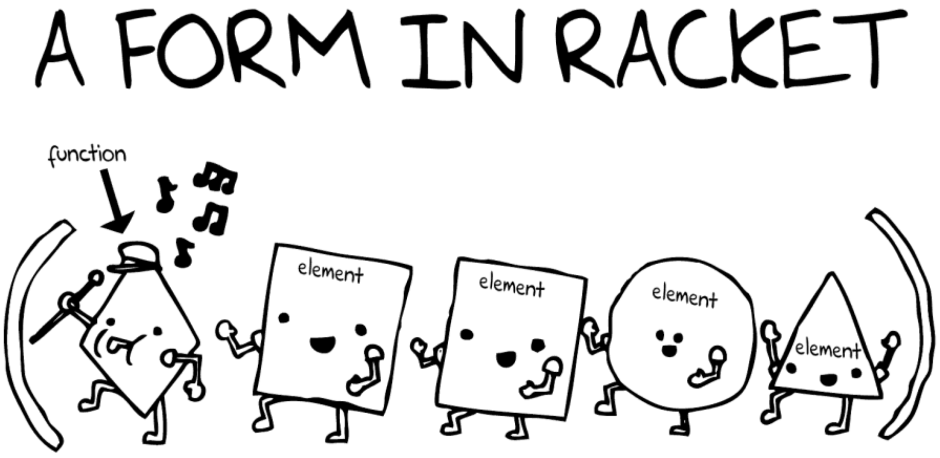
If you were to write a C++ compiler, you would need to do a lot of hard work so that the compiler could read this code and check the many C++ syntax rules.

Writing a Racket compiler or interpreter is much easier as far as syntax is concerned. The part of a Racket compiler that reads in the code, which Racketeers call the **reader**, is simpler than the equivalent part for a C++ compiler or the compiler for any other major programming language. Take a random piece of Racket code:

```
(define (square n)
  (* n n))
```

This function definition, which creates a function that squares a number, consists of nothing more than parentheses and "words." In fact, you can view it as just a bunch of nested lists.

So keep in mind that Racket has only one way of organizing bits of code: parentheses. The organization of a program is made completely clear only from the parentheses it uses. And that's all.



In addition to parentheses, Racket programmers also use square brackets `[]` and curly brackets `{}`. To keep things simple, we refer to all of these as "parentheses." As long as you match each kind of closing parenthesis to its kind of opening parenthesis, Racket will read the code. And as you may have noticed already, DrRacket is extremely helpful with matching parentheses.

The interchangeability of parentheses comes in handy for making portions of your code stand out for readers. For example, brackets are often used to group conditionals, while function applications always use parentheses. In fact, Racketeers have a number of conventions for where and when to use the various kinds of parentheses. Just read our code carefully, and you'll infer these conventions on your own. And if you prefer different conventions, go ahead, adopt them, be happy. But do stay consistent.

```
#|
NOTE: Code alone doesn't make readers happy, and therefore Racketeers
write comments. Racket has three kinds of comments. The first one is
called a line comment. Wherever Racket sees a semicolon (;), it con-
siders the rest of the line a comment, which is useful for people and
utterly meaningless for the machine. For emphasis, Racketeers use two
semicolons when they start a line comment at the beginning of a line.
The second kind of comment is a block comment. These comments are
useful for large blocks of commentary, say at the beginning of the
file. They start with #| and end with |#. While you may recognize the
first two kinds of comments from other languages, the third kind is
special to Racket and other parenthetical languages. An S-expression
comment starts with #; and it tells Racket to ignore the next paren-
thesized expression. In other words, with two keystrokes you can tem-
porarily delete or enable a large, possibly nested piece of code. Did
we mention that parentheses are great?
|#
```

3.3 The Building Blocks of Racket Semantics

Meaning matters most. In English, nouns and verbs are the basic building blocks of meaning. A noun such as "dog" evokes a certain image in our mind, and a verb such as "ate" connects our image to another in a moving sequence.

In Racket, pieces of data are the basic building blocks of meaning. We know what 5 means and we know that `'hello` is a symbol that represents a certain English word. What other sorts of data are there in Racket? There are many, including symbols, numbers, strings, and lists. Here, we'll show you the basic building blocks, or **data types**, that you'll use in Racket.

Booleans

Booleans are one of Racket's simple data forms. They represent answers to yes/no questions. So when we ask if a number is zero using the `zero?` function, we will see Boolean results:

```
> (zero? 1)
#f
> (zero? (sub1 1))
#t
```

When we ask if `1` is zero, the answer is `#f`, meaning false or no. If we subtract `1` from `1` and ask if that value is zero, we get `#t`, meaning true or yes.

Symbols

Symbols are another common type of data in Racket. A symbol in Racket is a stand-alone word preceded by a single quote or "tick" mark ('). Racket symbols are typically made up of letters, numbers, and characters such as + - / * = < > ? ! _ ^. Some examples of valid Racket symbols are 'foo, 'ice9, 'my-killer-app27, and even '--<<==>>--.

Symbols in Racket are case sensitive, but most Racketeers use uppercase sparingly. To illustrate this case sensitivity, we can use a function called symbol=? to determine if two symbols are identical:

```
> (symbol=? 'foo 'FoO)
#f
```

As you can see, the result is #f, which tells us that Racket considers these two symbols to be different.

Numbers

Racket supports both **floating-point numbers** and **integers**. As a matter of fact, it also has **rationals**, **complex numbers**, and a lot more. When you write a number, the presence of a decimal point determines whether your number is seen as a floating-point number or an integer. Thus, the exact number 1 and the floating-point number 1.0 are two different entities in Racket.

Racket can perform some amazing feats with numbers, especially when compared to most other languages. For instance, here we're using the function expt to calculate the 53rd power of 53:

```
> (expt 53 53)
2435684816502271213247760652010472551853345312868564084450513087957 67
20609150223301256150373
```

Isn't that cool? Most languages would choke on such a large integer.

You have also seen complex numbers. Consider this example:

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

Racket returns the imaginary number 0+1i for (sqrt -1), and when it multiplies this imaginary number by itself, it produces an exact -1.

Finally, something rational happens when you divide two integers:

```
> (/ 4 6)
2/3
```

The / function is dividing four by six. Mathematically speaking, this is just two over three. But chances are that if you've programmed in another language, you would expect this to produce a number like 0.66666...7. Of course, that's just an approximation of the real answer, which is the rational number two over three. Numbers in Racket behave more like numbers that you are used to from math class and less like the junk other languages try to pass off as numbers. So Racket returns a rational number, which is written as two integers with a division symbol between them. It is the mathematically ideal way to encode a fraction, and that is often what you want, too.

You will get a different answer if your calculation involves an inexact number:

```
> (/ 4.0 6)
0.6666666666666666
```

Compared with the previous example, this one uses `4.0` in place of `4`. You might think `4.0` and `4` are the same number, but in Racket, the decimal notation indicates an inexact number; `4.0` really means some number that is close to four. Consequently, when you divide a number that is close to four by six, you'll get back a number that is close to ⅔, namely `0.6666666666666666`.

Inexact numbers, like `4.0`, are called floating-point numbers, and basically they don't behave like any kind of number you've seen in a math class. But the important thing to remember is that if you never use decimal notation, you won't need to worry about how these kinds of numbers behave. Exact numbers in Racket are honest numbers like the ones you learned about in grade school.

```
#|
NOTE: People invented floating-point numbers because computer pro-
grams that use ordinary (also called "precise") numbers can sometimes
be too slow for scientists and engineers. For us, precise numbers are
usually fine, and when they are not, we'll dive into floating-point
numbers.
|#
```

Strings

Another basic building block is the **string**. Although strings aren't really that fundamental to Racket, any program that communicates with a human may need strings, because humans like to communicate with text. This book uses strings because you are probably used to them.

A string is written as a sequence of characters surrounded with double quotes. For example, `"tutti frutti"` is a string. When you ask DrRacket to evaluate a string, the result is just that string itself, as with any plain value:

```
> "tutti frutti"
"tutti frutti"
```

Like numbers, strings also come with operations. For example, you can add two strings together using the `string-append` function:

```
> (string-append "tutti" "frutti")
"tuttifrutti"
```

The `string-append` function, like the + function, is generalized to take an arbitrary number of arguments:

```
> (string-append "tutti" " " "frutti")
"tutti frutti"
```

There are other string operations like `substring`, `string-ref`, `string=?`, and more, all of which you can read about in Help Desk.

```
#|
NOTE: An easy way to look up something in Help Desk is to move your
cursor over a name and press "F1."
|#
```

3.4 Lists in Racket

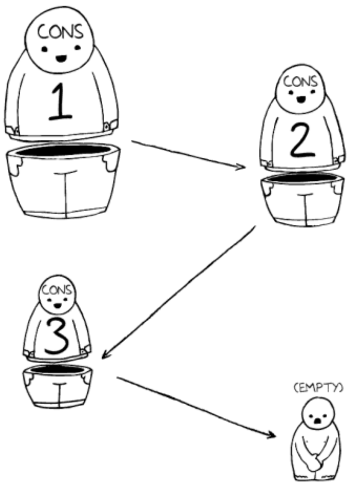
Lists are a crucial form of data in Racket. Racket data is like a big toolbox, and you can make amazing things if you know how to utilize your tools. You can't do anything without a trusty hammer, an ever-helpful screwdriver, and some needle-nose pliers. These basics are symbols, numbers, and strings in Racket. Then you have all the power tools—chain saws, drills, planers, and routers—that take everything to the next level, just like Racket lists and structures. Well, you really can't make anything in Racket without the basic cons cell, which is actually one of the most powerful tools Racket offers.

CONS Cells

Lists in Racket are held together with **cons cells**. Understanding the relationship between cons cells and lists will give you a better idea of how complex data in Racket works.

A cons cell is made of two little connected boxes, each of which can point to any other piece of data, such as a string or a number. Indeed, a cons cell can even point to another cons cell. By being able to point to different things, it's possible to link cons cells together into all kinds of data, including lists. In fact, lists in Racket are just an illusion—all of them are actually composed of cons cells.

For instance, suppose we create `(list 1 2 3)`. It's created using three cons cells. Each cell points to a number, as well as the next cons cell for the list. The final cons cell then points to `empty` to terminate the list, such as `(cons 1 (cons 2 (cons 3 empty)))`. If you've ever used a linked list in another programming language, this is the same basic idea. You can think of this arrangement as similar to a calling chain for your friends. "When I know about a party this weekend, I'll call Bob, and then Bob will call Lisa, who will call . . ." Each person in a calling chain is responsible for only one phone call, which activates the next call in the list. In the Realm of Racket, we also like to think of them as nesting dolls that shed layers until the last doll, which is rock solid.



Functions for CONS Cells

In this day and age, it is rare for a Racket programmer to manipulate cons cells as dotted pairs. Most of the time, these cells are used to build lists and nested lists, and there are great functions for dealing with all kinds of lists.

On some rare occasions, you may want to play with plain cons cells. So here is how you create a **raw cons cell**:

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

As you can see, the result is a list with a dot. You can give a cons cell a name with `define`:

```
> (define cell (cons 'a 'b))
```

And you can extract the pieces of data that you stuck into a cons cell:

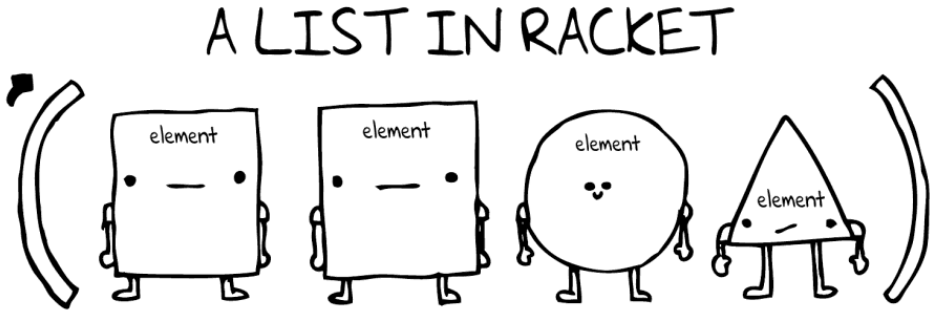
```
> (car cell)
'a
> (cdr cell)
'b
```

That is, if x is the name for a cons cell, car extracts the left piece of data from x and cdr extracts the right one.

Now you may wonder how anyone can be so crazy as to come up with names like car and cdr. We do, too. Therefore we focus on cons cells as the building blocks of lists and move on.

Lists and List Functions

Manipulating lists, not nested cons cells, is important in Racket programming. There are three basic functions for manipulating lists in Racket: cons, first, and rest. But to get started, you want to know that empty, '(), and (list) are all ways to say "empty list."



The CONS Function

If you want to link any two pieces of data in your Racket program, regardless of type, one common way to do that is with the cons function. When you call cons, Racket allocates a small chunk of memory, the cons cell, to hold references to the objects being linked. Here is a simple example where we cons the symbol chicken to the empty list:

```
> (cons 'chicken empty)
'(chicken)
```

Notice that the `empty` list is not printed in the output of our `cons` call. There's a simple reason for this: `empty` is a special value that is used to terminate a list in Racket. That said, the interactions panel is taking a shortcut and using the quote notation to describe a list with one element: `'chicken`.

The lesson here is that Racket will always go out of its way to hide the cons cells from you. The previous example can also be written like this:

```
> (cons 'chicken '())
'(chicken)
```

The empty list, `'()`, can be used interchangeably with `empty` in Racket. Thinking of `empty` as the terminator of a list makes sense. What do you get when you add a chicken to an empty list? Just a list with a chicken in it. Of course, `cons` can add items to the front of the list. For example, to add `'pork` to the front of `'(beef chicken)`, use `cons` like this:

```
> (cons 'pork '(beef chicken))
'(pork beef chicken)
```

When Racketeers talk about using `cons`, they say they are **consing** something. In this example, we consed `'pork` on to a list containing `'beef` and `'chicken`. Since all lists are made of cons cells, our `'(beef chicken)` list must have been created from its own two cons cells:

```
> (cons 'beef (cons 'chicken '()))
'(beef chicken)
```

Combining the previous two examples, we can see what all the lists look like when viewed as conses. This is what is really happening:

```
> (cons 'pork (cons 'beef (cons 'chicken '())))
'(pork beef chicken)
```

Basically, this is telling us that when we `cons` together three items, we get a list of three items.

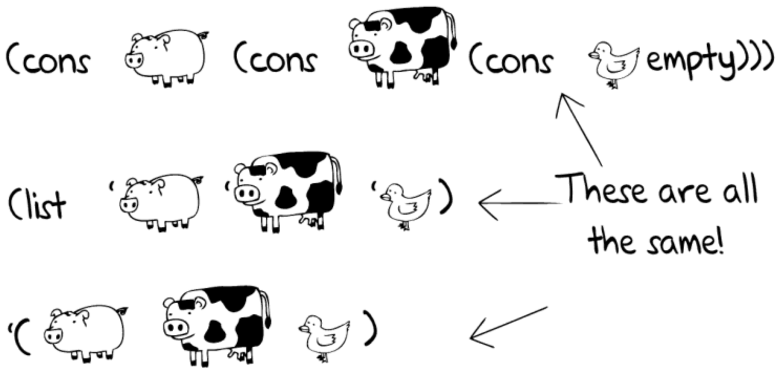
The interactions panel echoed back to us our entered items as a list, `'(pork beef chicken)`, but it could just as easily, though a little less conveniently, have reported back the items exactly as we entered them. Either response would have been perfectly correct. In Racket, a chain of cons cells and a list are exactly the same thing.

The LIST Function

For convenience, Racket has many functions built on top of the basic three—`cons`, `first`, and `rest`. A useful one is the `list` function, which does the dirty work of building our list all at once:

```
> (list 'pork 'beef 'chicken)
'(pork beef chicken)
```

Remember that there is no difference between a list created with the `list` function, one created by specifying individual cons cells, and one created with `'`. They're all the same animal. But consider the following before you rush out and buy all available quotes.



The FIRST and REST Functions

While `cons` constructs new cons cells and assembles them into lists, there are also operations for disassembling lists. The `first` function is used for getting the first element out of a list:

```
> (first (cons 'pork (cons 'beef (cons 'chicken empty))))
'pork
```

The `rest` function is used to grab the list out of the second part of the cell:

```
> (rest (list 'pork 'beef 'chicken))
'(beef chicken)
```

You can also nest `first` and `rest` to specify further which piece of data you are accessing:

```
> (first (rest '(pork beef chicken)))
'beef
```

You know that `rest` will take away the first item in a list. If you then take that shortened list and use `first`, you'll get the first item in the new list. Hence, using these two functions together retrieves the second item in the original list.

Nested Lists

Lists can contain any kind of data, including other lists:

```
> (list 'cat (list 'duck 'bat) 'ant)
'(cat (duck bat) ant)
```

This interaction shows a list containing three elements. The second element is `'(duck bat)`, which is also a list.

However, under the hood, these nested lists are still just made out of cons cells. Let's look at an example where we pull items out of nested lists.

```
> (first '((peas carrots tomatoes) (pork beef chicken)))
'(peas carrots tomatoes)
> (rest '(peas carrots tomatoes))
'(carrots tomatoes)
> (rest (first '((peas carrots tomatoes) (pork beef chicken))))
'(carrots tomatoes)
```

The `first` function gives us the first item in the list, which is a list in this case. Next, we use the `rest` function to chop off the first item from this inner list, leaving us with `'(carrots tomatoes)`. Using these functions together gives the same result.

As demonstrated in this example, cons cells allow us to create complex structures, and we use them here to build a nested list. To prove that our nested list consists solely of cons cells, here is how we could create the same nested list using only the `cons` function:

```
> (cons (cons 'peas (cons 'carrots (cons 'tomatoes '())))
        (cons (cons 'pork (cons 'beef (cons 'chicken '()))) '()))
'((peas carrots tomatoes) (pork beef chicken))
```

Since various combinations of `first` and `rest` are so common and useful, many are given their own name:

```
> (second '((peas carrots tomatoes) (pork beef chicken) duck))
'(pork beef chicken)
> (third '((peas carrots tomatoes) (pork beef chicken) duck))
'duck
> (first (second '((peas carrots tomatoes)
                   (pork beef chicken)
                   duck)))
'pork
```

In fact, functions for accessing the `first` through `tenth` elements are built in. These functions make it easy to manipulate lists in Racket, no matter how complicated they might be. If you are ever curious about built-in list functions, look in Help Desk.

3.5 Structures in Racket

Structures, like lists, are yet another means of packaging multiple pieces of data together in Racket. While lists are good for grouping an arbitrary number of items, structures are good for combining a fixed number of items. Say, for example, we need to track the name, student ID number, and dorm room number of every student on campus. In this case, we should use a structure to represent a student's information because each student has a fixed number of attributes: name, ID, and dorm. However, we would want to use a list to represent all of the students, since the campus has an arbitrary number of students, which may grow and shrink.

Structure Basics

Defining structures in Racket is simple and straightforward. If we wish to make the student structure for our example, we write the following **structure definition**:

```
> (struct student (name id# dorm))
```

This definition doesn't actually create any particular student, but instead it creates a new kind of data, which is distinct from all other kinds of data. When we say "creates a new kind of data," we really mean the structure definition provides functions for constructing and taking apart student structure values. Within `struct`, the first word—in this case, `student`—denotes the name of the structure and is also used as the name of the constructor for `student` values. The parentheses following the name of the structure enclose a series of names for the components of the structure, and the constructor takes that many values.

Let's create an **instance** of `student`:

```
> (define freshman1 (student 'Joe 1234 'NewHall))
```

Since the structure definition mentions three pieces, we apply the student constructor to three values, thus creating a single value that contains them all. This value is an instance, and it has three **fields**. Just as with any other kind of value, we can give names to structures for easy reference. If we ever need to retrieve information about our `freshman1` student, we just use the **accessors** for student structures:

```
> (student-name freshman1)
'Joe
> (student-id# freshman1)
1234
```

To access the information in a structure field, we call the appropriate accessor function. In this case, we want to pull the `name` from the student structure that we already created, so we'll use `student-name`. As you can see, the interactions panel shows `'Joe`, which is the value in `freshman1`'s `name` field. When you want to access a different field, you use the function for that field instead, say `student-id#`. As you may have guessed by now, the structure definition creates three such functions: `student-name`, `student-id#`, and `student-dorm`. We sometimes call them **field selectors** or just **selectors**.

In Racket, it is common practice to store data as lists of structures. Say we wanted to keep a list of all the freshman students in a computer science class. Since we could have anywhere from a handful to hundreds or thousands of students in the class, we want to use a list to represent it:

```
> (define mimi (student 'Mimi 1234 'NewHall))
> (define nicole (student 'Nicole 5678 'NewHall))
> (define rose (student 'Rose 8765 'NewHall))
> (define eric (student 'Eric 4321 'NewHall))
> (define in-class (list mimi nicole rose eric))
> (student-id# (third in-class))
8765
```

Here, four students are listed and combined in a list called `in-class`. All of the list functions we discussed in the previous section still apply, and we see that we can still access the fields of the student structures.

Nesting Structures

It can come in handy to have structures within structures and even within lists. For instance, in our previous example, we could keep all the students in one centralized `student-body` structure for freshmen, sophomores, juniors, and seniors, where each year stands for a list of student structures:

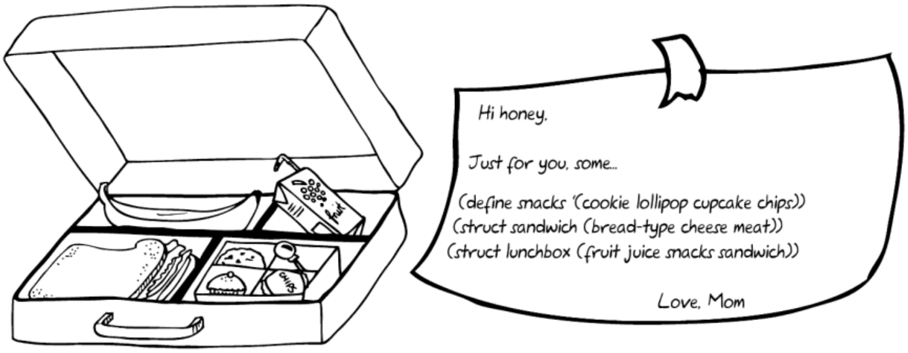
```
> (struct student-body (freshmen sophomores juniors seniors))
> (define all-students
    (student-body (list freshman1 (student 'Mary 0101 'OldHall))
                  (list (student 'Jeff 5678 'OldHall))
                  (list (student 'Bob 4321 'Apartment))
                  empty))
```

Here, we create the `student-body` structure with fields for the four different years. Next, we give the name `all-students` to one specific instance of `student-body`. As you can see, we expect lists of students in each of the four fields of a `student-body` instance.

To retrieve the name of the first freshman in the list, we need to properly layer our accessors:

```
> (student-name (first (student-body-freshmen all-students)))
'Joe
> (student-name (second (student-body-freshmen all-students)))
'Mary
> (student-name (first (student-body-juniors all-students)))
'Bob
```

We want the `student-name` of the `first` of the `student-body-freshmen` list. As we can see, `'Joe` is the name of the first freshman we created before. This also works to get `'Mary` and `'Bob`.



Hi honey,

Just for you, some...

```
(define snacks '(cookie lollipop cupcake chips))
(struct sandwich (bread-type cheese meat))
(struct lunchbox (fruit juice snacks sandwich))
```

Love, Mom

Structures and lists are useful for many different cases, such as organizing data into meaningful compartments that can be easily accessed. In this book, we will be using structures and lists for almost all programs.

Structure Transparency

By default, Racket creates opaque structures. Among other things, this means that when you create a specific structure and use it in the interactions panel, it does not print just as you created it. Rather, you will see something strange:

```
> (struct example (x y z))
> (define ex1 (example 1 2 3))
> ex1
#<example>
```

If you want to look inside structures, you must use the `#:transparent` option with your structure definition. By some sort of magical process, you will then be able to see your structures in the interactions panel:

```
> (struct example2 (p q r) #:transparent)
> (define ex2 (example2 9 8 7))
> ex2
(example2 9 8 7)
```

All of the structures in this book are supposed to be transparent, and therefore we don't bother to show the option when we define structures.

`Interrupt`—Chapter Checkpoint

In this chapter, you saw most of the building blocks of Racket's syntax and semantics:

- There are many kinds of basic data, like Booleans, symbols, numbers, and strings.
- You can make lists of data.
- You can make your own, new kinds of data with structures.
- You can mix it all up.

;; Chapter 4 (Conditions and Decisions)

```
#|
You've now seen Racket's simple syntax and a bunch of different
kinds of data. But can you write a program that answers a ques-
tion about some piece of data? And can you write a program that
chooses different values depending on the answers to these ques-
tions? In this chapter, we'll look at predicates and different
forms of conditional evaluation. Among them is an extremely ele-
gant multi-branch conditional that Racketeers use as their major
workhorse for many of their programming tasks.
|#
```

4.1 How to Ask

Racketeers think of a conditional as a form that asks questions about values and, depending on the answers, evaluates the appropriate expression. It is therefore natural that this chapter shows you how to ask questions before it introduces conditionals.

A Racket program can ask many kinds of questions, but it always asks questions with **predicates**. You may have heard of predicates in English class, but a Racket predicate is just a function that returns either true or false, which are conventionally written as #t