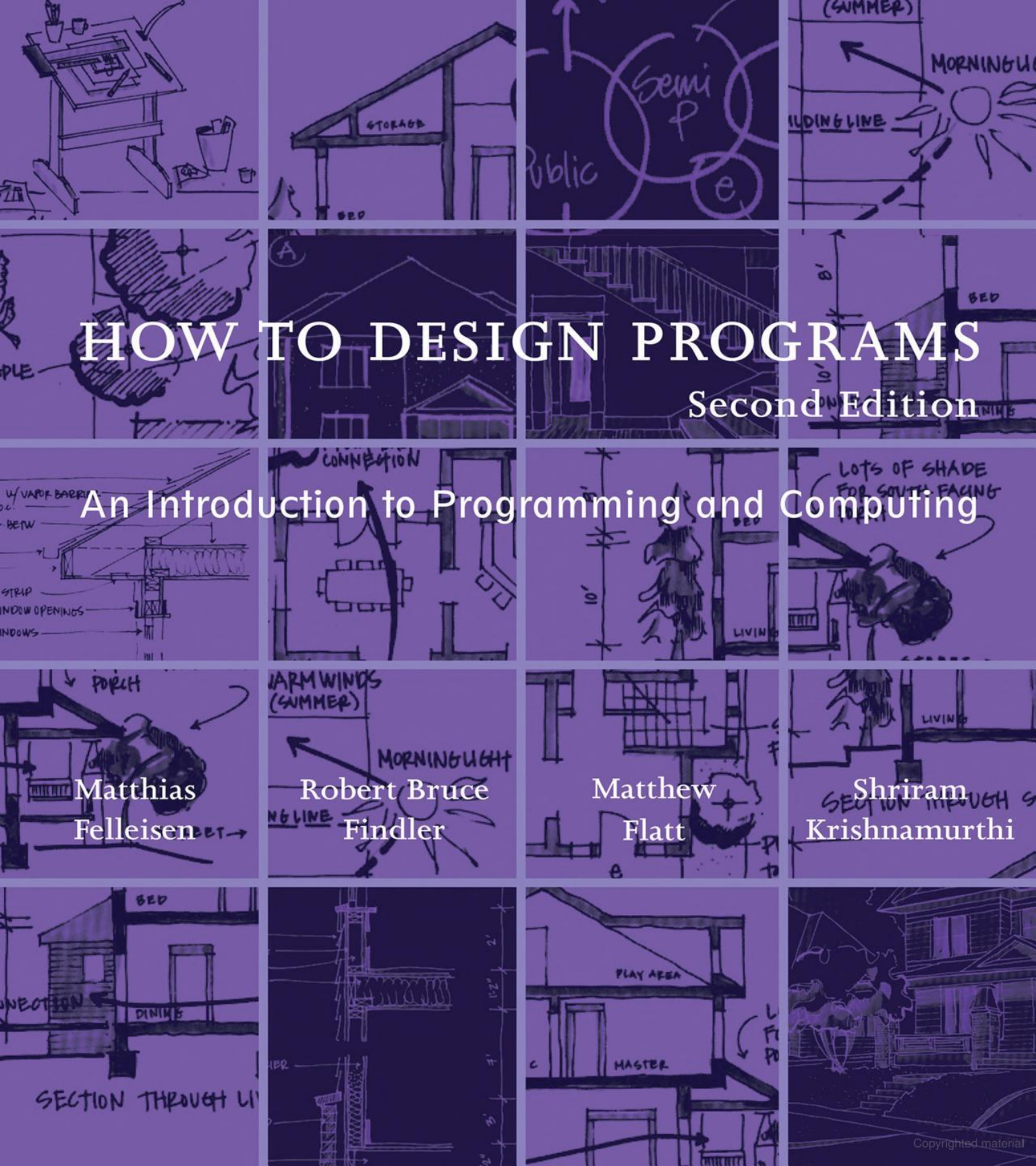

HOW TO DESIGN PROGRAMS

Second Edition

An Introduction to Programming and Computing

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi

How to Design Programs

An Introduction to Programming and Computing

Second Edition

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Scribble and LaTeX by the authors.

Library of Congress Cataloging-in-Publication Data

Names: Felleisen, Matthias.
Title: How to design programs : an introduction to programming and computing / Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi.
Description: Second edition. | Cambridge, MA : The MIT Press, [2017] | Revised edition of: How to design programs / Matthias Felleisen ... [et al.]. 2001. | Includes bibliographical references and index.

Identifiers: LCCN 2017018384 | ISBN 9780262534802 (pbk. : alk. paper)
Subjects: LCSH: Computer programming. | Electronic data processing.
Classification: LCC QA76.6 .H697 2017 | DDC 005.1/2–dc23
LC record available at https://lccn.loc.gov/2017018384

10 9 8 7 6 5 4 3 2 1

Contents

Preface

Many professions require some form of programming. Accountants program spreadsheets; musicians program synthesizers; authors program word processors; and web designers program style sheets. When we wrote these words for the first edition of the book (1995–2000), readers may have considered them futuristic; by now, programming has become a required skill and numerous outlets—books, on-line courses, K-12 curricula—cater to this need, always with the goal of enhancing people's job prospects.

The typical course on programming teaches a "tinker until it works" approach. When it works, students exclaim "It works!" and move on. Sadly, this phrase is also the shortest lie in computing, and it has cost many people many hours of their lives. In contrast, this book focuses on habits of *good programming*, addressing both professional and vocational programmers.

By "good programming," we mean an approach to the creation of software that relies on systematic thought, planning, and understanding from the very beginning, at every stage, and for every step. To emphasize the point, we speak of systematic *program design* and systematically *designed programs*. Critically, the latter articulates the rationale of the desired functionality. Good programming also satisfies an aesthetic sense of accomplishment; the elegance of a good program is comparable to time-tested poems or the black-and-white photographs of a bygone era. In short, programming differs from good programming like crayon sketches in a diner from oil paintings in a museum.

No, this book won't turn anyone into a master painter. But, we would not have spent fifteen years writing this edition if we didn't believe that

everyone can design programs

and

everyone can experience the satisfaction that comes with creative design.

Indeed, we go even further and argue that

> *program design—but* ***not programming****—deserves the same role in a liberal-arts education as mathematics and language skills.*

A student of design who never touches a program again will still pick up universally useful problem-solving skills, experience a deeply creative activity, and learn to appreciate a new form of aesthetic. The rest of this preface explains in detail what we mean with "systematic design," who benefits in what manner, and how we go about teaching it all.

Systematic Program Design

A program interacts with people, dubbed *users*, and other programs, in which case we speak of *server* and *client* components. Hence any reasonably complete program consists of many building blocks: some deal with input, some create output, while some bridge the gap between those two. We choose to use functions as fundamental building blocks because everyone encounters functions in pre-algebra and because the simplest programs are just such functions. The key is to discover which functions are needed, how to connect them, and how to build them from basic ingredients.

We drew inspiration from Michael Jackson's method for creating COBOL programs plus conversations with Daniel Friedman on recursion, Robert Harper on type theory, and Daniel Jackson on software design.

In this context, "systematic program design" refers to a mix of two concepts: design recipes and iterative refinement. The design recipes are a creation of the authors, and here they enable the use of the latter.

Design Recipes apply to both complete programs and individual functions. This book deals with just two recipes for complete programs: one for programs with a graphical user interface (GUI) and one for batch programs. In contrast, design recipes for functions come in a wide variety of flavors: for atomic forms of data such as numbers; for enumerations of different kinds of data; for data that compounds other data in a fixed manner; for finite but arbitrarily large data; and so on.

Instructors *Have students copy figure 1 on one side of an index card. When students are stuck, ask them to produce their card and point them to the step where they are stuck.*

The function-level design recipes share a common **design process**. Figure 1 displays its six essential steps. The title of each step specifies the expected outcome(s); the "commands" suggest the key activities. Examples play a central role at almost every stage. For the chosen data representation in step 1, writing down examples proves how real-world information is encoded as data and how data is interpreted as information. Step 3 says that a problem-solver must work through concrete scenarios to gain an understanding of what the desired function is expected to compute for specific examples. This understanding is exploited in step 5, when it is time to define the function. Finally, step 6 demands that examples are turned into

1. **From Problem Analysis to Data Definitions**
   Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.
2. **Signature, Purpose Statement, Header**
   State what kind of data the desired function consumes and produces. Formulate a concise answer to the question *what* the function computes. Define a stub that lives up to the signature.
3. **Functional Examples**
   Work through examples that illustrate the function's purpose.
4. **Function Template**
   Translate the data definitions into an outline of the function.
5. **Function Definition**
   Fill in the gaps in the function template. Exploit the purpose statement and the examples.
6. **Testing**
   Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

Figure 1: The basic steps of a function design recipe

automated test code, which ensures that the function works properly for some cases. Running the function on real-world data may reveal other discrepancies between expectations and results.

Each step of the design process comes with pointed questions. For certain steps—say, the creation of the functional examples or the template—the questions may appeal to the data definition. The answers almost automatically create an intermediate product. This scaffolding pays off when it comes time to take the one creative step in the process: the completion of the function definition. And even then, help is available in almost all cases.

Instructors *The most important questions are those for steps 4 and 5. Ask students to write down these questions in their own words on the back of their index card.*

The novelty of this approach is the creation of intermediate products for beginner-level programs. When a novice is stuck, an expert or an instructor can inspect the existing intermediate products. The inspection is likely to use the generic questions from the design process and thus drive the novice to correct himself or herself. And this self-empowering process is the key difference between programming and program design.

Iterative Refinement addresses the issue that problems are complex and multifaceted. Getting everything right at once is nearly impossible.

Instead, computer scientists borrow iterative refinement from the physical sciences to tackle this design problem. In essence, iterative refinement recommends stripping away all inessential details at first and finding a solution for the remaining core problem. A refinement step adds in one of these omitted details and re-solves the expanded problem, using the existing solution as much as possible. A repetition, also called an iteration, of these refinement steps eventually leads to a complete solution.

In this sense, a programmer is a miniscientist. Scientists create approximate models for some idealized version of the world to make predictions about it. As long as the model's predictions come true, everything is fine; when the predicted events differ from the actual ones, scientists revise their models to reduce the discrepancy. In a similar vein, when programmers are given a task, they create a first design, turn it into code, evaluate it with actual users, and iteratively refine the design until the program's behavior closely matches the desired product.

This book introduces iterative refinement in two different ways. Since designing via refinement becomes useful even when the design of programs becomes complex, the book introduces the technique explicitly in the fourth part, once the problems acquire a certain degree of difficulty. Furthermore, we use iterative refinement to state increasingly complex variants of the same problem over the course of the first three parts of the book. That is, we pick a core problem, deal with it in one chapter, and then pose a similar problem in a subsequent chapter—with details matching the newly introduced concepts.

DrRacket and the Teaching Languages

Learning to design programs calls for repeated hands-on practice. Just as nobody becomes a piano player without playing the piano, nobody becomes a program designer without creating actual programs and getting them to work properly. Hence, our book comes with a modicum of software support: a language in which to write down programs and a *program development environment* with which programs are edited like word documents and with which readers can run programs.

Instructors *For courses not aimed at beginners, it may be possible to use an off-the-shelf language with the design recipes.*

Many people we encounter tell us they wish they knew how to code and then ask *which programming language* they should learn. Given the press that some programming languages get, this question is not surprising. But it is also wholly inappropriate. Learning to program in a currently fashionable programming language often sets up students for eventual failure. Fashion in this world is extremely short lived. A typical "quick programming in

X" book or course fails to teach principles that transfer to the next fashion language. Worse, the language itself often distracts from the acquisition of transferable skills, at the level of both expressing solutions and dealing with programming mistakes.

In contrast, learning to design programs is primarily about the study of principles and the acquisition of transferable skills. The ideal programming language must support these two goals, but no off-the-shelf industrial language does so. The crucial problem is that beginners make mistakes *before* they know much of the language, yet programming languages always diagnose these errors as if the programmer already knew the whole language. As a result, diagnosis reports often stump beginners.

Our solution is to start with our own tailor-made teaching language, dubbed "Beginning Student Language" or BSL. The language is essentially the "foreign" language that students acquire in pre-algebra courses. It includes notation for function definitions, function applications, and conditional expressions. Also, expressions can be nested. This language is thus so small that an error diagnosis in terms of the whole language is still accessible to readers with nothing but pre-algebra under their belt.

Instructors *You may wish to explain that BSL is pre-algebra with additional forms of data and a host of pre-defined functions on those.*

A student who has mastered the structural design principles can then move on to "Intermediate Student Language" and other advanced dialects, collectively dubbed `*SL`. The book uses these dialects to teach design principles of abstraction and general recursion. We firmly believe that using such a series of teaching languages provides readers with a superior preparation for creating programs for the wide spectrum of professional programming languages (JavaScript, Python, Ruby, Java, and others).

Note The teaching languages are implemented in *Racket*, a programming language we built for building programming languages. Racket has escaped from the lab into the real world, and it is a programming vehicle of choice in a variety of settings, from gaming to the control of telescope arrays. Although the teaching languages borrow elements from the Racket language, this book does **not** teach Racket. Then again, a student who has completed this book can easily move on to Racket. **End**

When it comes to programming environments, we face an equally bad choice as the one for languages. A programming environment for professionals is analogous to the cockpit of a jumbo jet. It has numerous controls and displays, overwhelming anyone who first launches such a software application. Novice programmers need the equivalent of a two-seat, single-engine propeller aircraft with which they can practice basic skills. We have therefore created DrRacket, a programming environment for novices.

DrRacket supports highly playful, feedback-oriented learning with just

two simple interactive panes: a definitions area, which contains function definitions, and an interactions area, which allows a programmer to ask for the evaluation of expressions that may refer to the definitions. In this context, it is as easy to explore "what if" scenarios as in a spreadsheet application. Experimentation can start on first contact, using conventional calculator-style examples and quickly proceeding to calculations with images, words, and other forms of data.

An interactive program development environment such as DrRacket simplifies the learning process in two ways. First, it enables novice programmers to manipulate data directly. Because no facilities for reading input information from files or devices are needed, novices don't need to spend valuable time on figuring out how these work. Second, the arrangement strictly separates data and data manipulation from input and output of information from the "real world." Nowadays this separation is considered so fundamental to the systematic design of software that it has its own name: *model-view-controller architecture*. By working in DrRacket, new programmers are exposed to this fundamental software engineering idea in a natural way from the get-go.

Skills that Transfer

The skills acquired from learning to design programs systematically transfer in two directions. Naturally, they apply to programming in general as well as to programming spreadsheets, synthesizers, style sheets, and even word processors. Our observations suggest that the design process from figure 1 carries over to almost any programming language, and it works for 10-line programs as well as for 10,000-line programs. It takes some reflection to adopt the design process across the spectrum of languages and scale of programming problems; but once the process becomes second nature, its use pays off in many ways.

Learning to design programs also means acquiring two kinds of universally useful skills. Program design certainly teaches the same analytical skills as mathematics, especially (pre)algebra and geometry. But, unlike mathematics, working with programs is an active approach to learning. Creating software provides immediate feedback and thus leads to exploration, experimentation, and self-evaluation. The results tend to be interactive products, an approach that vastly increases the sense of accomplishment when compared to drill exercises in textbooks.

In addition to enhancing a student's mathematical skills, program design teaches analytical reading and writing skills. Even the smallest design

tasks are formulated as word problems. Without solid reading and comprehension skills, it is impossible to design programs that solve a reasonably complex problem. Conversely, program design methods force a creator to articulate his or her thoughts in proper and precise language. Indeed, if students truly absorb the design recipe, they enhance their articulation skills more than anything else.

To illustrate this point, take a second look at the process description in figure 1. It says that a designer must

1. analyze a problem statement, typically stated as a word problem;
2. extract and express its essence, abstractly;
3. illustrate the essence with examples;
4. make outlines and plans based on this analysis;
5. evaluate results with respect to expected outcomes; and
6. revise the product in light of failed checks and tests.

Each step requires analysis, precision, description, focus, and attention to details. Any experienced entrepreneur, engineer, journalist, lawyer, scientist, or any other professional can explain how many of these skills are necessary for his or her daily work. Practicing program design—on paper and in DrRacket—is a joyful way to acquire these skills.

Similarly, refining designs is not restricted to computer science and program creation. Architects, composers, writers, and other professionals do it, too. They start with ideas in their head and somehow articulate their essence. They refine these ideas on paper until their product reflects their mental image as much as possible. As they bring their ideas to paper, they employ skills analogous to fully absorbed design recipes: drawing, writing, or piano playing to express certain style elements of a building, describe a person's character, or formulate portions of a melody. What makes them productive with an iterative development process is that they have absorbed their basic design recipes and learned how to choose which one to use for the current situation.

This Book and Its Parts

The purpose of this book is to introduce readers without prior experience to the *systematic design of programs*. In tandem, it presents a *symbolic view*

of computation, a method that explains how the application of a program to data works. Roughly speaking, this method generalizes what students learn in elementary school arithmetic and middle school algebra. But have no fear. DrRacket comes with a mechanism—the algebraic stepper—that can illustrate these step-by-step calculations.

The book consists of six parts separated by five intermezzos and is bookended by a Prologue and an Epilogue. While the major parts focus on program design, the intermezzos introduce supplementary concepts concerning programming mechanics and computing.

The Prologue is a quick introduction to plain programming. It explains how to write a simple animation in `⋆SL`. Once finished, any beginner is bound to feel simultaneously empowered and overwhelmed. The final note therefore explains why plain programming is wrong and how a systematic, gradual approach to program design eliminates the sense of dread that every beginning programmer usually experiences. Now the stage is set for the core of the book:

- Part I explains the most fundamental concepts of systematic design using simple examples. The central idea is that designers typically have a rough idea of what data the program is supposed to consume and produce. A systematic approach to design must therefore extract as many hints as possible from the description of the data that flows into and out of a program. To keep things simple, this part starts with atomic data—numbers, images, and so on—and then gradually introduces new ways of describing data: intervals, enumerations, itemizations, structures, and combinations of these.

- Intermezzo 1 describes the teaching language in complete detail: its vocabulary, its grammar, and its meaning. Computer scientists refer to these as syntax and semantics. Program designers use this model of computation to predict what their creations compute when run or to analyze error diagnostics.

- Part II extends part I with the means to describe the most interesting and useful forms of data: arbitrarily large compound data. While a programmer may nest the kinds of data from part I to represent information, the nesting is always of a fixed depth and breadth. This part shows how a subtle generalization gets us from there to data of arbitrary size. The focus then switches to the systematic design of programs that process this kind of data.

- Intermezzo 2 introduces a concise and powerful notation for writing down large pieces of data: quotation and anti-quotation.

- Part III acknowledges that many of the functions from part II look alike. No programming language should force programmers to create pieces of code that are so similar to each other. Conversely, every good programming language comes with ways to eliminate such similarities. Computer scientists call both the step of eliminating similarities and its result *abstraction*, and they know that abstractions greatly increase a programmer's productivity. Hence, this part introduces design recipes for creating and using abstractions.

- Intermezzo 3 plays two roles. On the one hand, it injects the concept of *lexical scope*, the idea that a programming language ties every occurrence of a name to a definition that a programmer can find with an inspection of the code. On the other hand, it explains a library with additional mechanisms for abstraction, including so-called *for loops*.

- Part IV generalizes part II and explicitly introduces the idea of iterative refinement into the catalog of design concepts.

- Intermezzo 4 explains and illustrates why decimal numbers work in such strange ways in all programming languages. Every budding programmer ought to know these basic facts.

- Part V adds a new design principle. While structural design and abstraction suffice for most problems that programmers encounter, they occasionally lead to insufficiently "performant" programs. That is, structurally designed programs might need too much time or energy to compute the desired answers. Computer scientists therefore replace structurally designed programs with programs that benefit from ad hoc insights into the problem domain. This part of the book shows how to design a large class of just such programs.

- Intermezzo 5 uses examples from part V to illustrate how computer scientists think about performance.

- Part VI adds one final trick to the toolbox of designers: accumulators. Roughly speaking, an accumulator adds "memory" to a function. The addition of memory greatly improves the performance of structurally designed functions from the first four parts of the book. For the ad hoc programs from part V, accumulators can make the difference between finding an answer and never finding one.

The Epilogue is both an assessment and a look ahead to what's next.

Figure 2: The dependencies among parts and intermezzos

Independent readers ought to work through the entire book, from the first page to the last. We say "work" because we really mean that a reader ought to solve all exercises or at least know how to solve them.

Similarly, instructors ought to cover as many elements as possible, starting from the Prologue all the way through the Epilogue. Our teaching experience suggests that this is doable. Typically, we organize our courses so that our readers create a sizable and entertaining program over the course of the semester. We understand, however, that some circumstances call for significant cuts and that some instructors' tastes call for slightly different ways to use the book.

Figure 2 is a navigation chart for those who wish to pick and choose

from the elements of the book. The figure is a dependency graph. A solid arrow from one element to another suggests a mandatory ordering; for example, Part II requires an understanding of Part I. In contrast, a dotted arrow is mostly a suggestion; for example, understanding the Prologue is unnecessary to get through the rest of the book.

Based on this chart, here are three feasible paths through the book:

- A high school instructor may want to cover (as much as possible of) parts I and II, including a small project such as a game.
- A college instructor in a quarter system may wish to focus on part I, part II, part III, and part V, plus the intermezzos on `*SL` and scope.
- A college instructor in a semester system may prefer to discuss performance trade-offs in designs as early as possible. In this case, it is best to cover part I and part II and then the accumulator material from part VI that does not depend on part V. At that point, it is possible to discuss intermezzo 5 and to study the rest of the book from this angle.

Iteration of Sample Topics The book revisits certain exercise and sample topics time and again. For example, virtual pets are found all over part I and even show up in part II. Similarly, both part I and part II cover alternative approaches to implementing an interactive text editor. Graphs appear in part V and immediately again in part VI. The purpose of these iterations is to motivate iterative refinement and to introduce it through the backdoor. We urge instructors to assign these themed sequences of exercises or to create their own such sequences.

The Differences

This second edition of *How to Design Programs* differs from the first one in several major aspects:

1. It explicitly acknowledges the difference between designing a whole program and the functions that make up a program. Specifically, this edition focuses on two kinds of programs: event-driven (mostly GUI, but also networking) programs and batch programs.
2. The design of a program proceeds in a top-down planning phase followed by a bottom-up construction phase. We explicitly show how the interface to libraries dictates the shape of certain program elements. In particular, the very first phase of a program design yields

a wish list of functions. While the concept of a wish list exists in the first edition, this second edition treats it as an explicit design element.

3. Fulfilling an entry from the wish list relies on the function design recipe, which is the subject of the six major parts.

4. A key element of structural design is the definition of functions that compose others. This design-by-composition is especially useful for the world of batch programs. Like generative recursion, it requires a *eureka!*, specifically a recognition that the creation of intermediate data by one function and processing this intermediate result by a second function simplifies the overall design. This approach also needs a wish list, but formulating these wishes calls for an insightful development of an intermediate data definition. This edition of the book weaves in a number of explicit exercises on design by composition.

We thank Kathi Fisler for calling our attention to this point.

5. While testing has always been a part of our design philosophy, the teaching languages and DrRacket started supporting it properly only in 2002, just after we had released the first edition. This new edition heavily relies on this testing support.

6. This edition of the book drops the design of imperative programs. The old chapters remain available on-line. An adaptation of this material will appear in the second volume of this series, *How to Design Components*.

7. The book's examples and exercises employ new teachpacks. The preferred style is to link in these libraries via `require`, but it is still possible to add teachpacks via a menu in DrRacket.

8. Finally, this second edition differs from the first in a few aspects of terminology and notation:

Second Edition	First Edition
signature	contract
itemization	union
`'()`	`empty`
`#true`	`true`
`#false`	`false`

The last three differences greatly improve quotation for lists.

Acknowledgments from the First Edition
Four people deserve special thanks: Robert "Corky" Cartwright, who co-developed a predecessor of Rice University's introductory course with the first author; Daniel P. Friedman, for asking the first author to rewrite *The Little LISPer* (also MIT Press) in 1984, because it started this project; John Clements, who designed, implemented, and maintains DrRacket's stepper; and Paul Steckler, who faithfully supported the team with contributions to our suite of programming tools.

The development of the book benefited from many other friends and colleagues who used it in courses and/or gave detailed comments on early drafts. We are grateful to them for their help and patience: Ian Barland, John Clements, Bruce Duba, Mike Ernst, Kathi Fisler, Daniel P. Friedman, John Greiner, Géraldine Morin, John Stone, and Valdemar Tamez.

A dozen generations of Comp 210 students at Rice used early drafts of the text and contributed improvements in various ways. In addition, numerous attendees of our TeachScheme! workshops used early drafts in their classrooms. Many sent in comments and suggestions. As representative of these we mention the following active contributors: Ms. Barbara Adler, Dr. Stephen Bloch, Ms. Karen Buras, Mr. Jack Clay, Dr. Richard Clemens, Mr. Kyle Gillette, Mr. Marvin Hernandez, Mr. Michael Hunt, Ms. Karen North, Mr. Jamie Raymond, and Mr. Robert Reid. Christopher Felleisen patiently worked through the first few parts of the book with his father and provided direct insight into the views of a young student. Hrvoje Blazevic (sailing, at the time, as Master of the *LPG/C Harriette*), Joe Zachary (University of Utah), and Daniel P. Friedman (Indiana University) discovered numerous typos in the first printing, which we have now fixed. Thank you to everyone.

Finally, Matthias expresses his gratitude to Helga for her many years of patience and for creating a home for an absent-minded husband and father. Robby is grateful to Hsing-Huei Huang for her support and encouragement; without her, he would not have gotten anything done. Matthew thanks Wen Yuan for her constant support and enduring music. Shriram is indebted to Kathi Fisler for support, patience and puns, and for her participation in this project.

Acknowledgments
As in 2001, we are grateful to John Clements for designing, validating, implementing, and maintaining DrRacket's algebraic stepper. He has done so for nearly 20 years now, and the stepper has become an indispensable tool of explanation and instruction.

Over the past few years, several colleagues have commented on the various drafts and suggested improvements. We gratefully acknowledge the thoughtful conversations and exchanges with these individuals:

> Kathi Fisler (WPI and Brown University), Gregor Kiczales (University of British Columbia), Prabhakar Ragde (University of Waterloo), and Norman Ramsey (Tufts University).

Thousands of teachers and instructors attended our various workshops over the years, and many provided valuable feedback. But Dan Anderson, Stephen Bloch, Jack Clay, Nadeem Abdul Hamid, and Viera Proulx stand out, and we wish to call out their role in the crafting of this edition.

Guillaume Marceau, working with Kathi Fisler and Shriram, spent many months studying and improving the error messages in DrRacket. We are grateful for his amazing work.

Celeste Hollenbeck is the most amazing reader ever. She never tired of pushing back until she understood the prose. She never stopped until a section supported its thesis, its organization matched, and its sentences connected. Thank you very much for your incredible efforts.

We also thank the following: Saad Bashir, Steven Belknap, Stephen Bloch, Joseph Bogart Tomas Cabrera, Estevo Castro, Stephen Chang, Jack Clay, Richard Cleis, John Clements, Mark Engelberg, Christopher Felleisen, Sebastian Felleisen, Vladimir Gajić, Adrian German, Ryan Golbeck, Jane Griscti, Alberto E. F. Guerrero, Nadeem Abdul Hamid, Wayne Iba, Jordan Johnson, Marc Kaufmann, Gregor Kiczales, Eugene Kohlbecker, Jackson Lawler, Ben Lerner, Elena Machkasova, Jay Martin, Jay McCarthy, Ann E. Moskol, Paul Ojanen, Klaus Ostermann, Alanna Pasco, S. Pehlivanoglu, David Porter, Norman Ramsey, Ilnar Salimzianov, Brian Schack, Tubo Shi, Stephen Siegel, Kartik Singhal, Marc Smith, Dave Smylie, Vincent St-Amour, Éric Tanter, Sam Tobin-Hochstadt, Manuel del Valle, David Van Horn, Mitch Wand, Roelof Wobben, and Andrew Zipperer for comments on drafts of this second edition.

The HTML layout at `htdp.org` is the work of Matthew Butterick, who created these styles for our on-line documentation.

Finally, we are grateful to Ada Brunstein and Marie Lufkin Lee, our editors at MIT Press, who gave us permission to develop this second edition of *How to Design Programs* on the web. We also thank MIT's Christine Bridget Savage and John Hoey from Westchester Publishing Services for managing the final production process. John Donohue, Jennifer Robertson, and Mark Woodworth did a wonderful job of copy editing the manuscript.

How to Design Programs

Second Edition

Prologue: How to Program

When you were a small child, your parents taught you to count and perform simple calculations with your fingers: "1 + 1 is 2"; "1 + 2 is 3"; and so on. Then they would ask "what's 3 + 2?" and you would count off the fingers of one hand. They programmed, and you computed. And in some way, that's really all there is to programming and computing.

Download DrRacket from its web site.

Now it is time to switch roles. Start DrRacket. Doing so brings up the window of figure 3. Select "Choose language" from the "Language" menu, which opens a dialog listing "Teaching Languages" for "How to Design Programs." Choose "Beginning Student" (the Beginning Student Language, or BSL) and click *OK* to set up DrRacket. With this task completed, **you** can program, and the DrRacket software becomes the child. Start with the simplest of all calculations. You type

```
(+ 1 1)
```

into the top part of DrRacket, click *RUN*, and a `2` shows up in the bottom.

That's how simple programming is. You ask questions as if DrRacket were a child, and DrRacket computes for you. You can also ask DrRacket to process several requests at once:

```
(+ 2 2)
(* 3 3)
(- 4 2)
(/ 6 2)
```

After you click *RUN*, you see `4 9 2 3` in the bottom half of DrRacket, which are the expected results.

Let's slow down for a moment and introduce some words:

- The top half of DrRacket is called the *definitions area*. In this area, you create the programs, which is called *editing*. As soon as you add a word or change something in the definitions area, the *SAVE* button shows up in the top-left corner. When you click *SAVE* for the first

Figure 3: Meet DrRacket

time, DrRacket asks you for the name of a file so that it can store your program for good. Once your definitions area is associated with a file, clicking *SAVE* ensures that the content of the definitions area is stored safely in the file.

- *Programs* consist of *expressions*. You have seen expressions in mathematics. For now, an expression is either a plain number or something that starts with a left parenthesis "(" and ends in a matching right parenthesis ")"—which DrRacket rewards by shading the area between the pair of parentheses.

- When you click *RUN*, DrRacket evaluates the expressions in the definitions area and shows their result in the *interactions area*. Then, DrRacket, your faithful servant, awaits your commands at the *prompt* (>). The appearance of the prompt signals that DrRacket is waiting

for you to enter additional expressions, which it then evaluates like those in the definitions area:

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

Enter an expression at the prompt, hit the "return" or "enter" key on your keyboard, and watch how DrRacket responds with the result. You can do so as often as you wish:

```
> (+ 2 2)
4
> (* 3 3)
9
> (- 4 2)
2
> (/ 6 2)
3
> (sqr 3)
9
> (expt 2 3)
8
> (sin 0)
0
> (cos pi)
#i-1.0
```

Take a close look at the last number. Its "#i" prefix is short for "I don't really know the precise number so take that for now" or an *inexact number*. Unlike your calculator or other programming systems, DrRacket is honest. When it doesn't know the exact number, it warns you with this special prefix. Later, we will show you really strange facts about "computer numbers," and you will then truly appreciate that DrRacket issues such warnings.

By now you might be wondering whether DrRacket can add more than two numbers at once, and yes, it can! As a matter of fact, it can do it in two different ways:

```
> (+ 2 (+ 3 4))
9
> (+ 2 3 4)
9
```

This book does not teach you Racket, even if the editor is called DrRacket. See the Preface, especially the section on DrRacket and the Teaching Languages for details on the choice to develop our own language.

The first one is *nested arithmetic*, as you know it from school. The second one is *BSL arithmetic*; and the latter is natural, because in this notation you always use parentheses to group operations and numbers together.

In BSL, every time you want to use a "calculator operation," you write down an opening parenthesis, the operation you wish to perform, say +, the numbers on which the operation should work (separated by spaces or even line breaks), and, finally, a closing parenthesis. The items following the operation are called the *operands*. Nested arithmetic means that you can use an expression for an operand, which is why

```
> (+ 2 (+ 3 4))
9
```

is a fine program. You can do this as often as you wish:

```
> (+ 2 (+ (* 3 3) 4))
15
> (+ 2 (+ (* 3 (/ 12 4)) 4))
15
> (+ (* 5 5) (+ (* 3 (/ 12 4)) 4))
38
```

There are no limits to nesting, except for your patience.

Naturally, when DrRacket calculates for you, it uses the rules that you know and love from math. Like you, it can determine the result of an addition only when all the operands are plain numbers. If an operand is a parenthesized operator expression—something that starts with a "(" and an operation—it determines the result of that nested expression first. Unlike you, it never needs to ponder which expression to calculate first—because this first rule is the only rule there is.

The price for DrRacket's convenience is that parentheses have meaning. You must enter all these parentheses, and you may not enter too many. For example, while extra parentheses are acceptable to your math teacher, this is not the case for BSL. The expression `(+ (1) (2))` contains way too many parentheses, and DrRacket lets you know in no uncertain terms:

```
> (+ (1) (2))
function call:expected a function after the open parenthesis,
found a number
```

Once you get used to BSL programming, though, you will see that it isn't a price at all. First, you get to use operations on several operands at once, if it is natural to do so:

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

If you don't know what an operation does for several operands, enter an example into the interactions area and hit "return"; DrRacket lets you know whether and how it works. Or use HelpDesk to read the documentation. Second, when you read programs that others write, you will never have to wonder which expressions are evaluated first. The parentheses and the nesting will immediately tell you.

As you may have noticed, the names of operations in the on-line text are linked to the documentation in HelpDesk.

In this context, to program is to write down comprehensible arithmetic expressions, and to compute is to determine their value. With DrRacket, it is easy to explore this kind of programming and computing.

Arithmetic and Arithmetic

If programming were just about numbers and arithmetic, it would be as boring as mathematics. Fortunately, there is much more to programming than numbers: text, truths, images, and a great deal more.

Just kidding: mathematics is a fascinating subject, but you won't need much of it for now.

The first thing you need to know is that in BSL, text is any sequence of keyboard characters enclosed in double quotes (`"`). We call it a string. Thus, `"hello world"` is a perfectly fine string; and when DrRacket evaluates this string, it just echoes it back in the interactions area, like a number:

```
> "hello world"
"hello world"
```

Indeed, many people's first program is one that displays exactly this string.

Otherwise, you need to know that in addition to an arithmetic of numbers, DrRacket also knows about an arithmetic of strings. So here are two interactions that illustrate this form of arithmetic:

```
> (string-append "hello" "world")
"helloworld"
> (string-append "hello " "world")
"hello world"
```

Just like `+`, `string-append` is an operation; it makes a string by adding the second to the end of the first. As the first interaction shows, it does this literally, without adding anything between the two strings: no blank

space, no comma, nothing. Thus, if you want to see the phrase "hello world", you really need to add a space to one of these words somewhere; that's what the second interaction shows. Of course, the most natural way to create this phrase from the two words is to enter

```
(string-append "hello" " " "world")
```

because string-append, like +, can handle as many operands as desired.

Use F1 or the drop-down menu on the right to open HelpDesk. Look at the manuals for BSL and its section on pre-defined operations, especially those for strings.

You can do more with strings than append them. You can extract pieces from a string, reverse them, render all letters uppercase (or lowercase), strip blank spaces from the left and right, and so on. And best of all, you don't have to memorize any of that. If you need to know what you can do with strings, look up the term in HelpDesk.

If you looked up the primitive operations of BSL, you saw that *primitive* (sometimes called *pre-defined* or *built-in*) operations can consume strings and produce numbers:

```
> (+ (string-length "hello world") 20)
31
> (number->string 42)
"42"
```

There is also an operation that converts strings into numbers:

```
> (string->number "42")
42
```

If you expected "forty-two" or something clever along those lines, sorry, that's really not what you want from a string calculator.

The last expression raises a question, though. What if someone uses string->number with a string that is not a number wrapped within string quotes? In that case, the operation produces a different kind of result:

```
> (string->number "hello world")
#false
```

This is neither a number nor a string; it is a Boolean. Unlike numbers and strings, Boolean values come in only two varieties: #true and #false. The first is truth, the second falsehood. Even so, DrRacket has several operations for combining Boolean values:

```
> (and #true #true)
#true
> (and #true #false)
#false
> (or #true #false)
#true
> (or #false #false)
#false
> (not #false)
#true
```

and you get the results that the name of the operation suggests. (Don't know what `and`, `or`, and `not` compute? Easy: `(and x y)` is true if `x` and `y` are true; `(or x y)` is true if either `x` or `y` or both are true; and `(not x)` results in `#true` precisely when `x` is `#false`.)

It is also useful to "convert" two numbers into a Boolean:

```
> (> 10 9)
#true
> (< -1 0)
#true
> (= 42 9)
#false
```

Stop! Try the following three expressions: `(>= 10 10)`, `(<= -1 0)`, and `(string=? "design" "tinker")`. This last one is different again; but don't worry, you can do it.

With all these new kinds of data—yes, numbers, strings, and Boolean values are data—and operations floating around, it is easy to forget some basics, like nested arithmetic:

```
(and (or (= (string-length "hello world")
            (string->number "11"))
         (string=? "hello world" "good morning"))
     (>= (+ (string-length "hello world") 60) 80))
```

What is the result of this expression? How did you figure it out? All by yourself? Or did you just type it into DrRacket's interactions area and hit the "return" key? If you did the latter, do you think you would know how to do this on your own? After all, if you can't predict what DrRacket does for small expressions, you may not want to trust it when you submit larger tasks than that for evaluation.

To insert images such as this rocket into DrRacket, use the `Insert` *menu. Or, copy and paste the image from your browser into DrRacket.*

Before we show you how to do some "real" programming, let's discuss one more kind of data to spice things up: images. When you insert an image into the interactions area and hit "return" like this

> 

DrRacket replies with the image. In contrast to many other programming languages, BSL understands images, and it supports an arithmetic of images just as it supports an arithmetic of numbers or strings. In short, your programs can calculate with images, and you can do so in the interactions area. Furthermore, BSL programmers—like the programmers for other programming languages—create *libraries* that others may find helpful. Using such libraries is just like expanding your vocabularies with new words or your programming vocabulary with new primitives. We dub such libraries *teachpacks* because they are helpful with teaching.

Add `(require 2htdp/image)` *to the* ***definitions area****, or select* `Add Teachpack` *from the* `Language` *menu and choose* `image` *from the* `Preinstalled HtDP/2e Teachpack` *menu.*

One important library—the *2htdp/image* library—supports operations for computing the width and height of an image:

```
(* (image-width  ) (image-height  ))
```

Once you have added the library to your program, clicking *RUN* gives you `1176` because that's the area of a 28 by 42 image.

You don't have to use Google to find images and insert them in your DrRacket programs with the "Insert" menu. You can also instruct DrRacket to create simple images from scratch:

```
> (circle 10 "solid" "red")
```


```
> (rectangle 30 20 "outline" "blue")
```


When the result of an expression is an image, DrRacket draws it into the interactions area. But otherwise, a BSL program deals with images as data that is just like numbers. In particular, BSL has operations for combining images in the same way that it has operations for adding numbers or appending strings:

```
> (overlay (circle 5 "solid" "red")
```

```
            (rectangle 20 20 "solid" "blue"))
```

Overlaying these images in the opposite order produces a solid blue square:

```
> (overlay (rectangle 20 20 "solid" "blue")
           (circle 5 "solid" "red"))
```


Stop and reflect on this last result for a moment.

As you can see, `overlay` is more like `string-append` than `+`, but it does "add" images just like `string-append` "adds" strings and `+` adds numbers. Here is another illustration of the idea:

```
> (image-width (square 10 "solid" "red"))
10
> (image-width
    (overlay (rectangle 20 20 "solid" "blue")
             (circle 5 "solid" "red")))
20
```

These interactions with DrRacket don't draw anything at all; they really just measure their width.

Two more operations matter: `empty-scene` and `place-image`. The first creates a scene, a special kind of rectangle. The second places an image into such a scene:

```
(place-image (circle 5 "solid" "green")
             50 80
             (empty-scene 100 100))
```

and you get this:

Not quite. The image comes without a grid. We superimpose the grid on the empty scene so that you can see where exactly the green dot is placed.

As you can see from this image, the origin (or (0,0)) is in the upper-left corner. Unlike in mathematics, the y-coordinate is measured **downward**,

not upward. Otherwise, the image shows what you should have expected: a solid green disk at the coordinates (50,80) in a 100 by 100 empty rectangle.

Let's summarize again. To program is to write down an arithmetic expression, but you're no longer restricted to boring numbers. In BSL, arithmetic is the arithmetic of numbers, strings, Booleans, and even images. To compute, though, still means to determine the value of an expression—except that this value can be a string, a number, a Boolean, or an image.

And now you're ready to write programs that make rockets fly.

Inputs and Output

The programs you have written so far are pretty boring. You write down an expression or several expressions; you click *RUN*; you see some results. If you click *RUN* again, you see the exact same results. As a matter of fact, you can click *RUN* as often as you want, and the same results show up. In short, your programs really are like calculations on a pocket calculator, except that DrRacket calculates with all kinds of data, not just numbers.

That's good news and bad news. It is good because programming and computing ought to be a natural generalization of using a calculator. It is bad because the purpose of programming is to deal with lots of data and to get lots of different results, with more or less the same calculations. (It should also compute these results quickly, at least faster than we can.) That is, you need to learn more still before you know how to program. No need to worry though: with all your knowledge about arithmetic of numbers, strings, Boolean values, and images, you're almost ready to write a program that creates movies, not just some silly program for displaying "hello world" somewhere. And that's what we're going to do next.

Just in case you didn't know, a movie is a sequence of images that are rapidly displayed in order. If your algebra teachers had known about the "arithmetic of images" that you saw in the preceding section, you could have produced movies in algebra instead of boring number sequences. Well, here is one more such table:

x =	1	2	3	4	5	6	7	8	9	10
y =	1	4	9	16	25	36	49	64	81	?

Your teachers would now ask you to fill in the blank, that is, replace the "?" mark with a number.

It turns out that making a movie is no more complicated than completing a table of numbers like that. Indeed, it is all about such tables:

x =	1	2	3	4
y =				

To be concrete, your teacher should ask you here to draw the fourth image, the fifth, and the 1273rd one because a movie is just a lot of images, some 20 or 30 of them per second. So you need some 1200 to 1800 of them to make one minute's worth of it.

You may also recall that your teacher not only asked for the fourth or fifth number in some sequence but also for an expression that determines any element of the sequence from a given x. In the numeric example, the teacher wants to see something like this:

$$y = x \cdot x$$

If you plug in 1, 2, 3, and so on for x, you get 1, 4, 9, and so on for y—just as the table says. For the sequence of images, you could say something like

$$y = \text{ the image that contains a dot } x^2 \text{ pixels below the top.}$$

The key is that these one-liners are not just expressions but functions.

At first glance, functions are like expressions, always with a y on the left, followed by an = sign, and an expression. They aren't expressions, however. And the notation you often see in school for functions is utterly misleading. In DrRacket, you therefore write functions a bit differently:

```
(define (y x) (* x x))
```

The `define` says "consider `y` a function," which, like an expression, computes a value. A function's value, though, depends on the value of something called the *input*, which we express with `(y x)`. Since we don't know what this input is, we use a name to represent the input. Following the mathematical tradition, we use `x` here to stand in for the unknown input; but pretty soon, we will use all kinds of names.

This second part means you must supply one number—for `x`—to determine a specific value for `y`. When you do, DrRacket plugs the value for `x` into the expression associated with the function. Here the expression is `(* x x)`. Once `x` is replaced with a value, say `1`, DrRacket can compute the result of the expressions, which is also called the *output* of the function.

Click *RUN* and watch nothing happen. Nothing shows up in the interactions area. Nothing seems to change anywhere else in DrRacket. It is as if you hadn't accomplished anything. But you did. You actually defined a function and informed DrRacket about its existence. As a matter of fact, the latter is now ready for you to use the function. Enter

```
(y 1)
```

Mathematics also calls y(1) a function application, but your teachers forgot to tell you.

at the prompt in the interactions area and watch a `1` appear in response. The `(y 1)` is called a *function application* in DrRacket. Try

```
(y 2)
```

and see a `4` pop out. Of course, you can also enter all these expressions in the definitions area and click *RUN*:

```
(define (y x) (* x x))

(y 1)
(y 2)
(y 3)
(y 4)
(y 5)
```

In response, DrRacket displays: `1 4 9 16 25`, which are the numbers from the table. Now determine the missing entry.

What all this means for you is that functions provide a rather economic way of computing lots of interesting values with a single expression. Indeed, programs are functions; and once you understand functions well, you know almost everything there is to know about programming. Given their importance, let's recap what we know about functions so far:

- First,

  ```
  (define (FunctionName InputName) BodyExpression)
  ```

  is a *function definition*. You recognize it as such because it starts with the "`define`" keyword. It essentially consists of three pieces: two names and an expression. The first name is the name of the function; you need it to apply the function as often as you wish. The second name—called a *parameter*—represents the input of the function, which is unknown until you apply the function. The expression, dubbed *body*, computes the output of the function for a specific input.

- Second,

  (*FunctionName ArgumentExpression*)

  is a *function application*. The first part tells DrRacket which function you wish to use. The second part is the input to which you want to apply the function. If you were reading a Windows or a Mac manual, it might tell you that this expression "launches" the "application" called *FunctionName* and that it is going to process *ArgumentExpression* as the input. Like all expressions, the latter is possibly a plain piece of data or a deeply nested expression.

Functions can input more than numbers, and they can output all kinds of data, too. Our next task is to create a function that simulates the second table—the one with images of a colored dot—just like the first function simulated the numeric table. Since the creation of images from expressions isn't something you know from high school, let's start simply. Do you remember `empty-scene`? We quickly mentioned it at the end of the previous section. When you type it into the interactions area, like that:

```
≥ 100 60)
```


DrRacket produces an empty rectangle, also called a scene. You can add images to a scene with `place-image`:

```
50 23 (empty-scene 100 60))
```


Think of the rocket as an object that is like the dot in the above table from your mathematics class. The difference is that a rocket is interesting.

Next, you should make the rocket descend, just like the dot in the above table. From the preceding section, you know how to achieve this effect by increasing the y-coordinate that is supplied to `place-image`:

All that's needed now is to produce lots of these scenes easily and to display all of them in rapid order.

```
(define (picture-of-rocket height)
  (place-image  50 height (empty-scene 100 60)))
```

Figure 4: Landing a rocket (version 1)

In BSL, you can use all kinds of characters in names, including "-" and ".".

The first goal can be achieved with a function, of course; see figure 4. Yes, this is a function definition. Instead of `y`, it uses the name `picture-of-rocket`, a name that immediately tells you what the function outputs: a scene with a rocket. Instead of `x`, the function definition uses `height` for the name of its parameter, a name that suggests that it is a number and that it tells the function where to place the rocket. The body expression of the function is exactly like the series of expressions with which we just experimented, except that it uses `height` in place of a number. And we can easily create all of those images with this one function:

```
(picture-of-rocket 0)
```

```
(picture-of-rocket 10)
(picture-of-rocket 20)
(picture-of-rocket 30)
```

Try this out in the definitions area or the interactions area; both create the expected scenes.

The second goal requires knowledge about one additional primitive operation from the *2htdp/universe* library: `animate`. So, click *RUN* and enter the following expression:

Don't forget to add the 2htdp/universe library to your definitions area.

```
> (animate picture-of-rocket)
```

Stop and note that the argument expression is a function. Don't worry for now about using functions as arguments; it works well with `animate`, but don't try to define functions like `animate` at home just yet.

As soon as you hit the "return" key, DrRacket evaluates the expression; but it does not display a result, not even a prompt. It opens another window—a *canvas*—and starts a clock that ticks 28 times per second. Every time the clock ticks, DrRacket applies `picture-of-rocket` to the number of ticks passed since this function call. The results of these function calls are displayed in the canvas, and it produces the effect of an animated movie. The simulation runs until you close the window. At that point, `animate` returns the number of ticks that have passed.

The question is where the images on the window come from. The short explanation is that `animate` runs its operand on the numbers `0`, `1`, `2`, and so on, and displays the resulting images. The long explanation is this:

Exercise 298 explains how to design `animate`.

- `animate` starts a clock and counts the number of ticks;
- the clock ticks 28 times per second;
- every time the clock ticks, `animate` applies the function `picture-of-rocket` to the current clock tick; and
- the scene that this application creates is displayed on the canvas.

This means that the rocket first appears at height `0`, then `1`, then `2`, and so on, which explains why the rocket descends from the top of the canvas to the bottom. That is, our three-line program creates some 100 pictures in about 3.5 seconds, and displaying these pictures rapidly creates the effect of a rocket descending to the ground.

So here is what you learned in this section. Functions are useful because they can process lots of data in a short time. You can launch a function by

hand on a few select inputs to ensure that it produces the proper outputs. This is called testing a function. Or, DrRacket can launch a function on lots of inputs with the help of some libraries; when you do that, you are running the function. Naturally, DrRacket can launch functions when you press a key on your keyboard or when you manipulate the mouse of your computer. To find out how, keep reading. Whatever triggers a function application isn't important, but do keep in mind that (simple) programs are functions.

Many Ways to Compute

When you evaluate `(animate picture-of-rocket)`, the rocket eventually disappears into the ground. That's plain silly. Rockets in old science fiction movies don't sink into the ground; they gracefully land on their bottoms, and the movie should end right there.

This idea suggests that computations should proceed differently, depending on the situation. In our example, the `picture-of-rocket` program should work "as is" while the rocket is in flight. When the rocket's bottom touches the bottom of the canvas, however, it should stop the rocket from descending any farther.

In a sense, the idea shouldn't be new to you. Even your mathematics teachers define functions that distinguish various situations:

$$sign(x) = \begin{cases} +1 \text{ if } x > 0 \\ 0 \text{ if } x = 0 \\ -1 \text{ if } x < 0 \end{cases}$$

This *sign* function distinguishes three kinds of inputs: those numbers that are larger than 0, those equal to 0, and those smaller than 0. Depending on the input, the result of the function is +1, 0, or -1.

You can define this function in DrRacket without much ado using a `cond`itional expression:

```
(define (sign x)
  (cond
    [(> x 0) 1]
    [(= x 0) 0]
    [(< x 0) -1]))
```

Open a new tab in DrRacket and start with a clean slate.

After you click *RUN*, you can interact with `sign` like any other function:

```
> (sign 10)
1
> (sign -5)
-1
> (sign 0)
0
```

This is a good time to explore what the STEP button does. Add `(sign -5)` *to the definitions area and click STEP for the above* `sign` *program. When the new window comes up, click the right and left arrows there.*

In general, a *conditional expression* has the shape

```
(cond
  [ConditionExpression1 ResultExpression1]
  [ConditionExpression2 ResultExpression2]
  ...
  [ConditionExpressionN ResultExpressionN])
```

That is, a `cond`itional expression consists of as many *conditional lines* as needed. Each line contains two expressions: the left one is often called *condition*, and the right one is called *result*; occasionally we also use *question* and *answer*. To evaluate a `cond` expression, DrRacket evaluates the first condition expression, *`ConditionExpression1`*. If this yields `#true`, DrRacket replaces the `cond` expression with *`ResultExpression1`*, evaluates it, and uses the value as the result of the entire `cond` expression. If the evaluation of *`ConditionExpression1`* yields `#false`, DrRacket drops the first line and starts over. In case all condition expressions evaluate to `#false`, DrRacket signals an error.

With this knowledge, you can now change the course of the simulation. The goal is to not let the rocket descend below the ground level of a 100-by-60 scene. Since the `picture-of-rocket` function consumes the height where it should place the rocket in the scene, a simple test comparing the given height to the maximum height appears to suffice.

See figure 5 for the revised function definition. The definition uses the name `picture-of-rocket.v2` to distinguish the two versions. Using distinct names also allows us to use both functions in the interactions area and to compare the results. Here is how the original version works:

```
                -rocket 5555)
```


And here is the second one:

```
(define (picture-of-rocket.v2 height)
  (cond
    [(<= height 60)
     (place-image
      50 height
      (empty-scene 100 60))]
    [(> height 60)
     (place-image
      50 60
      (empty-scene 100 60))]))
```

Figure 5: Landing a rocket (version 2)

```
-rocket.v2 5555)
```


No matter what number you give to `picture-of-rocket.v2`, if it is over `60`, you get the same scene. In particular, when you run

```
> (animate picture-of-rocket.v2)
```

the rocket descends and sinks halfway into the ground before it stops.

Stop! What do you think we want to see?

Landing the rocket this far down is ugly. Then again, you know how to fix this aspect of the program. As you have seen, BSL knows an arithmetic of images. When `place-image` adds an image to a scene, it uses its center point as if it were the whole image, even though the image has a real height and a real width. As you may recall, you can measure the height of an image with the operation `image-height`. This function comes in handy here because you really want to fly the rocket only until its bottom touches the ground.

Putting one and one together you can now figure out that

```
(- 60 (/ (image-height ) 2))
```

is the point at which you want the rocket to stop its descent. You could figure this out by playing with the program directly, or you can experiment in the interactions area with your image arithmetic.

Here is a first attempt:

```
(place-image  50 (- 60 (image-height  ))
              (empty-scene 100 60))
```

Now replace the third argument in the above application with

```
(- 60 (/ (image-height  ) 2))
```

Stop! Conduct the experiments. Which result do you like better?

```
(define (picture-of-rocket.v3 height)
  (cond

    [(<= height (- 60 (/ (image-height  ) 2)))

     (place-image  50 height
                   (empty-scene 100 60))]

    [(> height (- 60 (/ (image-height  ) 2)))

     (place-image  50 (- 60 (/ (image-height  ) 2))
                   (empty-scene 100 60))]))
```

Figure 6: Landing a rocket (version 3)

When you think and experiment along these lines, you eventually get to the program in figure 6. Given some number, which represents the `height` of the rocket, it first tests whether the rocket's bottom is above the ground. If it is, it places the rocket into the scene as before. If it isn't, it places the rocket's image so that its bottom touches the ground.

One Program, Many Definitions

Now suppose your friends watch the animation but don't like the size of your canvas. They might request a version that uses `200`-by-`400` scenes. This simple request forces you to replace `100` with `400` in five places in the program and `60` with `200` in two other places—not to speak of the occurrences of `50`, which really means "middle of the canvas."

Stop! Before you read on, try to do just that so that you get an idea of how difficult it is to execute this request for a five-line program. As you read on, keep in mind that programs in the world consist of 50,000 or 500,000 or even 5,000,000 or more lines of program code.

In the ideal program, a small request, such as changing the sizes of the canvas, should require an equally small change. The tool to achieve this simplicity with BSL is `define`. In addition to defining functions, you can also introduce *constant definitions*, which assign some name to a constant. The general shape of a constant definition is straightforward:

```
(define Name Expression)
```

Thus, for example, if you write down

```
(define HEIGHT 60)
```

in your program, you are saying that `HEIGHT` always represents the number `60`. The meaning of such a definition is what you expect. Whenever DrRacket encounters `HEIGHT` during its calculations, it uses `60` instead.

Now take a look at the code in figure 7, which implements this simple change and also names the image of the rocket. Copy the program into DrRacket; and after clicking *RUN*, evaluate the following interaction:

```
> (animate picture-of-rocket.v4)
```

Confirm that the program still functions as before.

The program in figure 7 consists of four definitions: one function definition and three constant definitions. The numbers `100` and `60` occur only twice—once as the value of `WIDTH` and once as the value of `HEIGHT`. You may also have noticed that it uses `h` instead of `height` for the function parameter of `picture-of-rocket.v4`. Strictly speaking, this change isn't necessary because DrRacket doesn't confuse `height` with `HEIGHT`, but we did it to avoid confusing **you**.

When DrRacket evaluates `(animate picture-of-rocket.v4)`, it replaces `HEIGHT` with `60`, `WIDTH` with `100`, and `ROCKET` with the image every time it encounters these names. To experience the joys of real programmers, change the `60` next to `HEIGHT` into a `400` and click *RUN*. You see a

```
; constants
(define WIDTH  100)
(define HEIGHT  60)
(define MTSCN  (empty-scene WIDTH HEIGHT))

(define ROCKET  )
(define ROCKET-CENTER-TO-TOP
  (- HEIGHT (/ (image-height ROCKET) 2)))

; functions
(define (picture-of-rocket.v5 h)
  (cond
    [(<= h ROCKET-CENTER-TO-TOP)
     (place-image ROCKET 50 h MTSCN)]
    [(> h ROCKET-CENTER-TO-TOP)
     (place-image ROCKET 50 ROCKET-CENTER-TO-TOP MTSCN)]))
```

Figure 8: Landing a rocket (version 5)

- How would you change the program so that the background is always blue?
- How would you change the program so that the rocket lands on a flat rock bed that is 10 pixels higher than the bottom of the scene? Don't forget to change the scenery, too.

Better than pondering is doing. It's the only way to learn. So don't let us stop you. Just do it.

Magic Numbers Take another look at `picture-of-rocket.v5`. Because we eliminated all repeated expressions, all but one number disappeared from this function definition. In the world of programming, these numbers are called *magic numbers*, and nobody likes them. Before you know it, you forget what role the number plays and what changes are legitimate. It is best to name such numbers in a definition.

Here we actually know that `50` is our choice for an x-coordinate for the rocket. Even though `50` doesn't look like much of an expression, it really is a repeated expression, too. Thus, we have two reasons to eliminate `50` from the function definition, and we leave it to you to do so.

One More Definition

Danger ahead! This section introduces one piece of knowledge from physics. If physics scares you, skip it on a first reading; programming doesn't require physics knowledge.

Recall that `animate` actually applies its functions to the number of clock ticks that have passed since it was first called. That is, the argument to `picture-of-rocket` isn't a height but a time. Our previous definitions of `picture-of-rocket` use the wrong name for the argument of the function; instead of `h`—short for height—it ought to use `t` for time:

```
(define (picture-of-rocket t)
  (cond
    [(<= t ROCKET-CENTER-TO-TOP)
     (place-image ROCKET 50 t MTSCN)]
    [(> t ROCKET-CENTER-TO-TOP)
     (place-image ROCKET
                  50 ROCKET-CENTER-TO-TOP
                  MTSCN)]))
```

And this small change to the definition immediately clarifies that this program uses time as if it were a distance. What a bad idea.

Even if you have never taken a physics course, you know that a time is not a distance. So somehow our program worked by accident. Don't worry, though; it is all easy to fix. All you need to know is a bit of rocket science, which people like us call physics.

Physics?!? Well, perhaps you have already forgotten what you learned in that course. Or perhaps you have never taken a course on physics because you are way too young or gentle. No worries. This happens to the best programmers all the time because they need to help people with problems in music, economics, photography, nursing, and all kinds of other disciplines. Obviously, not even programmers know everything. So they look up what they need to know. Or they talk to the right kind of people. And if you talk to a physicist, you will find out that the distance traveled is proportional to the time:

$$d = v \cdot t$$

That is, if the velocity of an object is v, then the object travels d miles (or meters or pixels or whatever) in t seconds.

Of course, a teacher ought to show you a proper function definition:

$$d(t) = v \cdot t$$

because this tells everyone immediately that the computation of d depends on t and that v is a constant. A programmer goes even further and uses meaningful names for these one-letter abbreviations:

```
(define V 3)

(define (distance t)
  (* V t))
```

This program fragment consists of two definitions: a function `distance` that computes the distance traveled by an object traveling at a constant velocity, and a constant `V` that describes the velocity.

You might wonder why `V` is `3` here. There is no special reason. We consider `3` pixels per clock tick a good velocity. You may not. Play with this number and see what happens with the animation.

```
; properties of the "world" and the descending rocket
(define WIDTH  100)
(define HEIGHT  60)
(define V 3)
(define X 50)

; graphical constants
(define MTSCN  (empty-scene WIDTH HEIGHT))

(define ROCKET  )
(define ROCKET-CENTER-TO-TOP
  (- HEIGHT (/ (image-height ROCKET) 2)))

; functions
(define (picture-of-rocket.v6 t)
  (cond
    [(<= (distance t) ROCKET-CENTER-TO-TOP)
     (place-image ROCKET X (distance t) MTSCN)]
    [(> (distance t) ROCKET-CENTER-TO-TOP)
     (place-image ROCKET X ROCKET-CENTER-TO-TOP MTSCN)]))

(define (distance t)
  (* V t))
```

Figure 9: Landing a rocket (version 6)

Now we can fix `picture-of-rocket` again. Instead of comparing `t` with a height, the function can use `(distance t)` to calculate how far

down the rocket is. The final program is displayed in figure 9. It consists of two function definitions: `picture-of-rocket.v6` and `distance`. The remaining constant definitions make the function definitions readable and modifiable. As always, you can run this program with `animate`:

```
> (animate picture-of-rocket.v6)
```

In comparison to the previous versions of `picture-of-rocket`, this one shows that a program may consist of several function definitions that refer to each other. Then again, even the first version used `+` and `/`—it's just that you think of those as built into BSL.

As you become a true-blue programmer, you will find out that programs consist of many function definitions and many constant definitions. You will also see that functions refer to each other all the time. What you really need to practice is to organize them so that you can read them easily, even months after completion. After all, an older version of you—or someone else—will want to make changes to these programs; and if you cannot understand the program's organization, you will have a difficult time with even the smallest task. Otherwise, you mostly know what there is to know.

You Are a Programmer Now

The claim that you are a programmer may have come as a surprise to you at the end of the preceding section, but it is true. You know all the mechanics that there are to know about BSL. You know that programming uses the arithmetic of numbers, strings, images, and whatever other data your chosen programming languages support. You know that programs consist of function and constant definitions. You know, because we have told you, that in the end, it's all about organizing these definitions properly. Last but not least, you know that DrRacket and the teachpacks support lots of other functions and that DrRacket's HelpDesk explains what these functions do.

You might think that you still don't know enough to write programs that react to keystrokes, mouse clicks, and so on. As it turns out, you do. In addition to the `animate` function, the *2htdp/universe* library provides other functions that hook up your programs to the keyboard, the mouse, the clock, and other moving parts in your computer. Indeed, it even supports writing programs that connect your computer with anybody else's computer around the world. So this isn't really a problem.

In short, you have seen almost all the mechanics of putting together programs. If you read up on all the functions that are available, you can

write programs that play interesting computer games, run simulations, or keep track of business accounts. The question is whether this really means you are a programmer. Are you?

Stop! Don't turn the page yet. Think!

I FIXED-SIZE DATA

Every programming language comes with a language of data and a language of operations on data. The first language always provides some forms of atomic data; to represent the variety of information in the real world as data, a programmer must learn to compose basic data and to describe such compositions. Similarly, the second language provides some basic operations on atomic data; it is the programmer's task to compose these operations into programs that perform the desired computations. We use *arithmetic* for the combination of these two parts of a programming language because it generalizes what you know from grade school.

This first part of the book (I) introduces the arithmetic of BSL, the programming language used in the Prologue. From arithmetic, it is a short step to your first simple programs, which you may know as *functions* from mathematics. Before you know it, though, the process of writing programs looks confusing, and you will long for a way to organize your thoughts. We equate "organizing thoughts" with *design*, and this first part of the book introduces you to a systematic way of designing programs.

1 Arithmetic

Scan this first chapter quickly, skip ahead to the second one, and return here, when you encounter "arithmetic" that you don't recognize.

From the Prologue, you know how to write down the kind of *expression* you know from first grade in BSL notation:

- write "`(`",
- write down the name of a primitive operation `op`,
- write down the arguments, separated by some space, and
- write down "`)`".

Just as a reminder, here is a primitive expression:

```
(+ 1 2)
```

It uses +, the operation for adding two numbers, followed by two arguments, which are plain numbers. But here is another example:

```
(+ 1 (+ 1 (+ 1 1) 2) 3 4 5)
```

This second example exploits two points in the above description that are open to interpretation. First, primitive operations may consume more than two arguments. Second, the arguments don't have to be numbers per se; they can be expressions, too.

Evaluating expressions is also straightforward. First, BSL evaluates all the arguments of a primitive operation. Second, it "feeds" the resulting pieces of data to the operation, which produces a result. Thus,

```
(+ 1 2)
==
3
```

We use == to mean "is equal to according to the laws of computation."

and

```
(+ 1 (+ 1 (+ 1 1) 2) 3 (+ 2 2) 5)
==
(+ 1 (+ 1 2 2) 3 4 5)
==
(+ 1 5 3 4 5)
==
18
```

These calculations should look familiar because they are the same kind of calculations that you performed in mathematics classes. You may have written down the steps in a different way; you may have never been taught how to write down a sequence of calculation steps. Yet, BSL performs calculations just like you do, and this should be a relief. It guarantees that you understand what it does with primitive operations and primitive data, so there is some hope that you can predict what your programs will compute. Generally speaking, it is critical for a programmer to know how the chosen language calculates because otherwise a program's computation may harm the people who use them or on whose behalf the programs calculate.

The next volume, How to Design Components, *will explain how to design atomic data.*

The rest of this chapter introduces four forms of *atomic data* of BSL: numbers, strings, images, and Boolean values. We use the word "atomic" here in analogy to physics. You cannot peek inside atomic pieces of data, but you do have functions that combine several pieces of atomic data into another one, retrieve "properties" of them, also in terms of atomic data, and

so on. The sections of this chapter introduce some of these functions, also called *primitive operations* or *pre-defined operations*. You can find others in the documentation of BSL that comes with DrRacket.

1.1 The Arithmetic of Numbers

Most people think "numbers" and "operations on numbers" when they hear "arithmetic." "Operations on numbers" means adding two numbers to yield a third, subtracting one number from another, determining the greatest common divisor of two numbers, and many more such things. If we don't take arithmetic too literally, we may even include the sine of an angle, rounding a real number to the closest integer, and so on.

The BSL language supports *Number*s and arithmetic on them. As discussed in the Prologue, an arithmetic operation such as + is used like this:

```
(+ 3 4)
```

that is, in *prefix notation* form. Here are some of the operations on numbers that our language provides: `+`, `-`, `*`, `/`, `abs`, `add1`, `ceiling`, `denominator`, `exact->inexact`, `expt`, `floor`, `gcd`, `log`, `max`, `numerator`, `quotient`, `random`, `remainder`, `sqr`, and `tan`. We picked our way through the alphabet just to show the variety of operations. Explore what they compute, and then find out how many more there are.

If you need an operation on numbers that you know from your mathematics courses, chances are that BSL knows about it, too. Guess its name and experiment in the interactions area. Say you need to compute the *sin* of some angle; try

```
> (sin 0)
0
```

and use it happily ever after. Or look in the HelpDesk. You will find there that in addition to operations BSL also recognizes the names of some widely used numbers, for example, `pi` and `e`.

You might know `e` *from calculus. It's a real number, close to 2.718, called "Euler's constant."*

When it comes to numbers, BSL programs may use natural numbers, integers, rational numbers, real numbers, and complex numbers. We assume that you have heard of all but the last one. The last one may have been mentioned in your high school class. If not, don't worry; while complex numbers are useful for all kinds of calculations, a novice doesn't have to know about them.

A truly important distinction concerns the precision of numbers. For now, it is important to understand that BSL distinguishes *exact numbers* and *inexact numbers*. When it calculates with exact numbers, BSL preserves this precision whenever possible. For example, `(/ 4 6)` produces the precise fraction `2/3`, which DrRacket can render as a proper fraction, an improper fraction, or a mixed decimal. Play with your computer's mouse to find the menu that changes the fraction into decimal expansion.

Some of BSL's numeric operations cannot produce an exact result. For example, using the `sqrt` operation on `2` produces an irrational number that cannot be described with a finite number of digits. Because computers are of finite size and BSL must somehow fit such numbers into the computer, it chooses an approximation: `1.4142135623730951`. As mentioned in the Prologue, the `#i` prefix warns novice programmers of this lack of precision. While most programming languages choose to reduce precision in this manner, few advertise it and even fewer warn programmers.

Note on Numbers The word "Number" refers to a wide variety of numbers, including counting numbers, integers, rational numbers, real numbers, and even complex numbers. For most uses, you can safely equate Number with the number line from elementary school, though on occasion this translation is too imprecise. If we wish to be precise, we use appropriate words: *Integer*, *Rational*, and so on. We may even refine these notions using such standard terms as *PositiveInteger*, *NonnegativeNumber*, *NegativeNumber*, and so on. **End**

Exercise 1. Add the following definitions for `x` and `y` to DrRacket's definitions area:

```
(define x 3)
(define y 4)
```

Now imagine that `x` and `y` are the coordinates of a Cartesian point. Write down an expression that computes the distance of this point to the origin, that is, a point with the coordinates (`0`,`0`).

The expected result for these values is `5`, but your expression should produce the correct result even after you change these definitions.

Just in case you have not taken geometry courses or in case you forgot the formula that you encountered there, the point (*x*,*y*) has the distance

$$\sqrt{x^2 + y^2}$$

from the origin. After all, we are teaching you how to design programs, not how to be a geometer.

1.3 Mixing It Up

All other operations (in BSL) concerning strings consume or produce data other than strings. Here are some examples:

- `string-length` consumes a string and produces a number;
- `string-ith` consumes a string `s` together with a number `i` and extracts the 1String located at the `i`th position (counting from 0); and
- `number->string` consumes a number and produces a string.

Also look up `substring` and find out what it does.

If the documentation in HelpDesk appears confusing, experiment with the functions in the interactions area. Give them appropriate arguments, and find out what they compute. Also use **inappropriate** arguments for some operations just to find out how BSL reacts:

```
> (string-length 42)
string-length:expects a string, given 42
```

As you can see, BSL reports an error. The first part "string-length" informs you about the operation that is misapplied; the second half states what is wrong with the arguments. In this specific example, `string-length` is supposed to be applied to a string but is given a number, specifically `42`.

Naturally, it is possible to nest operations that consume and produce different kinds of data **as long as you keep track of what is proper and what is not**. Consider this expression from the the Prologue:

```
(+ (string-length "hello world") 20)
```

The inner expression applies `string-length` to `"hello world"`, our favorite string. The outer expression has `+` consume the result of the inner expression and `20`.

Let's determine the result of this expression in a step-by-step fashion:

```
(+ (string-length "hello world") 20)
==
(+ 11 20)
==
31
```

Not surprisingly, computing with such nested expressions that deal with a mix of data is no different from computing with numeric expressions. Here is another example:

```
(+ (string-length (number->string 42)) 2)
==
(+ (string-length "42") 2)
==
(+ 2 2)
==
4
```

Before you go on, construct some nested expressions that mix data in the **wrong** way, say,

```
(+ (string-length 42) 1)
```

Run them in DrRacket. Study the red error message but also watch what DrRacket highlights in the definitions area.

Exercise 3. Add the following two lines to the definitions area:

```
(define str "helloworld")
(define i 5)
```

Then create an expression using string primitives that adds `"_"` at position `i`. In general this means the resulting string is longer than the original one; here the expected result is `"hello_world"`.

Position means *i* characters from the left of the string, but programmers start counting at `0`. Thus, the 5*th* letter in this example is `"w"`, because the `0`th letter is `"h"`. **Hint** When you encounter such "counting problems" you may wish to add a string of digits below `str` to help with counting:

```
(define str "helloworld")
(define ind "0123456789")
(define i 5)
```

See exercise 1 for how to create expressions in DrRacket. ∎

Exercise 4. Use the same setup as in exercise 3 to create an expression that deletes the *i*th position from `str`. Clearly this expression creates a shorter string than the given one. Which values for `i` are legitimate? ∎

1.4 The Arithmetic of Images

Remember to require the 2htdp/image library in a new tab.

An *Image* is a visual, rectangular piece of data, for example, a photo or a geometric figure and its frame. You can insert images in DrRacket wher-

ever you can write down an expression because images are values, just like numbers and strings.

Your programs can also manipulate images with primitive operations. These primitive operations come in three flavors. The first kind concerns the creation of basic images:

- `circle` produces a circle image from a radius, a mode string, and a color string;
- `ellipse` produces an ellipse from two radii, a mode string, and a color string;
- `line` produces a line from two points and a color string;
- `rectangle` produces a rectangle from a width, a height, a mode string, and a color string;
- `text` produces a text image from a string, a font size, and a color string; and
- `triangle` produces an upward-pointing equilateral triangle from a size, a mode string, and a color string.

The names of these operations mostly explain what kind of image they create. All you must know is that *mode strings* means `"solid"` or `"outline"`, and *color strings* are strings such as `"orange"`, `"black"`, and so on.

Play with these operations in the interactions window:

```
> (circle 10 "solid" "green")
```

```
> (rectangle 10 20 "solid" "blue")
```



```
> (star 12 "solid" "gray")
```

Stop! The above uses a previously unmentioned operation. Look up its documentation and find out how many more such operations the *2htdp/image* library comes with. Experiment with the operations you find.

The second kind of functions on images concern image properties:

- `image-width` determines the width of an image in terms of pixels;
- `image-height` determines the height of an image;

They extract the kind of values from images that you expect:

```
> (image-width (circle 10 "solid" "red"))
20
> (image-height (rectangle 10 20 "solid" "blue"))
20
```

Stop! Explain how DrRacket determines the value of this expression:

```
(+ (image-width (circle 10 "solid" "red"))
   (image-height (rectangle 10 20 "solid" "blue")))
```

A proper understanding of the third kind of image-composing primitives requires the introduction of one new idea: the *anchor point*. An image isn't just a single pixel, it consists of many pixels. Specifically, each image is like a photograph, that is, a rectangle of pixels. One of these pixels is an implicit anchor point. When you use an image primitive to compose two images, the composition happens with respect to the anchor points, unless you specify some other point explicitly:

- `overlay` places all the images to which it is applied on top of each other, using the center as anchor point;
- `overlay/xy` is like `overlay` but accepts two numbers—`x` and `y`—between two image arguments. It shifts the second image by `x` pixels to the right and `y` pixels down—all with respect to the first image's top-left corner; unsurprisingly, a negative `x` shifts the image to the left and a negative `y` up; and
- `overlay/align` is like `overlay` but accepts two strings that shift the anchor point(s) to other parts of the rectangles. There are nine different positions overall; experiment with all possibilities!

The *2htdp/image* library comes with many other primitive functions for combining images. As you get familiar with image processing, you will want to read up on those. For now, we introduce three more because they are important for creating animated scenes and images for games:

- `empty-scene` creates a rectangle of some given width and height;
- `place-image` places an image into a scene at a specified position. If the image doesn't fit into the given scene, it is appropriately cropped;

arithmetic of numbers	arithmetic of images

`(+ 1 1) == 2`	`(overlay (square 4 "solid" "orange")`
	`(circle 6 "solid" "yellow"))`
	`==`
	 

`(+ 1 2) == 3`	`(underlay (circle 6 "solid" "yellow")`
	`(square 4 "solid" "orange"))`
	`==`
	 

`(+ 2 2) == 4`	`(place-image (circle 6 "solid" "yellow")`
	`10 10`
	`(empty-scene 20 20))`
	`==`
	 

...	...

Figure 10: Laws of image creation

- `scene+line` consumes a scene, four numbers, and a color to draw a line into the given image. Experiment with it to see how it works.

The laws of arithmetic for images are analogous to those for numbers; see figure 10 for some examples and a comparison with numeric arithmetic. Again, no image gets destroyed or changed. Like `+`, these primitives just make up new images that combine the given ones in some manner.

Exercise 5. Use the *2htdp/image* library to create the image of a simple boat or tree. Make sure you can easily change the scale of the entire image. ∎

Copy and paste the image into your DrRacket.

Exercise 6. Add the following line to the definitions area:


```
(define cat
```

Create an expression that counts the number of pixels in the image. ∎

1. The first expression is always evaluated. Its result must be a Boolean.
2. If the result of the first expression is `#true`, then the second expression is evaluated; otherwise the third one is. Whatever their results are, they are also the result of the entire `if` expression.

Right-click on the result and choose a different representation.

Given the definition of `x` above, you can experiment with `if` expressions in the interactions area:

```
> (if (= x 0) 0 (/ 1 x))
0.5
```

Using the laws of arithmetic, you can figure out the result yourself:

```
(if (= x 0) 0 (/ 1 x))
== ; because x stands for 2
(if (= 2 0) 0 (/ 1 2))
== ; 2 is not equal to 0, (= 2 0) is #false
(if #false 0 (/ 1 x))
(/ 1 2)
== ; normalize this to its decimal representation
0.5
```

In other words, DrRacket knows that `x` stands for `2` and that the latter is not equal to `0`. Hence, `(= x 0)` produces the result `#false`, meaning `if` picks its third sub-expression to be evaluated.

Stop! Imagine you edit the definition so that it looks like this:

```
(define x 0)
```

What do you think

```
(if (= x 0) 0 (/ 1 x))
```

evaluates to in this context? Why? Show your calculation.

In addition to =, BSL provides a host of other comparison primitives. Explain what the following four comparison primitives determine about numbers: <, <=, >, >=.

Strings aren't compared with = and its relatives. Instead, you must use `string=?` or `string<=?` or `string>=?` if you ever need to compare strings. While it is obvious that `string=?` checks whether the two given strings are equal, the other two primitives are open to interpretation. Look

up their documentation. Or, experiment, guess a general law, and then check in the documentation whether you guessed right.

You may wonder why it is ever necessary to compare strings with each other. So imagine a program that deals with traffic lights. It may use the strings `"green"`, `"yellow"`, and `"red"`. This kind of program may contain a fragment such as this:

```
(define current-color ...)



(define next-color
  (if (string=? "green" current-color) "yellow" ...))
```

The dots in the definition of `current-color` *aren't a part of the program, of course. Replace them with a string that refers to a color.*

It should be easy to imagine that this fragment deals with the computation that determines which light bulb is to be turned on next and which one should be turned off.

The next few chapters introduce better expressions than `if` to express conditional computations and, most importantly, systematic ways for designing them.

Exercise 8. Add the following line to the definitions area:


```
(define cat
```

Create a conditional expression that computes whether the image is tall or wide. An image should be labeled `"tall"` if its height is larger than or equal to its width; otherwise it is `"wide"`. See exercise 1 for how to create such expressions in DrRacket; as you experiment, replace the cat with a rectangle of your choice to ensure that you know the expected answer.

Now try the following modification. Create an expression that computes whether a picture is `"tall"`, `"wide"`, or `"square"`. ∎

1.7 Predicates: Know Thy Data

Remember the expression `(string-length 42)` and its result. Actually, the expression doesn't have a result, it signals an error. DrRacket lets you

know about errors via red text in the interactions area and highlighting of the faulty expression (in the definitions area). This way of marking errors is particularly helpful when you use this expression (or its relatives) deeply nested within some other expression:

```
(* (+ (string-length 42) 1) pi)
```

Experiment with this expression by entering it both into DrRacket's interactions area and in the definitions area (and then click on *RUN*).

Of course, you really don't want such error-signaling expressions in your program. And usually, you don't make such obvious mistakes as using `42` as a string. It is quite common, however, that programs deal with variables that may stand for either a number or a string:

```
(define in ...)

(string-length in)
```

A variable such as `in` can be a placeholder for any value, including a number, and this value then shows up in the `string-length` expression.

One way to prevent such accidents is to use a *predicate*, which is a function that consumes a value and determines whether or not it belongs to some class of data. For example, the predicate `number?` determines whether the given value is a number or not:

```
> (number? 4)
#true
> (number? pi)
#true
> (number? #true)
#false
> (number? "fortytwo")
#false
```

As you see, the predicates produce Boolean values. Hence, when predicates are combined with conditional expressions, programs can protect expressions from misuse:

```
(define in ...)

(if (string? in) (string-length in) ...)
```

Every class of data that we introduced in this chapter comes with a predicate. Experiment with `number?`, `string?`, `image?`, and `boolean?` to ensure that you understand how they work.

In addition to predicates that distinguish different forms of data, programming languages also come with predicates that distinguish different kinds of numbers. In BSL, numbers are classified in two ways: by construction and by their exactness. Construction refers to the familiar sets of numbers: `integer?`, `rational?`, `real?`, and `complex?`, but many programming languages, including BSL, also choose to use finite approximations to well-known constants, which leads to somewhat surprising results with the `rational?` predicate:

```
> (rational? pi)
#true
```

As for exactness, we have mentioned the idea before. For now, experiment with `exact?` and `inexact?` to make sure they perform the checks that their names suggest. Later we are going to discuss the nature of numbers in some detail.

Put `(sqrt -1)` *at the prompt in the interactions area and hit the "enter" key. Take a close look at the result. The result you see is the first so-called complex number anyone encounters. While your teacher may have told you that one doesn't compute the square root of negative numbers, the truth is that mathematicians and some programmers find it acceptable and useful to do so anyway. But don't worry: understanding complex numbers is not essential to being a program designer.*

Exercise 9. Add the following line to the definitions area of DrRacket:

```
(define in ...)
```

Then create an expression that converts the value of `in` to a positive number. For a String, it determines how long the String is; for an Image, it uses the area; for a Number, it decrements the number by `1`, unless it is already `0` or negative; for `#true` it uses `10` and for `#false` `20`.

See exercise 1 for how to create expressions in DrRacket. ∎

Exercise 10. Now relax, eat, sleep, and then tackle the next chapter. ∎

2 Functions and Programs

As far as programming is concerned, "arithmetic" is half the game; the other half is "algebra." Of course, "algebra" relates to the school notion of algebra as little/much as the notion of "arithmetic" from the preceding chapter relates to arithmetic taught in grade-school arithmetic. Specifically, the algebra notions needed are variable, function definition, function application, and function composition. This chapter reacquaints you with these notions in a fun and accessible manner.

2.1 Functions

Programs are functions. Like functions, programs consume inputs and produce outputs. Unlike the functions you may know, programs work with a variety of data: numbers, strings, images, mixtures of all these, and so on. Furthermore, programs are triggered by events in the real world, and the outputs of programs affect the real world. For example, a spreadsheet program may react to an accountant's key presses by filling some cells with numbers, or the calendar program on a computer may launch a monthly payroll program on the last day of every month. Lastly, a program may not consume all of its input data at once, instead it may decide to process data in an incremental manner.

Definitions While many programming languages obscure the relationship between programs and functions, BSL brings it to the fore. Every BSL program consists of several definitions, usually followed by an expression that involves those definitions. There are two kinds of definitions:

- *constant definitions*, of the shape `(define Variable Expression)`, which we encountered in the preceding chapter; and
- *function definitions*, which come in many flavors, one of which we used in the Prologue.

Like expressions, function definitions in BSL come in a uniform shape:

```
(define (FunctionName Variable ... Variable)
  Expression)
```

That is, to define a function, we write down

- "`(define (`",
- the name of the function,
- followed by several variables, separated by space and ending in "`)`",
- and an expression followed by "`)`".

And that is all there is to it. Here are some small examples:

- `(define (f x) 1)`
- `(define (g x y) (+ 1 1))`
- `(define (h x y z) (+ (* 2 2) 3))`

Functions don't have to be applied at the prompt in the interactions area. It is perfectly acceptable to use function applications nested within other function applications:

```
> (+ (ff 3) 2)
32
> (* (ff 4) (+ (ff 3) 2))
1280
> (ff (ff 1))
100
```

Exercise 11. Define a function that consumes two numbers, x and y, and that computes the distance of point (x,y) to the origin.

In exercise 1 you developed the right-hand side of this function for concrete values of x and y. Now add a header. ∎

Exercise 12. Define the function `cvolume`, which accepts the length of a side of an equilateral cube and computes its volume. If you have time, consider defining `csurface`, too.

Hint An equilateral cube is a three-dimensional container bounded by six squares. You can determine the surface of a cube if you know that the square's area is its length multiplied by itself. Its volume is the length multiplied with the area of one of its squares. (Why?) ∎

Exercise 13. Define the function `string-first`, which extracts the first 1String from a **non-empty** string. ∎

Exercise 14. Define the function `string-last`, which extracts the last 1String from a non-empty string. ∎

Exercise 15. Define `==>`. The function consumes two Boolean values, call them `sunny` and `friday`. Its answer is `#true` if `sunny` is false or `friday` is true. **Note** Logicians call this Boolean operation *implication*, and they use the notation *sunny* => *friday* for this purpose. ∎

Exercise 16. Define the function `image-area`, which counts the number of pixels in a given image. See exercise 6 for ideas. ∎

Exercise 17. Define the function `image-classify`, which consumes an image and conditionally produces `"tall"` if the image is taller than wide, `"wide"` if it is wider than tall, or `"square"` if its width and height are the same. See exercise 8 for ideas. ∎

Exercise 18. Define the function `string-join`, which consumes two strings and appends them with `"_"` in between. See exercise 2 for ideas. ∎

Exercise 19. Define the function `string-insert`, which consumes a string `str` plus a number `i` and inserts `"_"` at the ith position of `str`. As-

sume `i` is a number between `0` and the length of the given string (inclusive). See exercise 3 for ideas. Ponder how `string-insert` copes with `""`. ∎

Exercise 20. Define the function `string-delete`, which consumes a string plus a number `i` and deletes the *i*th position from `str`. Assume `i` is a number between `0` (inclusive) and the length of the given string (exclusive). See exercise 4 for ideas. Can `string-delete` deal with empty strings? ∎

2.2 Computing

Function definitions and applications work in tandem. If you want to design programs, you must understand this collaboration because you need to imagine how DrRacket runs your programs and because you need to figure out **what** goes wrong **when** things go wrong—and they **will** go wrong.

While you may have seen this idea in an algebra course, we prefer to explain it our way. So here we go. Evaluating a function application proceeds in three steps: DrRacket determines the values of the argument expressions; it checks that the number of arguments and the number of function parameters are the same; if so, DrRacket computes the value of the body of the function, with all parameters replaced by the corresponding argument values. This last value is the value of the function application. This is a mouthful, so we need examples.

Here is a sample calculation for `f`:

```
(f (+ 1 1))
== ; DrRacket knows that (+ 1 1) == 2
(f 2)
== ; DrRacket replaced all occurrences of x with 2
1
```

That last equation is weird because `x` does not occur in the body of `f`. Therefore, replacing the occurrences of `x` with `2` in the function body produces `1`, which is the function body itself.

For `ff`, DrRacket performs a different kind of computation:

```
(ff (+ 1 1))
== ; DrRacket again knows that (+ 1 1) == 2
(ff 2)
== ; DrRacket replaces a with 2 in ff's body
(* 10 2)
== ; and from here, DrRacket uses plain arithmetic
20
```

The best point is that when you combine these laws of computation with those of arithmetic, you can pretty much predict the outcome of any program in BSL:

```
(+ (ff (+ 1 2)) 2)
== ; DrRacket knows that (+ 1 2) == 3
(+ (ff 3) 2)
== ; DrRacket replaces a with 3 in ff's body
(+ (* 10 3) 2)
== ; now DrRacket uses the laws of arithmetic
(+ 30 2)
==
32
```

Naturally, we can reuse the result of this computation in others:

```
(* (ff 4) (+ (ff 3) 2))
== ; DrRacket substitutes 4 for a in ff's body
(* (* 10 4) (+ (ff 3) 2))
== ; DrRacket knows that (* 10 4) == 40
(* 40 (+ (ff 3) 2))
== ; now it uses the result of the above calculation
(* 40 32)
==
1280 ; because it is really just math
```

In sum, DrRacket is an incredibly fast algebra student; it knows all the laws of arithmetic and it is great at substitution. Even better, DrRacket cannot only determine the value of an expression; it can also show you **how** it does it. That is, it can show you step-by-step how to solve these algebra problems that ask you to determine the value of an expression.

Take a second look at the buttons that come with DrRacket. One of them looks like an "advance to next track" button on an audio player. If you click this button, the **stepper** window pops up and you can step through the evaluation of the program in the definitions area.

Enter the definition of `ff` into the definitions area. Add `(ff (+ 1 1))` at the bottom. Now click the *STEP*. The stepper window will show up; figure 11 shows what it looks like in version 6.2 of the software. At this point, you can use the forward and backward arrows to see all the computation steps that DrRacket uses to determine the value of an expression. Watch how the stepper performs the same calculations as we do.

Stop! Yes, you could have used DrRacket to solve some of your algebra homework. Experiment with the various options that the stepper offers.

Figure 11: The DrRacket stepper

Exercise 21. Use DrRacket's stepper to evaluate `(ff (ff 1))` step-by-step. Also try `(+ (ff 1) (ff 1))`. Does DrRacket's stepper reuse the results of computations? ∎

At this point, you might think that you are back in an algebra course with all these computations involving uninteresting functions and numbers. Fortunately, this approach generalizes to **all** programs, including the interesting ones, in this book.

Let's start by looking at functions that process strings. Recall some of the laws of string arithmetic:

```
(string-append "hello" " " "world") == "hello world"
(string-append "bye" ", " "world") == "bye, world"
...
```

Now suppose we define a function that creates the opening of a letter:

```
(define (opening first-name last-name)
  (string-append "Dear " first-name ","))
```

When you apply this function to two strings, you get a letter opening:

```
> (opening "Matthew" "Fisler")
"Dear Matthew,"
```

More importantly, though, the laws of computing explain how DrRacket determines this result and how you can anticipate what DrRacket does:

```
(opening "Matthew" "Fisler")
==  ; DrRacket substitutes "Matthew" for first-name
(string-append "Dear " "Matthew" ",")
==
"Dear Matthew,"
```

*Eventually you will encounter imperative operations, which do not combine or extract values but **modify** them. To calculate with such operations, you will need to add some laws to those of arithmetic and substitution.*

Since `last-name` does not occur in the definition of `opening`, replacing it with `"Fisler"` has no effect.

The rest of the book introduces more forms of data. To explain operations on data, we always use laws like those of arithmetic in this book.

Exercise 22. Use DrRacket's stepper on this program fragment:

```
(define (distance-to-origin x y)
  (sqrt (+ (sqr x) (sqr y))))
(distance-to-origin 3 4)
```

Does the explanation match your intuition? ∎

Exercise 23. The first 1String in `"hello world"` is `"h"`. How does the following function compute this result?

```
(define (string-first s)
  (substring s 0 1))
```

Use the stepper to confirm your ideas. ∎

Exercise 24. Here is the definition of ==>: y

```
(define (==> x y)
  (or (not x) y))
```

Use the stepper to determine the value of `(==> #true #false)`. ∎

Exercise 25. Take a look at this attempt to solve exercise 17:

```
(define (image-classify img)
  (cond
    [(>= (image-height img) (image-width img)) "tall"]
    [(= (image-height img) (image-width img)) "square"]
    [(<= (image-height img) (image-width img)) "wide"]))
```

Does stepping through an application suggest a fix? ∎

The advantage of following this slogan is that you get reasonably small functions, each of which is easy to comprehend and whose composition is easy to understand. Once you learn to design functions, you will recognize that getting small functions to work correctly is much easier than doing so with large ones. Better yet, if you ever need to change a part of the program due to some change to the problem statement, it tends to be much easier to find the relevant parts when it is organized as a collection of small functions as opposed to a large, monolithic block.

Here is a small illustration of this point with a sample problem:

> **Sample Problem** The owner of a monopolistic movie theater in a small town has complete freedom in setting ticket prices. The more he charges, the fewer people can afford tickets. The less he charges, the more it costs to run a show because attendance goes up. In a recent experiment the owner determined a relationship between the price of a ticket and average attendance.
>
> At a price of $5.00 per ticket, 120 people attend a performance. For each 10-cent change in the ticket price, the average attendance changes by 15 people. That is, if the owner charges $5.10, some 105 people attend on the average; if the price goes down to $4.90, average attendance increases to 135. Let's translate this idea into a mathematical formula:
>
> $$\textit{avg. attendance} = 120\ \textit{people} - \frac{\$(\textit{change in price})}{\$0.10} \cdot 15\ \textit{people}$$
>
> Stop! Explain the minus sign before you proceed.
>
> Unfortunately, the increased attendance also comes at an increased cost. Every performance comes at a fixed cost of $180 to the owner plus a variable cost of $0.04 per attendee.
>
> The owner would like to know the exact relationship between profit and ticket price in order to maximize the profit.

While the task is clear, how to go about it is not. All we can say at this point is that several quantities depend on each other.

When we are confronted with such a situation, it is best to tease out the various dependencies, one by one:

1. The problem statement specifies how the number of attendees depends on the ticket price. Computing this number is clearly a separate task and thus deserves its own function definition:

```
(define (attendees ticket-price)
  (- 120 (* (- ticket-price 5.0) (/ 15 0.1))))
```

2. The *revenue* is exclusively generated by the sale of tickets, meaning it is exactly the product of ticket price and number of attendees:

```
(define (revenue ticket-price)
  (* ticket-price (attendees ticket-price)))
```

3. The *cost* consists of two parts: a fixed part ($180) and a variable part that depends on the number of attendees. Given that the number of attendees is a function of the ticket price, a function for computing the cost of a show must also consume the ticket price so that it can reuse the `attendees` function:

```
(define (cost ticket-price)
  (+ 180 (* 0.04 (attendees ticket-price))))
```

4. Finally, *profit* is the difference between revenue and costs for some given ticket price:

```
(define (profit ticket-price)
  (- (revenue ticket-price)
     (cost ticket-price)))
```

   The BSL definition of `profit` directly follows the suggestion of the informal problem description.

These four functions are all there is to the computation of the profit, and we can now use the profit function to determine a good ticket price.

Exercise 27. Our solution to the sample problem contains several constants in the middle of functions. As "One Program, Many Definitions" already points out, it is best to give names to such constants so that future readers understand where these numbers come from. Collect all definitions in DrRacket's definitions area and change them so that all magic numbers are refactored into constant definitions. ▮

Exercise 28. Determine the potential profit for these ticket prices: $1, $2, $3, $4, and $5. Which price maximizes the profit of the movie theater? Determine the best ticket price to a dime. ▮

Here is an alternative version of the same program, given as a single function definition:

```
(define (profit price)
  (- (* (+ 120
           (* (/ 15 0.1)
              (- 5.0 price)))
        price)
     (+ 180
        (* 0.04
           (+ 120
              (* (/ 15 0.1)
                 (- 5.0 price)))))))
```

Enter this definition into DrRacket and ensure that it produces the same results as the original version for $1, $2, $3, $4, and $5. A single look should suffice to show how much more difficult it is to comprehend this one function compared to the above four.

Exercise 29. After studying the costs of a show, the owner discovered several ways of lowering the cost. As a result of these improvements, there is no longer a fixed cost; a variable cost of $1.50 per attendee remains.

Modify both programs to reflect this change. When the programs are modified, test them again with ticket prices of $3, $4, and $5 and compare the results. ∎

2.4 Global Constants

As the Prologue already says, functions such as `profit` benefit from the use of global constants. Every programming language allows programmers to define constants. In BSL, such a definition has the following shape:

- write "(`define` ",
- write down the name,
- followed by a space and an expression, and
- write down ")".

The name of a constant is a *global variable* while the definition is called a *constant definition*. We tend to call the expression in a constant definition the *right-hand side* of the definition.

Constant definitions introduce names for all forms of data: numbers, images, strings, and so on. Here are some simple examples:

```
; the current price of a movie ticket:
(define CURRENT-PRICE 5)

; useful to compute the area of a disk:
(define ALMOST-PI 3.14)

; a blank line:
(define NL "\n")

; an empty scene:
(define MT (empty-scene 100 100))
```

The first two are numeric constants, the last two are a string and an image. By convention, we use uppercase letters for global constants because it ensures that no matter how large the program is, the readers of our programs can easily distinguish such variables from others.

All functions in a program may refer to these global variables. A reference to a variable is just like using the corresponding constants. The advantage of using variable names instead of constants is that a single edit of a constant definition affects all uses. For example, we may wish to add digits to `ALMOST-PI` or enlarge an empty scene:

```
(define ALMOST-PI 3.14159)

; an empty scene:
(define MT (empty-scene 200 800))
```

Most of our sample definitions employ *literal constants* on the right-hand side, but the last one uses an expression. And indeed, a programmer can use arbitrary expressions to compute constants. Suppose a program needs to deal with an image of some size and its center:

```
(define WIDTH 100)
(define HEIGHT 200)

(define MID-WIDTH (/ WIDTH 2))
(define MID-HEIGHT (/ HEIGHT 2))
```

It can use two definitions with literal constants on the right-hand side and two *computed constants*, that is, variables whose values are not just literal constants but the results of computing the value of an expression.

Again, we state an imperative slogan:

> *For every constant mentioned in a problem statement, introduce one constant definition.*

Exercise 30. Define constants for the price optimization program at the movie theater so that the price sensitivity of attendance (15 people for every 10 cents) becomes a computed constant. ∎

2.5 Programs

You are ready to create simple programs. From a coding perspective, a program is just a bunch of function and constant definitions. Usually one function is singled out as the "main" function, and this main function tends to compose others. From the perspective of launching a program, however, there are two distinct kinds:

- a *batch program* consumes all of its inputs at once and computes its result. Its main function is the composition of auxiliary functions, which may refer to additional auxiliary functions, and so on. When we launch a batch program, the operating system calls the main function on its inputs and waits for the program's output.

- an *interactive program* consumes some of its inputs, computes, produces some output, consumes more input, and so on. When an input shows up, we speak of an *event*, and we create interactive programs as *event-driven* programs. The main function of such an event-driven program uses an expression to describe which functions to call for which kinds of events. These functions are called *event handlers*.

  When we launch an interactive program, the main function informs the operating system of this description. As soon as input events happen, the operating system calls the matching event handler. Similarly, the operating system knows from the description when and how to present the results of these function calls as output.

This book focuses mostly on programs that interact via *graphical user interfaces (GUI)*; there are other kinds of interactive programs, and you will get to know those as you continue to study computer science.

Batch Programs As mentioned, a batch program consumes all of its inputs at once and computes the result from these inputs. Its main function

We call the main function `convert`. It consumes two file names: `in` for the file where the Fahrenheit temperature is found and `out` for where we want the Celsius result. A composition of five functions computes `convert`'s result. Let's step through `convert`'s body carefully:

1. `(read-file in)` retrieves the content of the named file as a string;
2. `string->number` turns this string into a number;
3. `C` interprets the number as a Fahrenheit temperature and converts it into a Celsius temperature;
4. `number->string` consumes this Celsius temperature and turns it into a string; and
5. `(write-file out ...)` places this string into the file named `out`.

This long list of steps might look overwhelming, and it doesn't even include the `string-append` part. Stop! Explain

```
(string-append ... "\n")
```

In contrast, the average function composition in a pre-algebra course involves two functions, possibly three. Keep in mind, though, that programs accomplish a real-world purpose while exercises in algebra merely illustrate the idea of function composition.

At this point, we can experiment with `convert`. To start with, we use `write-file` to create an input file for `convert`:

You can also create `"sample.dat"` with a file editor.

```
> (write-file "sample.dat" "212")
"sample.dat"
> (convert "sample.dat" 'stdout)
100
'stdout
> (convert "sample.dat" "out.dat")
"out.dat"
> (read-file "out.dat")
"100"
```

For the first interaction, we use `'stdout` so that we can view what `convert` outputs in DrRacket's interactions area. For the second one, `convert` is given the name `"out.dat"`. As expected, the call to `convert` returns

this string; from the description of `write-file` we also know that it deposited a Fahrenheit temperature in the file. Here we read the content of this file with `read-file`, but you could also view it with a text editor.

In addition to running the batch program, it is also instructive to step through the computation. Make sure that the file `"sample.dat"` exists and contains just a number, then click the *STEP* button in DrRacket. Doing so opens another window in which you can peruse the computational process that the call to the main function of a batch program triggers. You will see that the process follows the above outline.

Exercise 31. Recall the `letter` program from chapter 2.3. Here is how to launch the program and have it write its output to the interactions area:

```
> (write-file
   'stdout
   (letter "Matthew" "Fisler" "Felleisen"))
Dear Matthew,

We have discovered that all people with the
last name Fisler have won our lottery. So,
Matthew, hurry and pick up your prize.

Sincerely,

Felleisen
'stdout
```

Of course, programs are useful because you can launch them for many different inputs. Run `letter` on three inputs of your choice.

Here is a letter-writing batch program that reads names from three files and writes a letter to one:

```
(define (main in-fst in-lst in-signature out)
  (write-file out
              (letter (read-file in-fst)
                      (read-file in-lst)
                      (read-file in-signature))))
```

The function consumes four strings: the first three are the names of input files and the last one serves as an output file. It uses the first three to read one string each from the three named files, hands these strings to `letter`,

and eventually writes the result of this function call into the file named by `out`, the fourth argument to `main`.

Create appropriate files, launch `main`, and check whether it delivers the expected letter in a given file. ∎

Interactive Programs Batch programs are a staple of business uses of computers, but the programs people encounter now are interactive. In this day and age, people mostly interact with desktop applications via a keyboard and a mouse. Furthermore, interactive programs can also react to computer-generated events, for example, clock ticks or the arrival of a message from some other computer.

Exercise 32. Most people no longer use desktop computers just to run applications but also employ cell phones, tablets, and their cars' information control screen. Soon people will use wearable computers in the form of intelligent glasses, clothes, and sports gear. In the somewhat more distant future, people may come with built-in bio computers that directly interact with body functions. Think of ten different forms of events that software applications on such computers will have to deal with. ∎

The purpose of this section is to introduce the mechanics of writing **interactive** BSL programs. Because many of the project-style examples in this book are interactive programs, we introduce the ideas slowly and carefully. You may wish to return to this section when you tackle some of the interactive programming projects; a second or third reading may clarify some of the advanced aspects of the mechanics.

By itself, a raw computer is a useless piece of physical equipment. It is called *hardware* because you can touch it. This equipment becomes useful once you install *software*, that is, a suite of programs. Usually the first piece of software to be installed on a computer is an *operating system*. It has the task of managing the computer for you, including connected devices such as the monitor, the keyboard, the mouse, the speakers, and so on. The way it works is that when a user presses a key on the keyboard, the operating system runs a function that processes keystrokes. We say that the keystroke is a *key event*, and the function is an *event handler*. In the same vein, the operating system runs an event handler for clock ticks, for mouse actions, and so on. Conversely, after an event handler is done with its work, the operating system may have to change the image on the screen, ring a bell, print a document, or perform a similar action. To accomplish these tasks, it also runs functions that translate the operating system's data into sounds, images, actions on the printer, and so on.

Naturally, different programs have different needs. One program may interpret keystrokes as signals to control a nuclear reactor; another passes

them to a word processor. To make a general-purpose computer work on these radically different tasks, different programs install different event handlers. That is, a rocket-launching program uses one kind of function to deal with clock ticks while an oven's software uses a different kind.

Designing an interactive program requires a way to designate some function as the one that takes care of keyboard events, another function for dealing with clock ticks, a third one for presenting some data as an image, and so forth. It is the task of an interactive program's main function to communicate these designations to the operating system, that is, the software platform on which the program is launched.

DrRacket is a small operating system, and BSL is one of its programming languages. The latter comes with the *2htdp/universe* library, which provides `big-bang`, a mechanism for telling the operating system which function deals with which event. In addition, `big-bang` keeps track of the *state of the program*. To this end, it comes with one required sub-expression, whose value becomes the *initial state* of the program. Otherwise `big-bang` consists of one required clause and many optional clauses. The required `to-draw` clause tells DrRacket how to render the state of the program, including the initial one. Each of the other, optional clauses tells the operating system that a certain function takes care of a certain event. Taking care of an event in BSL means that the function consumes the state of the program and a description of the event, and that it produces the next state of the program. We therefore speak of the *current state* of the program.

Terminology In a sense, a `big-bang` expression describes how a program connects with a small segment of the world. This world might be a game that the program's users play, an animation that the user watches, or a text editor that the user employs to manipulate some notes. Programming language researchers therefore often say that `big-bang` is a description of a small world: its initial state, how states are transformed, how states are rendered, and how `big-bang` may determine other attributes of the current state. In this spirit, we also speak of the *state of the world* and even call `big-bang` programs *world programs*. **End**

Let's study this idea step-by-step, starting with this definition:

```
(define (number->square s)
  (square s "solid" "red"))
```

The function consumes a positive number and produces a solid red square of that size. After clicking *RUN*, experiment with the function, like this:

```
> (number->square 5)
■
> (number->square 10)
■
> (number->square 20)
■
```

It behaves like a batch program, consuming a number and producing an image, which DrRacket renders for you.

Now try the following `big-bang` expression in the interactions area:

```
> (big-bang 100 [to-draw number->square])
```

A separate window appears, and it displays a 100×100 red square. In addition, the DrRacket interactions area does not display another prompt; it is as if the program keeps running, and this is indeed the case. To stop the program, click on DrRacket's *STOP* button or the window's *CLOSE* button:

```
> (big-bang 100 [to-draw number->square])
100
```

When DrRacket stops the evaluation of a `big-bang` expression, it returns the current state, which in this case is just the initial state: `100`.

Here is a more interesting `big-bang` expression:

```
> (big-bang 100
    [to-draw number->square]
    [on-tick sub1]
    [stop-when zero?])
```

This `big-bang` expression adds two optional clauses to the previous one: the `on-tick` clause tells DrRacket how to deal with clock ticks and the `stop-when` clause says when to stop the program. We read it as follows, starting with `100` as the initial state:

1. every time the clock ticks, subtract `1` from the current state;
2. then check whether `zero?` is true of the new state and if so, stop; and
3. every time an event handler returns a value, use `number->square` to render it as an image.