

v.6.3.0.2

How to Design Programs, Second Edition

Please send reports about mistakes to matthias @ ccs.neu.edu

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi

© 1 August 2014 [MIT Press](#) This material is copyrighted and provided under the Creative Commons [CC BY-NC-ND](#) license [[interpretation](#)].

Draft Release

This document is the **draft release** of HtDP/2e. It is updated on a frequent basis. We recommend the [stable version](#) for teaching.

Released on Saturday, October 31st, 2015 9:53:45am

v.6.3.0.2

Preface

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

Yet *good programming* is much more than a vocational skill. In contrast to programming—often taught with a “tinker until it works” approach—*good programming* emphasizes systematic thought, planning, and understanding. Good programming satisfies an aesthetic sense of accomplishment. Good programming is also critical for professionals who maintain programs over a long period.

Programming differs from good programming like crayon sketches in a diner from oil paintings in a museum. To emphasize this difference, we refer to the latter as *program design*. Don’t let the words “painting” and “museum” scare you. We wouldn’t have spent fifteen years writing this book, if we didn’t believe that *everyone can design programs* and *everyone can experience the satisfaction that comes with it*. Indeed, we go as far as saying that program design—not programming—deserves the same role in a liberal-arts education as mathematics and English:

everyone should learn how to design programs.

Even if you never design a program again, you will experience the joy of a creator, you will acquire a new sense of aesthetic, and you will pick up universally useful skills.

This Book

The purpose of this book is to introduce novice programmers to the *systematic design of programs*. It also presents a *symbolic view of computation*, which explains the process of running a program via simple manipulations of its text. As such, the book de-emphasizes the study of programming language details, the allusions to these strange things called stacks and heaps, the analysis of algorithmic minutiae, and the usual (mathematical) puzzles that substitute for programming knowledge in a typical first course.

Our design concepts draw on Michael A. Jackson’s method for creating COBOL programs and conversations with Daniel P. Friedman on recursion, Robert Harper on type theory, and Daniel Jackson on software design.

“Systematic program design” refers to a process that takes a complete novice from a problem statement to a well-organized solution in a step-by-step fashion. Each step produces a well-defined intermediate product. When the novice is stuck, an instructor can inspect these intermediate products to diagnose your progress and recommend corrective actions—without any reference to the specific programming problem.

Software support for the book includes a pedagogical programming environment and a

series of programming languages tailored to the book's design concepts. Initially, the environment roughly corresponds to the world of a so-called pre-algebra course. Both the environment and the language grow with the design concepts. This staging allows the environment to provide tailored feedback in terms of the concepts covered so far—in stark contrast to the usual approach of using professional environments, which only overwhelm beginners.

Designing programs in this context means that you, the reader, will acquire two kinds of skills. On one hand, program design 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. Furthermore, designing programs produces fun and useful things, which vastly increases the sense of accomplishment when compared to drill exercises in mathematics texts. On the other hand, program design teaches the same analytical reading and writing skills as English. Even the smallest design tasks are formulated as word problems. Without critical reading skills, you cannot design programs that solve the problem and match its specifications. Conversely, program design methods force you to articulate their thoughts in proper and precise English. Indeed, our observations suggests that if you truly absorb the design recipe, you will develop your articulation skills more than anything else.

Systematic Design

A program interacts with people, whom computer scientists call *users*. Some programs request all their inputs as soon as they are launched and then compute an answer without any further interaction with the user. Others request some input, produce an output, prompt users for more input, and so on. And, different programs use different devices to absorb inputs or deliver outputs.

Regardless of its mode of interaction, any useful program is likely to consist of many building blocks because it is difficult to construct a reliable artifact in one piece. The key is to discover which building blocks are needed, how to connect them, and how to build blocks when needed. We choose to call this activity *design* because its dictionary meaning matches this description. In support of the design activity, this book introduces two kinds of guidelines: recipes and refinement.

1. Problem Analysis

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 which data the desired function consumes and produces. Articulate what the function computes as a concise one-line statement. 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 your 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. The tests will help your successor ensure that the function works for these examples after future modifications.

Figure 1: The basic steps of a program design recipe

Design recipes come at two levels: for programs and for functions, which in our world are the basic building blocks of programs. Two recipes are about complete programs: one for graphical interactive programs and one for batch programs. All others are about the construction of individual functions with a focus on so-called *structural design*. This latter recipe deals with the design of functions that operate on structured data and whose organization reflects the structure of the data. It is best to think of this recipe as a grid with two dimensions. One dimension lists the steps of the design process; the other “measures” increasingly complex forms of data.

Figure 1 displays the six steps of the structural design process. Each step states the expected outcomes and some activities. Examples play a central role in the process. For the chosen data representation in step 1, writing down examples proves that you know how “real world” information is encoded as data and how data is interpreted as information. The request for functional examples in step 3 forces you to work through concrete scenarios and thus helps you understand what the function is expected to compute. In step 6, you turn examples into tests, which ensures that the function works properly for some cases.

Instructors Have students copy figure 1 on one side of an index card. When students are stuck, ask them to prove that they are card-carrying members of the design club. Once they produce the card, point them to the step where they are stuck.

While structural design is most of what programmers end up practicing in the real world, they also spend a good amount of time abstracting (“refactoring” in modern parlance) code or figuring out how to re-use existing abstractions. Abstracting means unifying similar program fragments into a single element and reusing this element in place of the original fragments; many languages already come with extremely powerful abstractions so that programmers don’t have to re-do basic work over and over again. Hence the book offers design recipes on creating abstractions and using existing abstractions.

The rest of the book expands and revises the structural design recipes in different directions.

Even this first look at the design recipes ought to clarify why we claim that design guidelines make sure you never really get stuck. Equipped with the design recipes, you don’t have to stare at a blank screen and wait for an idea to show up. You always have a concrete goal, an intermediate product to create. As a matter of fact, the design recipes provide guidance for each step in the form of pointed questions. The answers to these questions help create the intermediate product in an incremental fashion. Our students quickly discover that we ask the same questions over and over again, and that they can ask these questions on their own. They thus learn to help themselves. And

Instructors Tell students to write down the questions for the creation of structural templates and functions on the back of their index card.

you can do so, too.

With this explanation of design recipes and design processes, you can also see how the teaching of program design instills skills that are important in a variety of professions. To design a program properly, you must:

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

Each step requires analysis, precision, description, focus on and attention to specific details. Any experienced businessman, engineer, journalist, lawyer, scientist, or any other professional can tell you how many of these skills are necessary for his or her daily work. Practicing program design is one way to prepare yourself for these professions.

Iterative Refinement addresses the problem of designing complex programs. When a program addresses many concerns and consists of many functions, getting everything right at once is nearly impossible. In response, this book introduces the design concept of iterative refinement, which calls for creating complete programs in discrete layers. Its inner core is the essence of the desired functionality; additional refinement steps improve the functionality of the core and add additional functionality.

In this sense a programmer is a mini-scientist who creates approximate models of the desired product, translates them into first designs, evaluates them with users, and iteratively refines them until the product closely matches the initial idea. Indeed, the best programmers rewrite and refine their programs many times until they meet the desired standard.

Refining and reworking designs is not restricted to computer science and program creation. Architects, composers, writers and all professionals do it, too. They start with ideas in their head and somehow articulate their essence. They refine it on paper until their product reflects their mental image as much as possible. As they bring their ideas to paper, they employ basic 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. Conversely, they are productive with an iterative development process because they have absorbed basic skills and they have learned which "design recipe" to choose for the situation at hand.

DrRacket and the Teaching Languages

People often tell us that they wish to learn how to program and then they ask which programming language they should learn (first). Given the press that some programming languages get, this question is not surprising. But it is also inappropriate.

As the preceding sections explain, learning to program well is primarily about studying principles of design. The ideal programming language must support these principles *and* its implementation must explain all the mistakes novices make in terms that they understand. Our research shows that no off-the-shelf industrial language satisfies the second constraint. Their creators expect that the programmers immediately know the *entire* language because they have professional programmers in mind who roughly understand the fundamental

concepts of languages.

Our programming language ensures that a reader who works through the first part of this book will also master the concepts of pre-algebra.

Our solution is to use our own, incredibly small, teaching language, dubbed ***SL** in this preface. The language includes notation for function definitions, function applications, and nested, potentially conditional expressions. In essence, it is the foreign language that students in pre-algebra courses acquire. The difference is that our language comes with many more kinds of data than algebra—in addition to numbers, its basic data includes images and words. The language also comes with a number of pre-defined functions that make working in “pre-algebra” fun and entertaining; the very first program is a small animation.

When it comes to programming environments, we face an equally bad choice. A programming environment for professionals is the rough analogue of 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 novice programmers.

DrRacket consists of two simple panes: a Definitions area, where you define functions, and an Interactions area, where you interact with them. Even when there is nothing in the Definitions area, you can immediately experiment with expressions in the Interactions area. Thus, experimenting starts with pocket-calculator arithmetic, and it proceeds from there to calculations with images, words, lists, and other forms of data. This way, learning is closely supported by a highly playful, feedback-oriented support system.

An interactive evaluator simplifies the programming language and the learning process. Concretely, DrRacket has two advantages over conventional programming environments. First, it enables novice programmers to manipulate data directly. Hence the language does not need facilities for reading input and writing output, and 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. Nowadays this separation is considered so fundamental to the construction of software that it has its own name: called model-view-controller architecture. In short, working in DrRacket ensures that you pick up fundamental software engineering ideas in a natural way.

The Parts of the Book

The book consists of a prologue, six parts, five intermezzos, and an epilogue. While the parts focus on program design, the intermezzos introduce other topics concerning programming and computing. Here is a brief overview:

- **Prologue: How to Program** is a quick introduction to plain programming. It introduces everything needed to write a simple animation. Any novice to programming is bound to feel empowered, overwhelmed, or both. The final note explains that this is *not* how we should teach programming and that we really need to organize our thoughts in a better way than this prologue.
- **Fixed-Size Data** thoroughly explains the most basic mechanisms of program design and key concepts of computing. It starts with a look at arithmetic in our language, specifically the arithmetic of atomic data—numbers, words, images and so on—before it

introduces functions and programs. At that point, the book switches to concepts of systematic design of batch and interactive programs plus simple functions. The rest of this part expands the realm of data with intervals, enumerations, itemizations, structures, and various combinations of these.

- **Intermezzo: BSL** describes the teaching language in complete detail: its vocabulary, its grammar, and its meaning. Computer scientists refer to these as syntax and semantics.
- **Arbitrarily Large Data** extends **Fixed-Size Data** with the most interesting and useful form of data: arbitrarily large compound data. While a programmer may nest the kinds of data from **Fixed-Size Data** to represent information, the nesting is of a specific depth and breadth—meaning the piece of data is of a fixed size. This part shows how a subtle generalization gets us from there to data of arbitrary size and how to design functions that process this kind of data.
- **Intermezzo: Quote, Unquote** introduces a short-hand for writing down large pieces of data: quotation and anti-quotation. Although this notation is close to 60 years old, few appreciate its power and even fewer languages include such facilities.
- **Abstraction** acknowledges that many of the functions from **Arbitrarily Large Data** look alike. No programming language should force programmers to create and maintain pieces of code that are that similar to each other; conversely, every good programming language comes with some mechanism for unifying such similarities. Computer scientists call this *abstraction*, and they know that abstraction greatly increases a programmer’s productivity. Hence, this part introduces design recipes for creating abstractions and for using existing abstractions.
- **Intermezzo: Scope and Abstraction** plays two roles. First, it explains the important 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 a plain inspection of the program. Second, it introduces a library with additional mechanisms for abstraction, so-called *for loops* plus pattern matching. Both show how powerful the teaching languages actually are.
- **Intertwined Data** generalizes the story of **Arbitrarily Large Data** in several directions and also introduces the idea of iterative refinement into the catalog of design concepts.
- **Intermezzo: The Nature of Numbers** explains how “decimal” numbers really work in (all) programming languages. Every budding programmer ought to know these basic facts, and in the context of this book, the ideas are easy to discuss and illustrate.
- **Generative Recursion** injects a completely new design idea. While structural design and abstraction suffice for most problems that programmers encounter, they often lead to insufficiently “performant” programs. In other words, structurally designed programs might need too much time or memory 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, specifically which parts of the design remain systematic and which parts benefit from problem-specific insights.
- **Intermezzo: The Cost of Computation** uses the discussions from **Generative Recursion** to illustrate how computer scientists think about performance abstractly.
- **Accumulators** adds a final trick to the tool box 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 **Generative Recursion**, accumulators can make

the difference between finding an answer at all and never finding one.

[Epilogue: How Not to Program](#) is both an assessment and a look ahead to what's next.

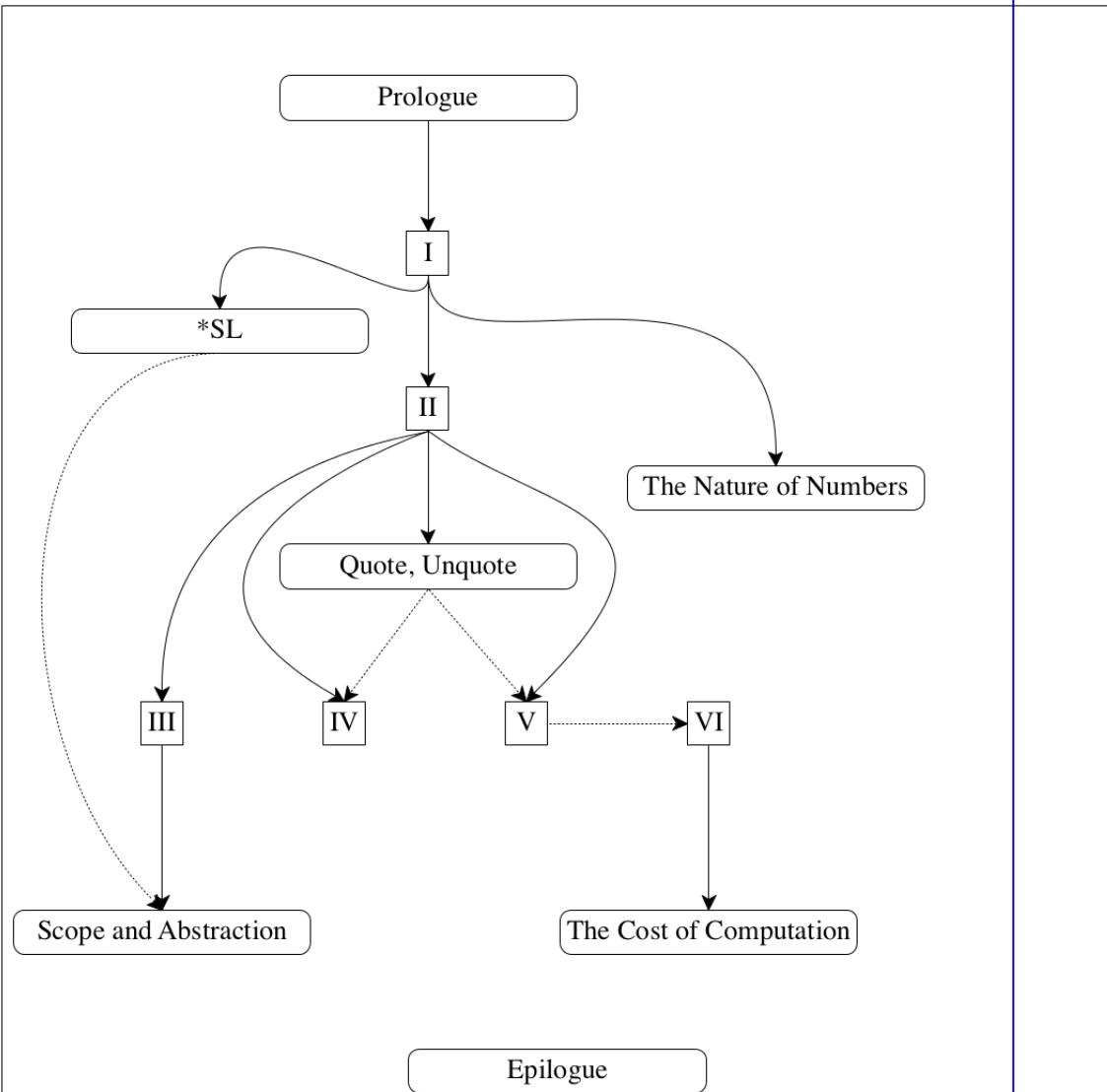


Figure 2: The dependencies among parts and intermezzos (draft)

To the Reader and the Instructor We would really like you to work through the entire book, from the first page to the last.

We would really like for instructors to cover as many pieces as possible, starting from Prologue all the way through the Epilogue. In our courses, we do so, and we make sure that, in the end, students create a sizable and entertaining program.

But, we understand that some circumstances call for significant cuts and that some instructors' tastes call for slightly different ways to interpret the book. For those, [figure 2](#) shows how the pieces depend on each other. Solid arrows suggest a mandatory sequence; dotted lines are mere suggestions. Here are three possible shortened paths through the book:

- A high-school instructor may want to cover (as much as possible of) parts [Fixed-Size Data](#) and [Arbitrarily Large Data](#), including a small project based on the design recipes of these two parts.
- A college instructor in a quarter system may wish to focus on parts [Fixed-Size Data](#), [Arbitrarily Large Data](#), [Abstraction](#), and [Generative Recursion](#), with 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 parts [Fixed-Size Data](#) and [Arbitrarily Large Data](#) and then those chapters and sections of part [Accumulators](#) that do not depend on [Generative Recursion](#). At that point, it is possible to discuss [Intermezzo: The Cost of Computation](#) and to return to the remaining pieces of the book as desired.

Reading the prologue first is a good idea regardless of the chosen path.

Iteration of Sample Topics The book revisits certain exercise and sample topics time and again. For example, pets are found all over [Fixed-Size Data](#) and even show up in [Arbitrarily Large Data](#). Similarly, both [Fixed-Size Data](#) and [Arbitrarily Large Data](#) cover alternative approaches to implementing an interactive text editor. Graphs appear in [Generative Recursion](#) and immediately again in [Accumulators](#). 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 sequences.

The Differences

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

1. The second edition explicitly acknowledges the difference between designing a program and designing a bunch of functions. In particular, this edition focuses on two kinds of programs: interactive and graphical programs and batch programs.
2. The design of a program proceeds in a top-down planning and a bottom-up construction fashion. 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, the second edition treats it as an explicit design element.
3. The design of each wish on the wish list relies on the function design recipe. As in the first edition, the six parts focus on structural design, compositional design, generative recursion, and designs with accumulators.
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 the intermediate data by a second function simplifies the overall design. Again, this kind of planning also creates 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.
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.

We thank Dr. Kathi Fisler for focusing our attention on this point.

5. This edition of the book drops the design of imperative programs. The old chapters remain available on-line. The material will be moved into 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 so-called `require` specifications, but it is still possible to add teachpacks via a menu in DrRacket.
3. Finally, this second edition differs from the first in terminology and notation:

Second Edition	First Edition
signature	contract
itemization	union
'()	<code>empty</code>
#true	<code>true</code>
#false	<code>false</code>

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’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 their courses and/or gave detailed comments on early drafts. We are grateful to them for their help and their patience: Ian Barland, John Clements, Bruce Duba, Mike Ernst, Kathi Fisler, Daniel P. Friedman, John Greiner, John Stone, Géraldine Morin, and Valdemar Tamez.

A dozen generations of *Comp 210* students at Rice University 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, Mr. Jack Clay, Dr. Richard Clemens, Mr. Kyle Gillette, Ms. Karen Buras, 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. Matthias 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 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 made significant comments on the various drafts and what kind of improvements they would like to see in a second one. We gratefully acknowledge the thoughtful conversations and exchanges with these individuals:

Kathi Fisler, Gregor Kiczales, Prabhakar Ragde, and Norman Ramsey.

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 second edition.

We also thank Ennas Abdussalam, Saad Bashir, Steven Belknap, Stephen Bloch, Elijah Botkin, Tomas Cabrera, Anthony Carrico, Rodolfo Carvalho, Estevo Castro, Stephen Chang, Nelson Chiu, Jack Clay, Richard Cleis, John Clements, Mark Engelberg, Andrew Fallows, Christopher Felleisen, Sebastian Felleisen, Vladimir Gajić, Adrian German, Jack Gitelson, Kyle Gillette, Scott Greene, Ryan Golbeck, Josh Grams, Jane Griscti, Tyler Hammond, Nan Halberg, Li Junsong, Nadeem Abdul Hamid, Jeremy Hanlon, Craig Holbrook, Wayne Iba, Jordan Johnson, Blake Johnson, Erwin Junge, Marc Kaufmann, Gregor Kiczales, Eugene Kohlbecker, Caitlin Kramer, Jackson Lawler, Saad Mahmood, Jay McCarthy, Mike McHugh, Wade McReynolds, Elena Machkasova, David Moses, Ann E. Moskol, Scott Newson, Paul Ojanen, Prof. Robert Ordóñez, Laurent Orseau, Klaus Ostermann, Sinan Pehlivanoğlu, Eric Parker, Nick Pleatsikas, Prathyush Pramod, Norman Ramsey, Krishnan Ravikumar, Jacob Rubin, Luis Sanjuán, Ryan “Havvy” Scheel, Lisa Scheuing, Willi Schiegel, Vinit Shah, Nick Shelley, Tubo Shi, Matthew Singer, Stephen Siegel, Kartik Singhal, Joe Snikeris, Marc Smith, Dave Smylie, Vincent St-Amour, Reed Stevens, Kevin Sullivan, Asumu Takikawa, Éric Tanter, Sam Tobin-Hochstadt, Thanos Tsouanas, Aaron Tsay, Mariska Twaalfhoven, Yuwang Yin, David Van Horn, Andre Venter, Jan Vitek, Mitchell Wand, Michael Wijaya, G. Clifford Williams, Ewan Whittaker-Walker, Julia Włochowski, Roelof Wobben Mardin Yadegar, and Huang Yichao for comments on previous drafts of this second edition.

The HTML layout is due to Matthew Butterick who created these styles for the Racket documentation.

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-line.

v.6.3.0.2

Prologue: How to Program

Consider a quick look at [On Teaching Part I](#).

When you were a small child, your parents probably taught you to count and later to 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.

Now you’re switching roles. You program, and the computer is a child. Of course, you start with the simplest of all calculations. You type

```
(+ 1 1)
```

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

Start DrRacket and select “Choose language” from the “Language” menu. This brings up a dialog listing “Teaching Languages” for “How to Design Programs” (and possibly other books). Choose “Beginning Student Language” and “OK” to set up DrRacket for this chapter.

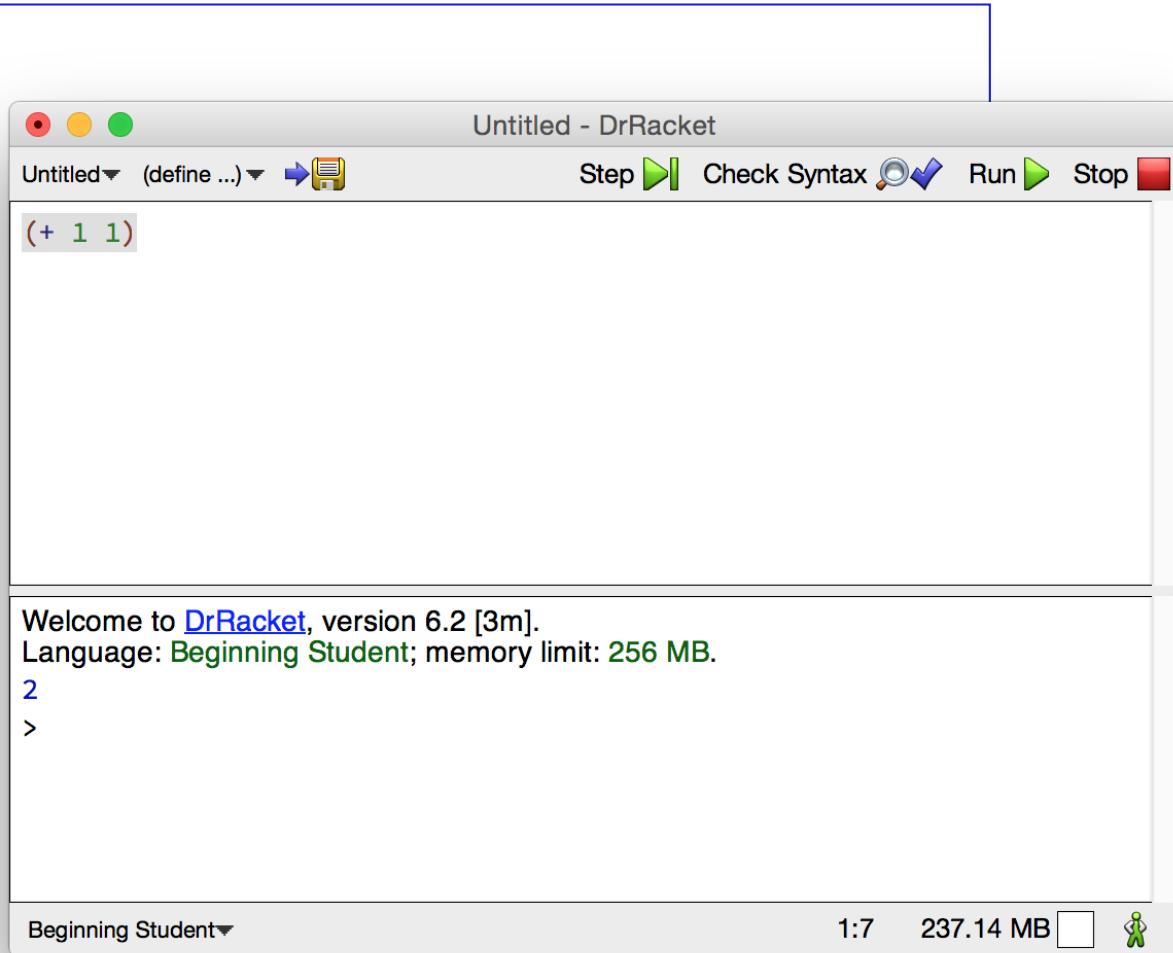


Figure 3: Meet DrRacket

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)
```

People often say the “computer does X for you” but in reality, a plain computer is pretty useless. It is **software** that computes.

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

Terminology At this point, we slow down for a moment and introduce some terms:

- 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 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 *inexact number*. Unlike your calculator or most other programming systems, DrRacket is extremely honest. When it doesn’t know the exact number,

it warns you with this special prefix. Later, we shall show you really strange facts about “computer numbers,” and you will then truly appreciate that DrRacket issues such warnings.

Enough terminology for now.

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
```

The first one is *nested arithmetic*, as you know it from school. The second one is *Racket arithmetic* and it is natural, if you always use parentheses to group operations and numbers together. This is as a good a time as any to discuss the nature of our notation—dubbed *Beginning Student Language* or just *BSL*—something you might have pondered for a while now.

In BSL, every time you want to use a “calculator operation,” you write down an opening parenthesis followed by the operation, the numbers on which the operation should work (separated by spaces or even line breaks), and ended by 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
```

This book does not teach you Racket, even if the software is called DrRacket. Instead it uses a series of teaching languages created for learning design principles. Once you have mastered these languages, you can quickly learn to program in all kinds of programming languages, including Racket and also JavaScript, Python, Ruby, and others.

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 to it.

The price for DrRacket’s convenience is that parentheses have meaning. You, the programmer, 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, but 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. Second, when you will read programs that others write—which you will do for about half the time of your programmer life—you will never have to wonder which expressions are evaluated first. The parentheses and the nesting will immediately tell you so.

Or you place the cursor next to the operation and hit F1. This action opens DrRacket's HelpDesk and searches for the documentation of the operation. Use the results concerning the HtDP teaching languages. As you may have noticed by now, this text is also 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 more.

Here are three programs that deal with text:

Just kidding: mathematics is a fascinating subject, but you knew that. You won't need too much of it for now, though if you want to be a really great programmer, you will need to study some.

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

After you click *RUN*, DrRacket displays three results: "helloworld" "hello world" "hello world"

To understand exactly what's going on, you first need to know that in BSL, text is any sequence of keyboard characters enclosed in double-quotes (""). Technically, this is called a string. Thus, "hello world", is a perfectly fine string and, when DrRacket evaluates this string, it displays it in the interactions area, just like a number. Indeed, many people's first program is one that displays the words "hello" and "world"—you wrote three of them already but the simplest one is to type in the string by itself:

```
"hello world"
```

Click *RUN* and admire the output of the program.

Otherwise, you need to know that in addition to an arithmetic of numbers, DrRacket also knows about an arithmetic of strings. Thus, `string-append` is an operation just like `+`; it makes a string by adding the second to the end of the first. As the first line 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 and third line show. 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 deal with as many operands as you wish.

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 it up in HelpDesk.

If you did look up the primitive operations of BSL, you saw that *primitive* (sometimes called *pre-defined* or *built-in*) operations can consume strings and produce numbers. You therefore can, if you so desire, add the length of a string to 20:

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

Use F1 or the drop-down menu on the right to open HelpDesk, look at the manuals for HtDP languages (Beginning Student Language) and its section on pre-defined operations. It lists all the operations and especially those that work on strings.

and DrRacket evaluates this expression like any other one. That is, an arithmetic operation doesn't have to be about just numbers or just strings. Many of them mix and match as needed. And then there are operations that convert strings into numbers and numbers into strings and you get what you expect:

```
| > (number->string 42)
"42"
| > (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 second expression raises a question, though. What if `string->number` isn't used with a string that is a number wrapped in string quotes? In that case the operation produces a totally 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`.)

Although it isn't possible to convert one number into a Boolean, it is certainly useful to

“convert” two numbers into a Boolean:

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

Guess what the results are before you move on:

```
#true
#true
#false
```

Now try these: `(>= 10 10)`, `(<= -1 0)`, and `(string=? "design" "tinker")`. This last one is totally 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.

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



>

To insert images such as this rocket into DrRacket, use the **Insert** menu and select the ``Insert image ...'' item. Or, if you’re reading this book on-line, copy-and-paste the image from your browser into DrRacket.

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.

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

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

`(require 2htdp/image)` specifies that you wish to add the definitions of the *2htdp/image* library to your program. Alternatively, use the “Language” drop-down menu, choose ``Add Teachpack ...” and pick *2htdp/image* from the *Preinstalled HtDP/2e Teachpack* menu.

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")

```

Best of all, DrRacket doesn't just draw images, because it really considers them values just like numbers. So naturally BSL has operations for combining images just like it has operations for adding numbers or appending strings:

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

```

Overlaying the same 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:

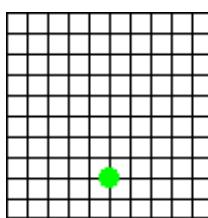
```
> (image-width (square 10 "solid" "red"))
10
> (image-width
  (overlay (rectangle 20 20 "solid" "blue")
           (circle 5 "solid" "red"))))
20
```

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

You should know about two more operations: `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 **downwards**, not upwards. 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. With BSL, your arithmetic is the arithmetic of numbers, strings, Boolean values, and even images. To compute though still means to determine the value of the expression(s) except that this value can be a string, a number, a Boolean, or an image.

And now you're basically 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.

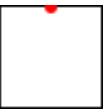
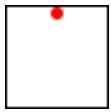
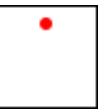
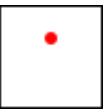
That's good news and bad news. It is good because programming and computing ought to be a natural generalization of using a calculator. It is bad because the purpose of programming is to deal with lots of data and to get lots of different results, with more or less the same calculations. (It should also 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. Remember those tables that your teachers would show you? Here is one more:

$$\begin{array}{cccccccccc} x = & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ y = & 1 & 4 & 9 & 16 & 25 & 36 & 49 & 64 & 81 & ? \end{array}$$

Your teachers would now ask you to fill in the blank, i.e., 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	5	6
$y =$						?

To be concrete, your teacher should ask you here to draw the sixth image, the seventh, and the 1234th 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 fifth, sixth, or seventh 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

$y =$ the image that contains a dot x^2 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 (usually) learn 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 shall use all kinds of names.

This second part means you must supply one value—a number—for x to determine a specific value for y . When you do, DrRacket plugs in 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 expression, 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)
```

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 in mathematics, too. Your teachers just forgot to tell you.

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 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. A function definition 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—most programmers call it 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. As seen, the expression involves the parameter, and it may also consist of many other expressions.

- Second,

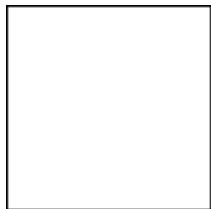
(FunctionName ArgumentExpression)

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 (software) “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 (number, string, image, Boolean) or a complex, 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 definitions area, like that:

```
(empty-scene 100 100)
```

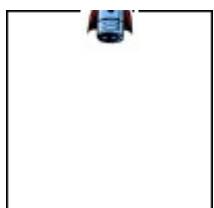
clicking *RUN* produces an empty rectangle, also called a scene:



You can add images to a scene with `place-image`:

```
(place-image  50 0 (empty-scene 100 100))
```

produces a scene with a rocket hovering near the center of the top:



Think of the rocket as an object that is like the dot—though more interesting—in the above table from your mathematics class.

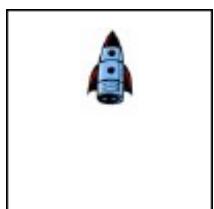
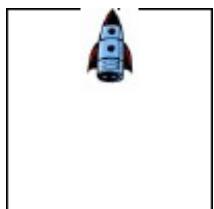
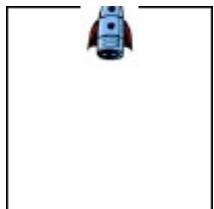
Next you should make the rocket descend, just like the dot in the above table. From the preceding section you know how to achieve this effect by increasing the y-coordinate that is supplied to `place-image`:

```
(place-image  50 10 (empty-scene 100 100))
```

```
(place-image  50 20 (empty-scene 100 100))

(place-image  50 30 (empty-scene 100 100))
```

Clicking *RUN* yields three scenes:



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

The first goal can be achieved with a function of course:

```
(define (create-rocket-scene height)
  
  (place-image 50 height (empty-scene 100 100)))
```

Yes, this is a function definition. Instead of *y*, it uses the name *create-rocket-scene*, 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 just 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:

```
(create-rocket-scene 0)
(create-rocket-scene 10)
(create-rocket-scene 20)
(create-rocket-scene 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

Now add the *2htdp/universe* library to your definitions.

expression:

```
| > (animate create-rocket-scene)
```

Stop for a moment 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 this 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 an interactions prompt. It opens another window—a *canvas*—and starts a clock that ticks 28 times per second. Every time the clock ticks, DrRacket applies `create-rocket-scene` 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 click *STOP* or 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, etc. and displays the resulting images. The long explanation is this:

[Exercise 303 explains how to design `animate`.](#)

- `animate` starts a clock, and `animate` counts the number of clock ticks;
- the clock ticks 28 times per second;
- every time the clock ticks, `animate` applies the function `create-rocket-scene` 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 1, then 2, then 3, etc., 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.

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 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 just functions.

Many Ways To Compute

When you run the `create-rocket-scene` program from the preceding section, the rocket eventually disappears in 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 `create-rocket-scene` 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 further.

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

that distinguish various situations:

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

This *sign* distinguishes three kinds of inputs: those numbers larger than 0, those equal to 0, and those smaller than 0. Depending on the input, the result of the function—or output as we may occasionally call it—is +1, 0, or -1.

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

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

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



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

To illustrate how `sign` works, we added three applications of the function to the definitions area. If you click *RUN*, you see 1, -1, and 0.

This is a good time to explore what the *STEP* button does. 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 `conditional` expression consists of many *conditional lines*. Each line contains two expressions: the left one is often called *condition* and the right one is called *result*. To evaluate a `cond` expression, DrRacket evaluates the first condition expression, *ConditionExpression1*. If this evaluation yields `#true`, DrRacket replaces the `cond` expression with the first result expression (*ResultExpression1*) and evaluates it. Whatever value DrRacket obtains is the result of the entire `cond` expression. If the evaluation of *ConditionExpression1* yields `#false`, DrRacket drops the first line and moves on to the second line, which is treated just like the first one. In case all condition expressions evaluate to `#false`, DrRacket signals an error.

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

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

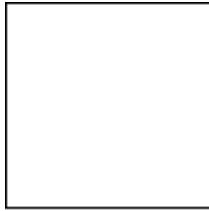
Figure 4: Landing a rocket (version 2)

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 100 scene. Since the `create-rocket-scene` function consumes the height at which it is to place the rocket in the scene, a simple test comparing the given height to the maximum height appears to suffice.

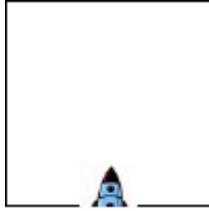
See [figure 4](#) for the revised function definition. We call the revised function `create-rocket-scene.v2` to distinguish it from the original version. Doing so also allows us to use both functions in the interactions area of DrRacket and to compare their results:

In BSL, you can really use all kinds of characters in function names, including “.” or “-” which you have already seen.

```
> (create-rocket-scene 5555)
```



```
> (create-rocket-scene.v2 5555)
```



No matter what number over `100` you give to `create-rocket-scene.v2`, you get the same scene. In particular, when you run the simulation

```
> (animate create-rocket-scene.v2)
```

the rocket descends, sinks half way into the ground, and finally comes to a halt.

Landing the rocket half-way under ground is ugly. Then again, you basically know how to fix this aspect of the program. As you learned from the preceding sections, DrRacket knows an arithmetic of images. Images have a center point and, when `place-image` adds an image to a scene, it uses this center point as if it were the image. This explains why the rocket is half way under ground at the end: DrRacket thinks of the image as if it were a point, but 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`, which is to images like `+` is to numbers. 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—also known as playing around—you can now figure out that

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



is the point at which you want the rocket to stop its descent. **Hint** You could figure this out by playing with the program directly. Or you can experiment in the interactions area with your image arithmetic. Enter this expression, which is one natural guess:

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



and then this one:

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




Which result do you like better?

```
(define (create-rocket-scene.v3 height)
  (cond
    [(<= height (- 100 (/ (image-height) 2)))
     (place-image 50 height (empty-scene 100 100))]
    [(> height (- 100 (/ (image-height) 2)))
     (place-image 50 (- 100 (/ (image-height) 2))
       (empty-scene 100 100))]))
```






Figure 5: Landing a rocket (version 3)

When you think and experiment along these lines, you eventually get to the program in figure 5. 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 imagine this. Your manager at your hot game company doesn't like the size of your program's canvas and requests a version that uses 200 by 400 scenes. This simple request forces you to replace 100 with 400 in five places in the program, which includes the `animate` line, and to replace 100 with 200 in two other places—not to speak of the occurrences of 50, which really suggest “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 real programs consists of 50,000 or 500,000 or 5,000,000 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 100)
```

in your program, you are saying that `HEIGHT` always represents the number `100`. The meaning of such a definition is what you expect. Whenever DrRacket encounters `HEIGHT` during its calculations, it uses `100` instead. Of course, you can also add

```
| (define WIDTH 100)
```

to your program to have one single place where you specify the width of the scene.

```
(define (create-rocket-scene.v4 h)
  (cond
    [(<= h (- HEIGHT (/ (image-height ) 2)))]
    [place-image  50 h (empty-scene WIDTH HEIGHT)])
    [(> h (- HEIGHT (/ (image-height ) 2)))]
    [place-image  50 (- HEIGHT (/ (image-height ) 2))
      (empty-scene WIDTH HEIGHT))]))
```

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

Figure 6: Landing a rocket (version 4)

Now take a look at the code in [figure 6](#), which implements this simple change. Copy the program into DrRacket, and after clicking *RUN*, evaluate the following interaction:

```
> (animate create-rocket-scene.v4)
```

Confirm that the program still functions as before.

The program in [figure 6](#) consists of three definitions: one function definition and two constant definitions. The number `100` occurs only twice: once as the value of `WIDTH` and once as the value of `HEIGHT`. The last line starts the simulation, just as in version 3 of the program. You may also have noticed that it uses `h` instead of `height` for the function parameter of `create-rocket-scene.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 create-rocket-scene.v4)`, it replaces `HEIGHT` with `100` and `WIDTH` with `100` every time it encounters these names. To experience the joys of real programmers, change the `100` next to `HEIGHT` into a `400` and click *RUN*. You see a rocket descending and landing in a `100` by `400` scene. One small change did it all.

In modern parlance, you have just experienced your first *program refactoring*. Every time you re-organize your program to prepare yourself for likely future change requests, you refactor your program. Put it on your resume. It sounds good, and your future employer probably enjoys reading such buzzwords, even if it doesn't make you a good programmer. What a good programmer would never live with, however, is that the program contains the same expression three times:

```

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

Every time your friends and colleagues read this program, they need to understand what this expression computes, namely, the distance between the top of the canvas and the center point of a rocket resting on the ground. Every time DrRacket computes the value of the expressions it has to perform three arithmetic operations: (1) determine the height of the image; (2) divide it by 2; and (3) subtract the result from HEIGHT. And every time it comes up with the same number.

This observation calls for the introduction of one more definition to your program:

```
(define ROCKET-CENTER-TO-TOP
```



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



plus the replacement of every other occurrence of `(- HEIGHT (/ (image-height) 2))` with `ROCKET-CENTER-TO-TOP`. You might wonder whether the new definition should be placed above or below the definition for HEIGHT. More generally, you should be wondering whether the ordering of definitions matters. The answer is that for constant definitions, the order matters, and for function definitions it doesn't. As soon as DrRacket encounters a constant definition, it determines the value of the expression and then associates the name with this value. Thus, for example,

```
(define HEIGHT (* 2 CENTER))
(define CENTER 100)
```

is meaningless because DrRacket hasn't seen the definition for CENTER yet when it encounters the definition for HEIGHT. In contrast,

```
(define CENTER 100)
(define HEIGHT (* 2 CENTER))
```

works as expected. First, DrRacket associates CENTER with 100. Second, it evaluates `(* 2 CENTER)`, which yields 200. Finally, DrRacket associates 200 with HEIGHT.

While the order of constant definitions matters, it doesn't matter whether you first define constants and then functions or vice versa. Indeed, if your program consisted of more than one function, it wouldn't matter in which order you defined those. For pragmatic reasons, it is good to introduce all constant definitions first, followed by the definitions of important functions, with the less important ones bringing up the rear guard. When you start writing your own multi-definition programs, you will soon see why this ordering is a good idea.

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

(define ROCKET
  (place-image rocket 50 100 MTSCN))

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

; functions
(define (create-rocket-scene.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 7: Landing a rocket (version 5)

Once you eliminate all repeated expressions, you get the program in [figure 7](#). It consists of one function definition and five constant definitions. Beyond the placement of the rocket's center, these constant definitions also factor out the image itself as well as the creation of the empty scene.

Before you read on, ponder the following changes to your program:

The program also contains two *line comments*, introduced with semi-colons (";"). While DrRacket ignores such comments, people who read programs should not because comments are intended for human readers. It is a "back channel" of communication between the author of the program and all of its future readers to convey information about the program.

- How would you change the program to create a 200 by 400 scene?
- How would you change the program so that it depicts the landing of a green UFO (unidentified flying object)? Drawing the UFO per se is easy:

```
(overlay (circle 10 "solid" "green")
         (rectangle 40 4 "solid" "green"))
```

- 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 the definition of `create-rocket-scene.v5`. As 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 whimsical choice for an x-coordinate for the rocket. Even though `50` doesn't look like much of an expression, it actually is a repeated expression, too. In other words, we have two reasons to eliminate `50` from the function definition, and we leave it to you to do so.

One More Definition

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 `create-rocket-scene` isn't a height but a time. Our previous definitions of `create-rocket-scene` use the wrong name for the argument of the function; instead `h`—short for height—it ought to use `t` for time:

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

```
(define (create-rocket-scene t)
  (cond
    [(<= t ROCKET-CENTER-TO-TOP)
     (place-image ROCKET X t MTSCN)])
```

```
[(> t ROCKET-CENTER-TO-TOP)
 (place-image ROCKET X 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 best of programmers all the time because they need to help people with problems in music, economics, photography, physics, 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.

By now you know that a good teacher would 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”
(define WIDTH 100)
(define HEIGHT 100)

; properties of the descending rocket
(define A 1)

; various other constants
(define MTSCN (empty-scene WIDTH HEIGHT))


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

(define X 50)
```

```
; functions
(define (create-rocket-scene.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)
  (* 1/2 A (sqr t)))
```

Figure 8: Landing a rocket (version 6)

Now we can fix `create-rocket-scene`. Instead of comparing its argument `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 8](#). It consists of two function definitions: `create-rocket-scene.v6` and `distance`. The remaining constant definitions make the function definitions readable and modifiable. As always, you can run this program with `animate`:

```
| > (animate create-rocket-scene.v6)
```

In comparison to the previous versions of `create-rocket-scene`, this one shows that a program may consist of more than one function definition that refer to each other. This revelation shouldn't surprise you. Even the first version of `create-rocket-scene` used `+` and `/` and other functions. It's just that you think of those as built into DrRacket.

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 create these definitions and organize them so that you can read them easily, even months after completion. After all, you or someone else will want to make changes to these programs, and if you don't know how to read them and if you didn't organize them well, you will have a difficult time with even the smallest task. Otherwise you mostly know what there is to know and you can program.

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 is to know. You know that programming—and computing—is about 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 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 provide 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.

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

Not!

When you look at the “programming” book shelves in any random book store, you will see loads of books that promise to turn you into a programmer in 21 days or faster. Some boast that you don’t need to know anything. Once you have worked through the first part of this book, however, you know that neither of these approaches can create a solid understanding of programming.

Acquiring the mechanical skills of programming—learning how to write instructions or expressions that the computer understands, getting to know what functions are available in the libraries, and similar activities—are not helping you much with **real** programming. To make such claims is like saying that a 10-year old who knows how to dribble can play on a professional soccer (football) team. It is also like claiming that memorizing a thousand words from the dictionary and a few rules from a grammar book teaches you a foreign language.

Proper programming is far more than the mechanics of acquiring a language. It is about reading problem statements and extracting the important concepts. It is about figuring out what is really wanted. It is about exploring examples to strengthen your intuitive understanding of the problem. It is about organizing knowledge, and it is about knowing what you don’t know yet. It is about filling those last few gaps. It is about making sure that you know how and why your code works, and it means conveying this knowledge to all future readers of your code. In short, proper programming is about solving problems systematically and conveying the ideas within the code.

The rest of this book is all about these things; very little of the book’s content is about the mechanics of DrRacket, BSL or libraries. The book shows you how good computer programmers think about problems. And promised, you will even learn to see that this way of solving problems applies to other situations in life, e.g., the work of doctors and journalists, lawyers and engineers, or car mechanics and photographers.

Oh, and by the way, the rest of the book uses a tone that is more appropriate for a serious text than this prologue. Enjoy!

p.s.

What the book is also *not* about Many introductory books on programming contain a lot of material about the authors' favorite application discipline for programming: mathematics, physics, music, and so on. To some extent including such material is natural, because programming is obviously useful in those areas. Then again, this approach distracts from proper programming and usually fails to tease apart the incidental from the essential. In contrast, this book focuses on programming and problem solving and what computer science can teach you in this regard. We have made every attempt to minimize the use of knowledge from other areas; for those few occasions when we went too far, we apologize.

v.6.3.0.2

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 part introduces the arithmetic of Beginning Student language (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

It is not necessary to read the entire chapter before you proceed. You may wish to scan some of the material here, skip ahead, and return, when you encounter “arithmetic” functions that you don’t recognize.

From [Prologue: How to Program](#), 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 say that two expressions are equal 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 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 pieces of data into another one,

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

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; or even determining the greatest common divisor of two numbers. 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 *Numbers* and arithmetic in all these forms. 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 these do in the interactions area, and then find out how many more there are and what they do.

If you need an operation on numbers that you know from grade school or high school, chances are that BSL knows about it, too. Guess its name and experiment in the interaction 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`.

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. If not, don’t worry; while complex numbers are useful for all kinds

You might not know `e` if you have not studied calculus. It’s a real number, close to 2.718, commonly called “Euler’s constant.”

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 as a mixed decimal. Play with your computer's mouse to find the menu that changes the fraction into decimal expansion and other presentations.

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 a mathematical 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 fewer even 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*, etc. **End**

Exercise 1. The direct goal of this exercise is to create an expression that computes the distance of some specific Cartesian point (x,y) from the origin $(0,0)$. The indirect goal is to introduce some basic programming habits, especially the use of the interactions area to develop expressions.

The values for `x` and `y` are given as definitions in the definitions area (top half) of DrRacket:

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

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.

To develop the desired expression, it is best to hit *RUN* and to experiment in the interactions area. The *RUN* action tells DrRacket what the current values of `x` and `y` are so that you can experiment with expressions that involve `x` and `y`:

```
> x
3
> y
4
> (+ x 10)
13
> (* x y)
12
```

Once you have the expression that produces the correct result, copy it from the interactions area to the definitions area, right below the two variable definitions.

To confirm that the expression works properly, change the two definitions so that `x` represents `12` and `y` stands for `5`. If you click *RUN* now, the result should be `13`.

Your mathematics teacher would say that you computed the **distance formula**. To use the formula on alternative inputs, you need to open DrRacket, edit the definitions of `x` and `y` so they represent the desired coordinates, and click *RUN*. But this way of reusing the distance formula is cumbersome and naive. Instead, we will soon show you a way to define functions, which makes re-using formulas

straightforward. For now, we use this kind of exercise to call attention to the idea of functions and to prepare you for programming with them. ▀

1.2 The Arithmetic of Strings

A wide-spread prejudice about computers concerns its innards. Many believe that it is all about bits and bytes—whatever those are—and possibly numbers, because everyone knows that computers can calculate. While it is true that electrical engineers must understand and study the computer as just such an object, beginning programmers and everyone else need never (ever) succumb to this thinking.

Programming languages are about computing with information, and information comes in all shapes and forms. For example, a program may deal with colors, names, business letters, or conversations between people. Even though we could encode this kind of information as numbers, it would be a horrible idea. Just imagine remembering large tables of codes, such as `0` means “red” and `1` means “hello,” etc.

Instead most programming languages provide at least one kind of data that deals with such symbolic information. For now, we use BSL’s strings. Generally speaking, a *String* is a sequence of the characters that you can enter on the keyboard enclosed in double quotes, plus a few others, about which we aren’t concerned just yet. In [Prologue: How to Program](#), we have seen a number of BSL strings: `"hello"`, `"world"`, `"blue"`, `"red"`, etc. The first two are words that may show up in a conversation or in a letter; the others are names of colors that we may wish to use.

Note We use *1String* to refer to the keyboard characters that make up a *String*. For example, `"red"` consists of three such *1Strings*: `"r"`, `"e"`, `"d"`. As it turns out, there is a bit more to the definition of *1String* but for now, thinking of them as *Strings* of length 1 is fine. **End**

BSL includes only one operation that exclusively consumes and produces strings: `string-append`, which, as we have seen in [Prologue: How to Program](#) concatenates two given strings into one. Think of `string-append` as an operation that is just like `+`. While the latter consumes two (or more) numbers and produces a new number, the former consumes two or more strings and produces a new string:

```
> (string-append "what a " "lovely " "day" " for learning BSL")
"what a lovely day for learning BSL"
```

Nothing about the given numbers changes when `+` adds them up; and nothing about the given strings changes when `string-append` concatenates them into one big string. If you wish to evaluate such expressions, you just need to think that the obvious laws hold for `string-append` just like they hold for `+`:

<code>(+ 1 1) == 2</code>	<code>(string-append "a" "b") == "ab"</code>
<code>(+ 1 2) == 3</code>	<code>(string-append "ab" "c") == "abc"</code>
<code>(+ 2 2) == 4</code>	<code>(string-append "a" " " "c") == "a c"</code>
....

Exercise 2. Add the following two lines to the definitions area:

```
(define prefix "hello")
(define suffix "world")
```

Then use string primitives to create an expression that concatenates `prefix` and `suffix` and adds `"_"` between them. When you run this program, you will see `"hello_world"` in the interactions area.

See [exercise 1](#) for how to create expressions using DrRacket. ▀

1.3 Mixing It Up

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

- `string-length` consumes a string and produces a (natural) number;
- `string-i-th` consumes a string `s` together with a natural number `i` and then 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 interaction area. Give them appropriate arguments, 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 Prologue:

```
(+ (string-length "hello world") 60)
```

The inner expression applies `string-length` to our favorite string, `"hello world"`. The outer expression has `+` consume the result of the inner expression and `60`. Let us calculate out the result:

```
(+ (string-length "hello world") 60)
==
(+ 11 60)
==
71
```

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, e.g.,

```
(+ (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 computer scientists start counting at `0`. Thus, the *5th* letter in this example is `"w"`, because the `0th` 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](#). Then create an expression that deletes the i th position from `str`. Clearly this expression creates a shorter string than the given one; contemplate which values you may choose for `i`. ■

1.4 The Arithmetic of Images

Images represent symbolic data somewhat like strings. Like strings, you used DrRacket to insert images wherever you would insert an expression into your program, because images are values just like numbers and strings.

To work with images, use the `2htdp/image` library.

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 need to know is that *mode strings* means either "`solid`" or "`outline`" and *color strings* are strings such as "`orange`", "`black`", etc.

Play with these operations in the interactions window:

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

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

> (star 12 "solid" "green")

```

Stop! The above uses a previously-unmentioned operation. Look up its documentation and find out how many more image creation operations there are in BSL. Look up the documentation for `regular-polygon`. Experiment!

The second kind of functions on images concern image properties:

- `image-width` determines the width of a given 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
```

Determine the value of

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

step by step.

A proper understanding of the third kind of image primitives—functions that compose images—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. You encountered this function in the [Arithmetic and Arithmetic](#).
- `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.
- `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 framed rectangle of a specified 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.
- `scene+line` consumes a scene, four numbers, and a color to draw a line of that color into the given image. Again, experiment with it to find out how the four arguments work together.

The laws of arithmetic for images are analogous to those for numbers:

```
(+ 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")
                  100 100
                  (empty-scene 200 200))
      ==
      
....
```

Again, no image gets destroyed or changed. Like `+`, these image combinations just make up new images that combine the given ones in the requested manner.

Exercise 5. Add the following line to the definitions area:

Copy and paste the image into your DrRacket.



```
(define cat )
```

Create an expression that counts the number of pixels in the image. See [exercise 1](#) for how to create expressions in DrRacket. |

Exercise 6. Use the picture primitives to create the image of a simple automobile. |

Exercise 7. Use the picture primitives to create the image of a simple boat. |

Exercise 8. Use the picture primitives to create the image of a simple tree. |

1.5 The Arithmetic of Booleans

We need one last kind of primitive data before we can design programs: Boolean values. There are only two kinds of *Boolean* values: `#true` and `#false`. Programs use Boolean values for representing decisions or the status of switches.

Computing with Boolean values is simple, too. In particular, BSL programs get away with three operations: `or`, `and`, and `not`. These operations are kind of like addition, multiplication, and negation for numbers. Of course, because there are only two Boolean values, it is actually possible to demonstrate how these functions work in all possible situations:

- `or` checks whether any of the given Boolean values is `#true`:

```
> (or #true #true)
#true
> (or #true #false)
#true
> (or #false #true)
#true
> (or #false #false)
#false
```

- `and` checks whether all of the given Boolean values are `#true`:

```
> (and #true #true)
#true
> (and #true #false)
#false
> (and #false #true)
#false
> (and #false #false)
#false
```

- and `not` always picks the Boolean that isn't given:

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

It is no surprise that `or` and `and` may be used with more than two Boolean values.

Finally, there is more to `or` and `and` than these explanations suggest, but to explain the extra bit requires another look at mixing up data in nested expressions.

Exercise 9. Some everyday problems are expressible as a Boolean expression:

Dr. Nadeem Hamid suggested this formulation of the exercise.

Sample Problem: Suppose you want to decide whether today is an appropriate day to go to the mall. You go to the mall if it is either not sunny or if today is Friday (because that is when stores post new sales items).

Here is how you could go about it using your new knowledge about Booleans. First add the following two lines to the definitions area of DrRacket:

```
(define sunny #true)
(define friday #false)
```

Now create an expression that computes whether sunny is false or friday is true. So in this particular case, the answer is `#false`. (Why?)

See [exercise 1](#) for how to create expressions in DrRacket. How many possible combinations of `#true` and `#false` can you associate with sunny and friday? ■

1.6 Mixing It Up with Booleans

One important use of Boolean values concerns calculations with many different kinds of data. We know from the prologue that BSL programs may name values via definitions. For example, we could start a program like this

```
(define x 2)
```

and then compute its inverse:

```
(define inverse-of-x (/ 1 x))
```

This works fine, as long as we don't edit the program and change `x` to `0`.

This is where Boolean values come in, in particular conditional calculations. First, the primitive function `=` determines whether two (or more) numbers are equal. If so, it produces `#true`, otherwise `#false`. Second, there is a kind of BSL expression that we haven't mentioned so far: the `if` expression. It uses the word "if" as if it were a primitive function; it isn't. The word "if" is followed by three expressions, separated by blank spaces (that includes tabs, line breaks, etc). Naturally the entire expression is enclosed in parentheses. Here is an example:

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

This `if` expression contains the sub-expressions `(= x 0)`, `0`, and `(/ 1 x)`. The evaluation of this expression proceeds in two steps:

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. Whatever their results are, they are also the result of the entire `if` expression.

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

You can experiment with `if` expressions in the interactions area:

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

And, using the laws of arithmetic, you can figure out the result of this interaction 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 subexpression 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 are ever in a position where you 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 with them, guess, and then check in the documentation whether you guessed right.

The dots here aren't a part of the program of course. Replace them with a string that refers to a color.

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" ...))
```

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 10. Add the following line to the definitions area:



```
(define cat )
```

Create an expression that computes whether the image is `"tall"` or `"wide"`. An image should be labeled `"tall"` if its height is larger or equal to its width; otherwise it is `"wide"`. See [exercise 1](#) for how to create expressions in DrRacket; as you experiment, replace the image of the cat with rectangles of your choice to ensure 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 high-lighting of the expression. 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 into both DrRacket's interactions area and in the definitions area (plus clicking 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 adding `#true` to `"hello world"`. 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 place holder 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: `string?`, `image?`, and `boolean?`. Experiment with them to ensure 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, 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

Evaluate (`sqrt -1`) in the interactions area and take a close look at the result. The result you see is the first so-called complex number anyone encounters. While your mathematics teacher may have told you that one doesn't compute the square root of negative numbers, truth is that real mathematicians and programmers find it acceptable and useful to do so anyway. But don't worry: understanding complex numbers is not essential to

idea before. For now, experiment with `exact?` and `inexact?` to make sure they

being a programmer.

perform the checks that their names suggest. Later we are going to discuss the nature of numbers in some detail.

Exercise 11. Add the following line to the definitions area of DrRacket:

```
(define in "hello")
```

Then create an expression that converts whatever `in` represents to a 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, 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 12. 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 re-acquaints 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, the calendar program on a computer may launch a monthly payroll program on the last day of every month, or a spreadsheet program may react to an accountant’s key presses by filling some cells with numbers. Lastly, programs may not consume all of its input data at once; instead a program 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, we write down

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

And that is all there is to function definitions. Here are some silly examples:

- `(define (f x) 1)`

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

Before we explain why these examples are silly, we need to explain what function definitions mean. Roughly speaking, a function definition introduces a new operation on data; put differently, it adds an operation to our vocabulary if we think of the primitive operations as the ones that are always available. Like a primitive function, a defined function consumes inputs. The number of variables determines how many inputs—also called *arguments* or *parameters*—a function consumes. Thus, `f` is a one-argument function, sometimes called a *unary* function. In contrast, `g` is a two-argument function, also dubbed *binary*, and `h` is a *ternary* or three-argument function. The expression—often referred to as the *function body*—determines the output.

The examples are silly because the expressions inside the functions do not involve the variables. Since variables are about inputs, not mentioning them in the expressions means that the function's output is independent of their input and therefore always the same. We don't need to write functions or programs if the output is always the same.

Variables aren't data; they represent data. For example, a constant definition such as

```
| (define x 3)
```

says that `x` always stands for `3`. The variables in a *function header*, i.e., the variables that follow the function name, are placeholders for **unknown** pieces of data, the inputs of the function, and mentioning them in the function body means referring to these pieces of data—once they become known.

Consider the following fragment of a definition:

```
| (define (ff a) ...)
```

Its function header is `(ff a)`, meaning `ff` consumes one piece of input, and the variable `a` is a placeholder for this input. Of course, at the time we define a function, we don't know what its input(s) will be. Indeed, the whole point of defining a function is that we can use the function many times on many different inputs.

Useful function bodies refer to the function parameters. A reference to a function parameter is really a reference to the piece of data that is the input to the function. If we complete the definition of `ff` like this

```
| (define (ff a)
|   (* 10 a))
```

we are saying that the output of a function is ten times its input. Presumably this function is going to be supplied with numbers as inputs, because it makes no sense to multiply images or Boolean values or strings by `10`.

For now, the only remaining question is how a function obtains its inputs. And to this end, we turn to the notion of applying a function.

Applications A *function application* puts **defined** functions to work and it looks just like the applications of a pre-defined operation:

- write “(”,
- write down the name of a defined function `f`,
- write down as many arguments as `f` consumes, separated by some space, and
- write down “)”.

With this bit of explanation, you can now experiment with functions in the interactions area just as we suggested you experiment with primitives to find out what they compute. The following four experiments, for example, confirm that `f` from above produces the same value no matter what input it

is applied to:

```
> (f 1)
1
> (f 2)
1
> (f "hello world")
1
> (f #true)
1
```

Stop! Evaluate `(f (circle 3 "solid" "red"))` in the interactions area.

Remember to add `(require 2htdp/image)` to DrRacket.

See, even images as inputs don't change `f`'s behavior. But here is what happens when the function is applied to too few or too many arguments:

```
> (f)
f: expects 1 argument, but found none
> (f 1 2 3 4 5)
f: expects only 1 argument, but found 5
```

DrRacket signals an error that is just like those you see when you apply a primitive to the wrong number of arguments:

```
> (+)
+: expects at least 2 arguments, but found none
```

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 13. 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 for this function for concrete values of x and y . Now all you really need to do is add a function header. ▀

Exercise 14. Define the function `cube-volume`, which accepts the length of a side of an equilateral cube and computes its volume. If you have time, consider defining `cube-surface`, too.

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

Exercise 15. Define the function `string-first`, which extracts the first `1String` from a non-empty string. Don't worry about empty strings. ▀

Exercise 16. Define the function `string-last`, which extracts the last `1String` from a non-empty string. Don't worry about empty strings. ▀

Exercise 17. Define the function `bool-implies`. It consumes two Boolean values, call them `sunny` and `friday`. The answer of the function is `#true` if `sunny` is false or `friday` is true.

Note Logicians call this Boolean operation *implication* and often use the notation $=>$, pronounced

“implies,” for this purpose. While we could use this name in BSL, it is too similar to the comparison operations for numbers `<=` and `>=` and might cause confusion. See [exercise 9](#). ■

Exercise 18. Define the function `image-area`, which counts the number of pixels in a given image. See [exercise 5](#) for ideas. ■

Exercise 19. Define the function `image-classify`, which consumes an image and produces “`tall`” if the image is taller than it is wide, “`wide`” if it is wider than it is tall, or “`square`” if its width and height are the same. See [exercise 10](#) for ideas. ■

Exercise 20. Define the function `string-join`, which consumes two strings and appends them with “`_`” in between. See [exercise 2](#) for ideas. ■

Exercise 21. Define the function `string-insert`, which consumes a string and a number `i` and which inserts “`_`” at the `i`th position of the string. Assume `i` is a number between `0` and the length of the given string (inclusive). See [exercise 3](#) for ideas. Also ponder the question how `string-insert` ought to cope with empty strings. ■

Exercise 22. Define the function `string-delete`, which consumes a string and a number `i` and which 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
```

And that last equation is weird, because `x` does not occur in the body of `f`. Replacing all occurrences of `x` with `2` produces just the function body, which is `1`.

For `ff`, whose parameter is `a` and whose body is `(* 10 a)`, 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 on, 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 (* 10 a), 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 DrRacket reuses the result of the above calculation,
(* 40 32)
==
1280 ; because it is really just math

```

In sum, DrRacket is basically just 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” buttons 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 9 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.

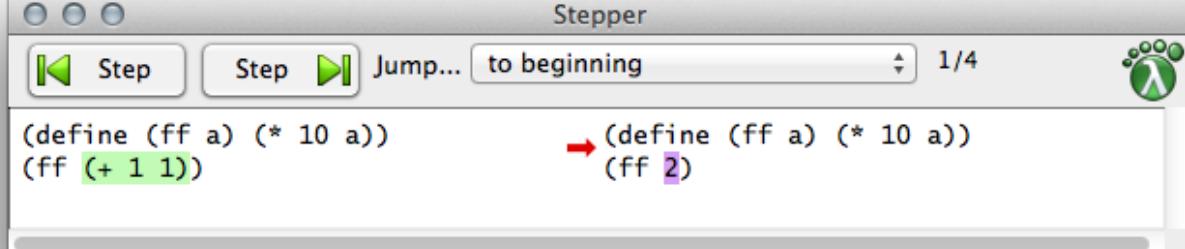


Figure 9: The DrRacket stepper

Exercise 23. 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 "hello" ", " "world") == "hello, 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,"
```

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 and operations on data. Surprisingly the laws of arithmetic suffice to explain most of the programs. `Abstraction` generalizes the law of substitution, and this is the only new law you encounter here.

Eventually you will encounter so-called imperative operations, which do not combine or extract values but **change** them. For such operations, it is necessary to supplement the laws of arithmetic and substitution with a few others. But, the basic way of determining the results of programs remains the same.

Exercise 24. Use DrRacket's stepper on this program fragment:

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

Exercise 25. 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. ■

Exercise 26. Here is the definition of `bool-implies`:

```
(define (bool-implies x y)
  (or (not x) y))
```

Use the stepper to determine the value of `(bool-implies #true #false)`. ■

Exercise 27. Take a look at this attempt to solve [exercise 19](#):

```
(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"])
```

Use DrRacket's stepper to determine the result of

```
(image-classify (circle 3 "solid" "red"))
```

Does the stepping suggest how to fix this attempt? □

Exercise 28. What do you expect as value of the following program:

```
(define (string-insert s i)
  (string-append (substring s 0 i) "_" (substring s i)))
(string-insert "helloworld" 6)
```

Confirm your expectation with DrRacket and its stepper. □

2.3 Composing Functions

This section's example uses the `2htdp/batch-io` library. Use either the `require` form or the drop-down menu to add it to your definitions.

A program rarely consists of a single function definition and an application of that function. Instead, a typical program consists of a “main” function, and it uses other functions—some that come with the chosen programming language and some programmer-defined ones—and turns the result of one function application into the input for another. In analogy to algebra, we call this way of defining functions *composition*, and we call these additional functions *auxiliary functions* or *helper functions*.

Consider a program for filling in letter templates. It consumes the first and last name of the addressee as well as a signature. It produces a complete letter by plugging in holes in the opening, the body, and the signature of the template.

Here is a complete definition, consisting of four functions:

```
(define (letter fst lst signature-name)
  (string-append
    (opening fst)
    "\n\n"
    (body fst lst)
    "\n\n"
    (closing signature-name)))

(define (opening fst)
  (string-append "Dear " fst ","))

(define (body fst lst)
  (string-append
    "We have discovered that all people with the last name " "\n"
    lst " have won our lottery. So, " fst ", " "\n"
    "hurry and pick up your prize."))

(define (closing signature-name)
  (string-append
    "Sincerely,"
    "\n\n"
    signature-name
    "\n"))
```

The first one is the “main” function. It uses three auxiliary functions to produce the three pieces of the letter—the opening, the body, and the complete signature—and then composes the three results in the

correct order with `string-append`.

Stop! Enter these definitions into DrRacket's definitions area, click on *RUN*, and evaluate these expressions in the interactions area:

```
> (letter "Matthew" "Fisler" "Felleisen")
"Dear Matthew,\n\nWe have discovered that all people with ..\n"
> (letter "Kathi" "Felleisen" "Findler")
"Dear Kathi,\n\nWe have discovered that all people with ..\n"
```

Note how the result is a long string that contains occurrences of "`\n`". These substrings represent the newline character within a string.

Think of `'stdout` as a String for now.

If we were to **print** this string—render it as text in a file or a console—the lines would be broken at these places, which is precisely what you would want:

```
> (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
'standard
```

The `require` brings in `write-file`, a function that can print a string to a file or console. [Programs](#) explains this concept in some more depth.

In general, when a problem refers to distinct tasks of computation, a program should consist of one function per task and a main function that puts it all together. We formulate this idea as a simple slogan:

Define one function per task.

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: Imagine the owner of a monopolistic movie theater. He has complete freedom in setting ticket prices. The more he charges, the fewer the people who 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 us translate this idea into a mathematical formula:

$$\text{avg. attendance} = 120 - \frac{\text{change in price}}{0.10} \cdot 15 \text{ people}$$

Stop! Explain the minus sign before you proceed.

Unfortunately, the increased attendance also comes at an increased cost. Every performance comes at a fixed costs 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 so that he can maximize his 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))))
```

The function mentions the four constants—120, 5.0, 15, and 0.1—determined by the owner's experience.

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)))
```

The number of attendees is calculated by the *attendees* function of course.

3. The *costs* consist 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 also consumes the price of a ticket and uses it to compute the number of tickets sold with attendees:

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

Again, this function also uses *attendees* to determine the number of attendees.

4. Finally, *profit* is the difference between revenue and costs:

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

Even the definition of profit suggests that we use the functions *revenue* and *cost*. Hence, the *profit* function must consume the price of a ticket and hand this number to the two functions it uses.

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 29. 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 30. Determine the potential profit for the following ticket prices: \$1, \$2, \$3, \$4, and \$5. Which price should the owner of the movie theater choose to maximize his profits? Determine the best ticket price down 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 31. After studying the costs of a show, the owner discovered several ways of lowering the cost. As a result of his improvements, he no longer has a fixed cost. He now simply pays \$1.50 per attendee.

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 [Prologue: How to Program](#) 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 shapes and 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 upper-case 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 location of the center.

Again, we use a simple slogan to remind you about the importance of constants:

Introduce definitions for all constants mentioned in a problem statement.

Exercise 32. Define constants for the price optimization program 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 composes 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. We call the appearance of an input 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.

In this section we present simple examples of both batch and interactive programs.

Batch Programs As mentioned, a batch program consumes all of its inputs at once and computes the result from these inputs. Its main function may expect the arguments themselves or the names of files from which to retrieve the inputs; similarly, it may just return the output or it may place it in a file.

Once programs are created, we want to use them. In DrRacket, we launch batch programs in the interactions area so that we can watch the program at work.

Programs are even more useful if they can retrieve the input from some file and deliver the output to

some other file. The name batch program originated from the early days of computing when a program read an entire file (or several files) and placed the result in some other file(s), without any intervention after the launch. Conceptually, we can think of the program as reading an entire file at once and producing the result file(s) all at once.

We create such file-based batch programs with the `2htdp/batch-io` library, which adds two functions to our vocabulary (among others):

- `read-file`, which reads the content of an entire file as a string, and
- `write-file`, which creates a file from a given string.

These functions write strings to files and read strings from them:

```
> (write-file "sample.dat" "212")
"sample.dat"
> (read-file "sample.dat")
"212"
```

Before you evaluate these expressions in DrRacket,
save the definitions area in a file.

After the first interaction is completed, the file named `"sample.dat"` contains this text

`212`

and nothing else. The result of `write-file` is an acknowledgment that it has placed the string in the file. If the file already exists, it replaces its content with the given string; otherwise, it creates a file and makes the given string its content. The second interaction, (`read-file "sample.dat"`), produces `"212"` because it turns the content of `"sample.dat"` into a string.

The names `'stdout` and `'stdin` are short for standard output and input device, respectively.

For historical and pragmatic reasons, `write-file` also accepts `'stdout`, a special kind of token, as the first argument. It then displays the resulting file content in the current interactions area, for example:

```
> (write-file 'stdout "212\n")
212
'standard-output
```

By analogy, `read-file` accepts `'stdin` in lieu of a file name and then reads input from the current interactions area.

Let us illustrate the creation of a batch program with a simple example. Suppose we wish to create a program that converts a temperature measured on a Fahrenheit thermometer into a Celsius temperature. Don't worry, this question isn't a test about your physics knowledge; here is the conversion formula:

This book is not about memorizing facts, but we do expect you to know where to find them. Do you know where to find out how temperatures are converted?

$$c = \frac{5}{9} \cdot (f - 32)$$

Naturally in this formula f is the Fahrenheit temperature and c is the Celsius temperature. While this formula might be good enough for a pre-algebra text book, a mathematician or a programmer would write $c(f)$ on the left side of the equation to remind readers that f is a given value and c is computed from f .

Translating this into BSL is straightforward:

```
(define (f2c f)
  (* 5/9 (- f 32)))
```

Recall that `5/9` is a number, a rational fraction to be precise, and more importantly, that c depends on the given f , which is what the function notation expresses.

Launching this batch program in the interactions area works as usual:

```
> (f2c 32)
0
> (f2c 212)
100
> (f2c -40)
-40
```

But suppose we wish to use this function as part of a program that reads the Fahrenheit temperature from a file, converts this number into a Celsius temperature, and then creates another file that contains the result.

Once we have the conversion formula in BSL, creating the main function means composing `f2c` with existing primitive functions:

```
(define (convert in out)
  (write-file out
    (string-append
      (number->string
        (f2c
          (string->number
            (read-file in))))))
  "\n"))
```

We have called the main function `convert`, and it consumes two filenames: `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 us step through `convert`'s body carefully to understand how this works:

1. `(read-file in)` retrieves the content of the file called `in` as a string;
2. `string->number` turns this string into a number;
3. `f2c` 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;
5. and `(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 what

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

does.

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 this idea.

Since `write-file` is a way to create files, we can now launch `convert`:

You can also create "sample.dat" with a file editor.
Just be careful that the editor doesn't add a newline
or any other invisible characters.

```
> (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 use another text editor to look at the file and to view the result.

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 33. Recall the `letter` program from [Composing Functions](#). We launched this program once, with the inputs `"Matthew"`, `"Fisler"`, and `"Felleisen"`. 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
'standard
```

Of course, programs are useful because you can launch them for many different inputs, and this is true for `letter`, too. 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 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.

Note Once you understand [Programming with Lists](#), you will be able to use other functions from the `2htdp/batch-io` library, and then you will have no problem writing letters for tens of thousands of lucky lottery winners. ■

Interactive Programs No matter how you look at it, batch programs are old-fashioned. Even if businesses have used them for decades to automate useful tasks, people prefer interactive programs. Indeed, in this day and age, people mostly interact with desktop applications via a keyboard and a mouse generating input events such as key presses or mouse clicks. 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 34. Most people no longer use desktop computers to run applications but 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. ▀

We use “re-introduce” because [Prologue: How to Program](#) introduces the mechanics already. Even if you have read the Prologue, we urge you to read this section, too, because it provides a different perspective.

The purpose of this section is to re-introduce the mechanics of writing **interactive** BSL programs. Because many of the large 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 key strokes. We say that the key stroke 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, or print a document. To accomplish these tasks, it also runs functions that translate the operating system’s data into sounds, images, and actions on the printer.

Naturally, different programs have different needs. One program may interpret key strokes 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 tick, 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, one of its programming languages, comes with the *2htdp/universe* library, which provides this communication mechanism. That is, **big-bang** is your means to install event handlers and functions that translate data into presentable form. A **big-bang** expression consists of one required sub-expression and one required clause. The sub-expression evaluates to the *initial state* of the program, and the required clause tells DrRacket how to render such a state. Other clauses—all optional—describe how to transform the state that **big-bang** tracks, which is why we also 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. We 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*. The book uses this terminology when it switches from the mechanics of world programs to their design in [Designing World Programs](#). End

Let us understand this idea step-by-step, starting with this function definition:

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

The function consumes a positive number and produces 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 x 100 red square. In addition, the DrRacket interactions area does not display another prompt; it is as if the program keeps running and indeed, this is 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 an 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 is finished with its work, use **number->square** to render the state as an image.

Now hit the return key and observe what happens. Eventually the evaluation of the expressions terminates and DrRacket displays **0**.

Stop! Explain what happens when you terminate the program evaluation by clicking *STOP* before it stops by itself.

The **big-bang** expression keeps track of the current state. Initially this state is **100**. Every time the clock ticks, it calls the clock tick handler and gets a new state. Hence, the state of **big-bang** changes as follows:

100, 99, 98, ..., 2, 1, 0

When the state's value becomes **0**, the evaluation is done. For every other state—from **100** to **1**—**big-bang** translates the state into an image, using **number->square** as the **to-draw** clause tells it to. Hence, the window displays a red square that shrinks from 100 x 100 pixels to 1 x 1 pixel over 100 clock ticks.

Let's add a clause for dealing with key events. First, we need a function that consumes the current state and a string that describes the key event and that returns a new state:

```
(define (reset s ke)
```

```
> (big-bang 100)
```

This function throws away its arguments and returns `100`, which is the initial state of the `big-bang` expression we wish to modify. Second, we add an `on-key` clause to the `big-bang` expression:

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

It says to call `reset` every time a key is pressed.

Stop! Explain what happens when you hit *RETURN*, count to 10, and press "a".

What you will see is that the red square shrinks again, one pixel per clock tick. As soon as you press the "a" key on the keyboard though, the red square re-inflates to full size, because `reset` is called on the current length of the square and "a" and returns `100`. This number becomes `big-bang`'s new state and `number->square` renders it as a full-sized red square.

In order to understand the evaluation of `big-bang` expressions in general, let us look at a schematic one:

```
(big-bang cw0
  [on-tick tock]
  [on-key ke-h]
  [on-mouse me-h]
  [to-draw render]
  [stop-when end?]
  ...)
```

This `big-bang` expression specifies three event handlers: `tock`, `ke-h`, and `me-h`. The last one is a mouse event handler, which consumes four inputs—the current state, two coordinates, and a string, which represents the kind of mouse event—and, like the other event handlers, returns a new state.

The evaluation of this `big-bang` expression starts with `cw0`, which is usually an expression. DrRacket, our operating system, installs the value of `cw0` as the current state. It uses `render` to translate the current state into an image, which is then displayed in a separate window. Indeed, `render` is the **only** means for a `big-bang` expression to present data to the external world.

Here is how events are processed:

- Every time the clock ticks, DrRacket applies `tock` to `big-bang`'s current state and receives a value in response; `big-bang` treats this return value as the next current state.
- Every time a key is pressed, DrRacket applies `ke-h` to `big-bang`'s current state and a string that represents the key; for example, pressing the "a" key is represented with "a" and the left arrow key with "left". When `ke-h` returns a value, `big-bang` treats it as the next current state.
- Every time a mouse enters the window, leaves it, moves, or is pressed, DrRacket applies `me-h` to `big-bang`'s current state, the event's x- and y-coordinates, and a string that represents the kind of mouse event that happened; for example, pressing a mouse's button is represented with "button-down". When `me-h` returns a value, `big-bang` treats it as the next current state.

All events are processed in order; if two events seem to happen at the same time, DrRacket acts as a tie-breaker and arranges them in some order.

After an event is processed, `big-bang` uses both `end?` and `render` to check the current state:

- `(end? cw)` produces a Boolean value. If it is `#t`, `big-bang` stops the computation immediately. Otherwise it proceeds.
- `(render cw)` is expected to produce an image and `big-bang` displays this image in a separate

window.

current state	cw_0	cw_1	...
event	e_0	e_1	...
on clock tick	(<code>tock cw0</code>)	(<code>tock cw1</code>)	...
on key stroke	(<code>ke-h cw0 e0</code>)	(<code>ke-h cw1 e1</code>)	...
on mouse event	(<code>me-h cw0 e0 ...</code>)	(<code>me-h cw1 e1 ...</code>)	...
its image	(<code>render cw0</code>)	(<code>render cw1</code>)	...

Figure 10: How `big-bang` works

The table in [figure 10](#) concisely summarizes this process. In the first row, it lists the current states: the first one, the second, and so forth. The second row enumerates a series of events e_i in the order in which DrRacket perceives them. Each e_i might be a clock tick, a key press, or a mouse event. The next three rows specify the result of dealing with the event:

- If e_i is a clock tick, `big-bang` evaluates (`tock cw0`) to produce cw_1 , (`tock cw1`) in the second, and so forth.
- Next, if the event is a key event, (`ke-h cw0 e0`) is evaluated and yields cw_1 . In the second column, `big-bang` uses (`ke-h cw1 e1`) instead.

The handler must be applied to the event itself because, in general, programs are going to react to each key differently.

Stop! Why does `tock` not get applied to the event?

- Finally, if the event is a mouse event, `big-bang` runs (`me-h cw0 #, @math \{e_0\} ...`) to get cw_1 . The call is schematic because a mouse event e_0 is really associated with several pieces of data—its nature and its coordinates—and we just wish to indicate that much.

Stop! How does `big-bang` proceed if e_1 is a mouse event?

With `render`, this series of current states is mapped to a series of images, which DrRacket displays in the separate window.

Let us interpret this table with the following sequence of events: the user presses the “a” key, then the clock ticks, and finally the user presses uses the mouse to trigger a “button down” event at position (90,100). Then

1. cw_1 is the result of (`ke-h cw0 "a"`), i.e., the fourth cell in the second column;
2. cw_2 is the result of (`tock cw1`), i.e., the third cell in the third column; and
3. cw_3 is the result of (`me-h cw2 90 100 "button-down"`).

We can actually express these three steps as a sequence of three definitions:

```
(define cw1 (ke-h cw0 "a"))
(define cw2 (tock cw1))
(define cw3 (me-h cw2 "button-down" 90 100))
```

Stop! Determine manually how `big-bang` displays each of these three states as an image. Furthermore, we can also compute cw_3 via a function composition:

```
(define cw3 (me-h (tock (ke-h cw0 "a")) 90 100 "button-down")))
```

Event handlers really are functions, and you can compose them just like ordinary functions. So it is perfectly acceptable to use the result of one function application and use it as the input for another one.

In short, the sequence of events determines in which order you traverse the above table of possible

states to arrive at the one and only one current state for each time slot. Note that DrRacket does not touch the current state; it merely safeguards it and passes it to event handlers and other functions when needed.

From here, it is straightforward to define a first interactive program:

```
(define (main y)
  (big-bang y
    [on-tick sub1]
    [stop-when zero?]
    [to-draw place-dot-at]
    [on-key stop])))

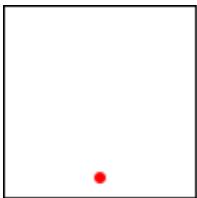
(define (place-dot-at y)
  (place-image (circle 3 "solid" "red") 50 y (empty-scene 100 100)))

(define (stop y ke)
  0)
```

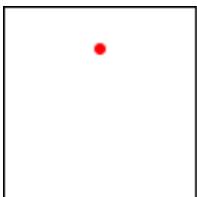
The program consists of three functions: a `main` function launches a `big-bang` interactive program; `place-dot-at` translates the current state into an image; and `stop` throws away its inputs and produces `0`.

After clicking *RUN*, we can ask DrRacket to evaluate applications of these handler functions. This is one way to confirm their workings or to just explore how they work:

```
> (place-dot-at 89)
```



```
> (place-dot-at 22)
```



```
> (stop 89 "q")
```

```
0
```

Stop! Try now to understand how `main` reacts when you press a key.

One way to find out is to launch `main`:

```
> (main 90)
```

Take a deep breath.

By now, you may feel that these first two chapters are overwhelming. They introduced so many new

concepts, including a new language, its vocabulary, its meaning, its idioms, a tool for writing down texts in this vocabulary, running these so-called “programs,” and the inevitable question of how to create them when presented with a problem statement. To overcome this feeling, the next chapter takes a step back and explains how to design programs systematically from scratch, especially interactive programs. So take a breather and continue when ready.

3 How to Design Programs

The first few chapters of this book show that learning to program requires some mastery of many concepts. On the one hand, programming needs some language, a notation for communicating what we wish to compute. The languages for formulating programs are artificial constructions, though acquiring a programming language shares some elements with acquiring a natural language: we need to learn the vocabulary of the programming language, to figure out its grammar, and to understand what its “phrases” mean.

On the other hand, it is critical to learn how to get from a problem statement to a program. We need to determine what is relevant in the problem statement and what can be ignored. We need to tease out what the program consumes, what it produces, and how it relates inputs to outputs. We have to know, or find out, whether the chosen language and its libraries provide certain basic operations for the data that our program is to process. If not, we might have to develop auxiliary functions that implement these operations. Finally, once we have a program, we must check whether it actually performs the intended computation. And this might reveal all kinds of errors, which we need to be able to understand and fix.

All this sounds rather complex and you might wonder why we don’t just muddle our way through, experimenting here and there, leaving well enough alone when the results look decent. This approach to programming, often dubbed “garage programming,” is common and succeeds on many occasions; sometimes it is the launching pad for a start-up company. Nevertheless, the start-up cannot sell the results of the “garage effort” because only the original programmers and their friends can use them.

A good program comes with a short write-up that explains what it does, what inputs it expects, and what it produces. Ideally, it also comes with some assurance that it actually works. In the best circumstances, the program’s connection to the problem statement is evident so that a small change to the problem statement is easy to translate into a small change to the program. Software engineers call this a “programming product.”

All this extra work is necessary because programmers don’t create programs for themselves. Programmers write programs for other programmers to read, and on occasion, people run these programs to get work done. Most programs are large, complex collections of collaborating functions, and nobody can write all these functions in a day. Programmers join projects, write code, leave projects; others take over their programs and work on them. Another difficulty is that the programmer’s clients tend to change their mind about what problem they really want solved. They usually have it almost right, but more often than not, they get some details wrong. Worse, complex logical constructions such as programs almost always suffer from human errors; in short, programmers make mistakes. Eventually someone discovers these errors and programmers must fix them. They need to re-read the programs from a month ago, a year ago, or twenty years ago and change them.

The word “other” also includes older versions of the programmer who usually cannot recall all the thinking that the younger version put into the production of the program.

Exercise 35. Research the “year 2000” problem and what it meant for programmers. ■

In this book, we present a design recipe that integrates a step-by-step process with a way of organizing programs around problem data. For the readers who don’t like to stare at blank screens for a long time, this design recipe offers a way to make progress in a systematic manner. For those of you who teach others to design programs, the recipe is a device for diagnosing a novice’s difficulties. For others, our recipe might be something that they can apply to other areas, say medicine, journalism, or engineering. For those who wish to become real programmers, the design recipe also offers a way to understand and work on existing programs—though not all programmers use a method like this design recipe to come

up with programs. The rest of this chapter is dedicated to the first baby steps into the world of the design recipe; the following chapters and parts refine and expand the recipe in one way or another.

3.1 Designing Functions

Information and Data The purpose of a program is to describe a computational process of working through information and producing new information. In this sense, a program is like the instructions a mathematics teacher gives to grade school students. Unlike a student, however, a program works with more than numbers; it calculates with navigation information, looks up a person’s address, turns on switches, or inspects the state of a video game. All this information comes from a part of the real world—often called the program’s *domain*—and the results of a program’s computation represents more information in this domain.

Information plays a central role in our description. Think of information as facts about the program’s domain. For a program that deals with a furniture catalog, a “table with five legs” or a “square table of two by two meters” are pieces of information. A game program deals with a different kind of domain, where “five” might refer to the number of pixels per clock tick that some objects travels on its way from one part of the canvas to another. Or, a payroll program is likely to deal with “five deductions.”

For a program to process information, it must turn it into some form of *data* in the programming language; then it processes the data; and once it is finished, it turns the resulting data into information again. An interactive program may even intermingle these steps, acquiring more information from the world as needed and delivering information in between.

We use BSL and DrRacket so that you do **not** have to worry about the translation of information into data. In DrRacket’s BSL you can apply a function directly to data and observe what it produces. As a result, we avoid the serious chicken-and-egg problem of writing functions that convert information into data and vice versa. For simple kinds of information, designing such program pieces is trivial; for anything other than simple information, you need to know about parsing, for example, and **that** immediately requires a lot of expertise in program design.

Software engineers use the slogan *model-view-controller* (MVC) for the way BSL and DrRacket separate data processing from parsing information into data and turning data into information. Indeed, it is now accepted wisdom that well-engineered software systems enforce this separation, even though most introductory books still co-mingle them. Thus, working with BSL and DrRacket allows you to focus on the design of the core of programs and, when you have enough experience with that, you can learn to design the information-data translation parts.

Here we use three pre-installed teachpacks—`2htdp/batch-io`, `2htdp/image`, `2htdp/universe`—to demonstrate how complete programs are designed. That is, this book—starting with this chapter—provides a design recipe for batch and interactive programs to give you an idea of how complete programs are designed. But, keep in mind that the libraries of full-fledged programming languages offer many more tools than these teachpacks.

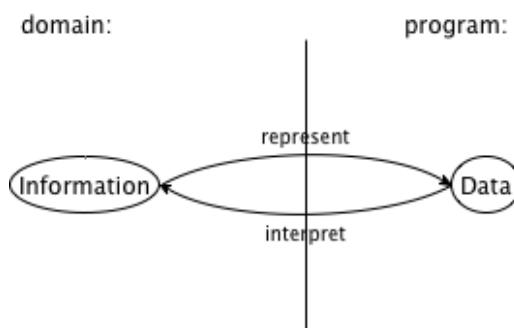


Figure 11: From information to data, and back

Given the central role of information and data, program design must clearly start with the connection between them. Specifically, we—the programmers—must decide how to use our chosen programming language to *represent* the relevant pieces of information as data and how we should *interpret* data as

information. [Figure 11](#) explains this idea with an abstract diagram.

To make this idea concrete, let us consider some examples. Suppose you are designing a program that consumes information in the form of numbers and that also produces numbers. Then choosing a representation and an interpretation requires understanding what a piece of data such as `42` means in the domain:

- `42` may refer to the number of pixels from the top margin in the domain of images;
- `42` may denote the number of pixels per clock tick that a simulation or game object moves;
- `42` may mean a temperature, on the Fahrenheit, Celsius, or Kelvin scale for the domain of physics;
- `42` may specify the size of some table if the domain of the program is a furniture catalog; or
- `42` could just count the number of characters in a string.

The key is to know how to go from numbers as information to numbers as data and vice versa.

Since this knowledge is so important for everyone who reads the program, we often write it down in the form of comments, which we call *data definitions*. The purpose of a data definition is two-fold. On the one hand, it names a *class* or a collection of data, typically using a meaningful word. On the other hand, it informs readers how to create elements of this class of data and how to decide whether some arbitrary piece of data is an element of this collection.

Computer scientists use “class” to mean something like a “mathematical set.”

Here is a data definition that could be useful for one of our early examples:

```
; Temperature is a Number.
; interpretation degrees Celsius
```

The first line introduces the name of the data collection, `Temperature`, and tells us that the class consists of all `Numbers`. So, for example, if we ask whether `102` is a temperature, you can respond with “yes” because `102` is a number and all numbers are temperatures. Similarly, if we ask whether “`cold`” is a `Temperature`, you will say “no” because no string belongs to `Temperature`. And, if we asked you to make up a sample `Temperature`, you might come up with something like `-400`.

If you happen to know that the lowest possible temperature is approximately `-274C`, you may wonder whether it is possible to express this knowledge in a data definition. Since data definitions in BSL are really just English descriptions of classes, you may indeed define the class of temperatures in a much more accurate manner than shown here. In this book, we use a stylized form of English for such data definitions, and the next chapter introduces the style for imposing constraints such as “larger than `-274`.”

At this point, you have encountered the names of some classes of data: `Number`, `String`, `Image`, and `Boolean` values. With what you know right now, formulating a new data definition means nothing more than introducing a new name for an existing form of data, for example, “temperature” for numbers. Even this limited knowledge, though, suffices to explain the outline of our design process.

The Process Once you understand how to represent input information as data and to interpret output data as information, the design of an individual function proceeds according to a straightforward process:

1. Express how you wish to represent information as data. A one-line comment suffices, for example,

```
; We use plain numbers to represent temperatures.
```

Formulate data definitions, like the one for `Temperature` above for the classes of data you consider critical for the success of your program.

2. Write down a signature, a purpose statement, and a function header.

A *function signature* (shortened to *signature* here) is a BSL comment that tells the readers of your

design how many inputs your function consumes, from what collection of data they are drawn, and what kind of output data it produces. Here are three examples:

- for a function that consumes one string and produces a number:

```
; String -> Number
```

- for a function that consumes a temperature and that produces a string:

```
; Temperature -> String
```

As this signature points out, introducing a data definition as an alias for an existing form of data makes it easy to read the intention behind signatures.

Nevertheless, we recommend to stay away from aliasing data definitions for now. A proliferation of such names can cause quite some confusion. It takes practice to balance the need for new names and the readability of programs, and there are more important ideas to understand for now.

- for a function that consumes a number, a string, and an image and that produces an image:

```
; Number String Image -> Image
```

A *purpose statement* is a BSL comment that summarizes the purpose of the function in a single line. If you are ever in doubt about a purpose statement, write down the shortest possible answer to the question

what does the function compute?

Every reader of your program should understand what your functions compute **without** having to read the function itself.

A multi-function program should also come with a purpose statement. Indeed, good programmers write two purpose statements: one for the reader who may have to modify the code and another one for the person who wishes to use the program but not read it.

Finally, a *header* is a simplistic function definition, also called a *stub*. Pick one parameter for each input data class in the signature; the body of the function can be any piece of data from the output class. The following three function headers match the above three signatures:

- (`define (f a-string) 0)`
- (`define (g n) "a")`
- (`define (h num str img) (empty-scene 100 100))`

Our parameter names reflect what kind of data the parameter represents. Sometimes, you may wish to use names that suggest the purpose of the parameter.

When you formulate a purpose statement, it is often useful to employ the parameter names to clarify what is computed. For example,

```
; Number String Image -> Image
; adds s to img, y pixels from top, 10 pixels to the left
(define (add-image y s img)
  (empty-scene 100 100))
```

At this point, you can click the *RUN* button and experiment with the function. Of course, the result is always the same value, which makes these experiments quite boring.

3. Illustrate the signature and the purpose statement with some functional examples. To construct a *functional example*, pick one piece of data from each input class from the signature and determine what you expect back.

Suppose you are designing a function that computes the area of a square. Clearly this function

consumes the length of the square's side, and that is best represented with a (positive) number. The first process step should have produced something like this:

```
; Number -> Number
; computes the area of a square whose side is len
(define (area-of-square len) 0)
```

Add the examples between the purpose statement and the function header:

```
; Number -> Number
; computes the area of a square whose side is len
; given: 2, expect: 4
; given: 7, expect: 49
(define (area-of-square len) 0)
```

- The next step is to take inventory, to understand what are the givens and what we need to compute. For the simple functions we are considering right now, we know that they are given data via parameters. While parameters are placeholders for values that we don't know yet, we do know that it is from this unknown data that the function must compute its result. To remind ourselves of this fact, we replace the function's body with a *template*.

We owe the term “inventory” to Dr. Stephen Bloch.

For now, the template contains just the parameters, e.g.,

```
; Number -> Number
; computes the area of a square whose side is len
; given: 2, expect: 4
; given: 7, expect: 49
(define (area-of-square len)
  (... len ...))
```

The dots remind you that this isn't a complete function, but a template, a suggestion for an organization.

The templates of this section look boring. Later, when we introduce complex forms of data, templates become interesting, too.

- It is now time to *code*. In general, to code means to program, though often in the narrowest possible way, namely, to write executable expressions and function definitions.

To us, coding means to replace the body of the function with an expression that attempts to compute from the pieces in the template what the purpose statement promises. Here is the complete definition for `area-of-square`:

```
; Number -> Number
; computes the area of a square whose side is len
; given: 2, expect: 4
; given: 7, expect: 49
(define (area-of-square len)
  (sqr len))
```

To complete the `add-image` function takes a bit more work than that:

```
; Number String Image -> Image
; adds s to img, y pixels from top, 10 pixels to the left
; given:
; 5 for y,
; "hello" for s, and
; (empty-scene 100 100) for img
; expected:
; (place-image (text "hello" 10 "red") 10 5 (empty-scene 100 100))
(define (add-image y s img)
```

```
(place-image (text s 10 "red") 10 y img))
```

In particular, the function needs to turn the given string s into an image, which is then placed into the given scene.

5. The last step of a proper design is to test the function on the examples that you worked out before. For now, click the *RUN* button and enter function applications that match the examples in the interactions area:

```
> (area-of-square 2)
4
> (area-of-square 7)
49
```

The results must match the output that you expect; you must inspect each result and make sure it is equal to what is written down in the example portion of the design. If the result doesn't match the expected output, consider the following three possibilities:

- a. You miscalculated and determined the wrong expected output for some of the examples.
- b. Alternatively, the function definition computes the wrong result. When this is the case, you have a *logical error* in your program, also known as a *bug*.
- c. Both the examples and the function definition are wrong.

When you do encounter a mismatch between expected results and actual values, we recommend that you first re-assure yourself that the expected results are correct. If so, assume that the mistake is in the function definition. Otherwise, fix the example and then run the tests again. If you are still encountering problems, you may have encountered the third, somewhat rare situation.

3.2 Finger Exercises: Functions

The first few of the following exercises are almost copies of those in [Functions](#), though where the latter use the word “define” the exercises below use the word “design.” What this difference means is that you should work through the design recipe to create these functions and your solutions should include all relevant pieces.

As the title of the section suggests, these exercises are practice exercises to help you internalize the process. Until the steps become second nature, never skip one, because doing so leads to easily avoidable errors. There is plenty of room left in programming for complicated errors; we have no need to waste our time on silly ones.

Exercise 36. Design the function `string-first`, which extracts the first character from a non-empty string. Don’t worry about empty strings. |

Exercise 37. Design the function `string-last`, which extracts the last character from a non-empty string. |

Exercise 38. Design the function `image-area`, which counts the number of pixels in a given image. |

Exercise 39. Design the function `string-rest`, which produces a string like the given one with the first character removed. |

Exercise 40. Design the function `string-remove-last`, which produces a string like the given one with the **last** character removed. |

3.3 Domain Knowledge

It is natural to wonder what knowledge it takes to code up the body of a function. A little bit of reflection tells you that this step demands an appropriate grasp of the domain of the program. Indeed, there are two forms of such *domain knowledge*:

1. Knowledge from external domains such as mathematics, music, biology, civil engineering, art, etc.
Because programmers cannot know all of the application domains of computing, they must be prepared to understand the language of a variety of application areas so that they can discuss problems with domain experts. This language is often that of mathematics, but in some cases, the programmers must learn a language as they work through problems with domain experts.
2. And knowledge about the library functions in the chosen programming language. When your task is to translate a mathematical formula involving the tangent function, you need to know or guess that your chosen language comes with a function such as BSL's `tan`. When, however, you need to use BSL to design image-producing functions, you will benefit from understanding the possibilities of the `2htdp/image` library.

Since you can never predict the area you will be working in, or which programming language you will have to use, it is imperative that you have a solid understanding of the full possibilities of whatever computer languages are around and suitable. Otherwise some domain expert with half-baked programming knowledge will take over your job.

You can recognize problems that demand domain knowledge from the data definitions that you work out. As long as the data definitions use classes that exist in the chosen programming language, the definition of the function body (and program) mostly relies on expertise in the domain. Later, when we introduce complex forms of data, the design of functions demands computer science knowledge.

3.4 From Functions to Programs

Not all programs consist of a single function definition. Some require several functions, many also use constant definitions. No matter what, it is always important to design each function of a program systematically, though both global constants and the presence of auxiliary functions change the design process a bit.

When you have defined global constants, your functions may use those global constants to compute results. To remind yourself of their existence, you may wish to add these constants to your templates; after all, they belong to the inventory of things that may contribute to the function definition.

Multi-function programs come about because interactive programs automatically need functions that handle key and mouse events, functions that render the state as music, and possibly more. Even batch programs may require several different functions because they perform several separate tasks. Sometimes the problem statement itself suggests these tasks; other times you will discover the need for auxiliary functions as you are in the middle of designing some function.

For these reasons, we recommend keeping around a list of needed functions or a *wish list*. Each entry on a wish list should consist of three things: a meaningful name for the function, a signature, and a purpose statement. For the design of a batch program, put the main function on the wish list and start designing it. For the design of an interactive program, you can put the event handlers, the `stop-when` function, and the scene-rendering function on the list. As long as the list isn't empty, pick a wish and design the function. If you discover during the design that you need another function, put it on the list. When the list is empty, you are done.

We owe the term “*wish list*” to Dr. John Stone.

3.5 On Testing

Testing quickly becomes a labor-intensive chore. While it is easy to run small programs in the interactions area, doing so requires a lot of mechanical labor and intricate inspections. As programmers grow their systems, they wish to conduct many tests. Soon this labor becomes overwhelming, and programmers start to neglect it. At the same time, testing is the first tool for discovering and preventing basic flaws. Sloppy testing quickly leads to buggy functions—that is, functions with hidden problems—and buggy functions retard projects, often in multiple ways.

Hence, it is critical to mechanize tests instead of performing them manually. Like many programming

languages, BSL includes a testing facility, and DrRacket is aware of this facility. To introduce this testing facility, we take a second look at the function that converts temperatures in Fahrenheit to Celsius temperatures from [Programs](#). Here is the definition, reformulated according to the design recipe:

```
; Number -> Number
; converts Fahrenheit temperatures to Celsius temperatures
; given 32, expect 0
; given 212, expect 100
; given -40, expect -40
(define (f2c f)
  (* 5/9 (- f 32)))
```

Testing the function's examples requires three computations and three comparisons between two numbers each. You can formulate these tests and add them to the definitions area in DrRacket:

```
(check-expect (f2c -40) -40)
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)
```

When you now click the *RUN* button, you see a report from BSL that the program passed all three tests—and you had nothing else to do.

In addition to getting tests to run automatically, the `check-expect` forms show another advantage when tests fail. To see how this works, change one of the above tests so that the result is wrong, for example

```
(check-expect (f2c -40) 40)
```

When you now click the *RUN* button, an additional window pops up. The window's text explains that one of three tests failed. For the failed test, the window displays three pieces: the computed value, the result of the function call (`-40`); the expected value (`40`); and a hyperlink to the text of the failed test case.

```
; Number -> Number
; converts Fahrenheit temperatures to Celsius temperatures

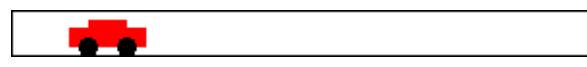
(check-expect (f2c -40) -40)
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)

(define (f2c f)
  (* 5/9 (- f 32)))
```

Figure 12: Testing in BSL

You can place `check-expect` specifications above or below the function definitions that they test. When you click *RUN*, DrRacket collects all `check-expect` specifications and evaluates them **after** all function definitions have been added to the “vocabulary” of operations. The above figure shows how to exploit this freedom to combine the example and test step. Instead of writing down the examples as comments, you can translate them directly into tests. When you're all done with the design of the function, clicking *RUN* performs the test. And if you ever change the function for some reason—to improve its look or to add a piece of functionality—the next click re-tests the function.

Last but not least, `check-expect` also works for images. That is, you can test image-producing functions. Say you wish to design the function `render`, which places the image of a car, dubbed `CAR`, into a background scene, named `BACKGROUND`. For the design of this function, you may formulate the following tests:

```
(check-expect (render 50) 

```

```
(check-expect (render 200)
```



```
)
```

Alternatively, you could formulate expressions—possibly in the interactions area of DrRacket—that compute the expected results:

```
(check-expect (render 50) (place-image CAR 50 Y-CAR BACKGROUND))
(check-expect (render 200) (place-image CAR 200 Y-CAR BACKGROUND))
```

This alternative approach helps you figure out how to express the function body and is therefore preferable.

Because it is so useful to have DrRacket conduct the tests and not to check everything yourself manually, we immediately switch to this style of testing for the rest of the book. This form of testing is dubbed *unit testing*, and BSL's unit testing framework is especially tuned for novice programmers. One day you will switch to some other programming language; one of your first tasks will be to figure out its unit testing framework.

For additional information on how to formulate examples as tests, see [BSL Tests](#).

3.6 Designing World Programs

While the previous chapter introduces the *2htdp/universe* library in an ad hoc way, this section demonstrates how the design recipe helps you create world programs systematically. It starts with a brief summary of the *2htdp/universe* library based on data definitions and function signatures. The second part spells out a design recipe for world programs, and the last one starts a series of exercises that runs through several of the next few chapters.

[Figure 13](#) presents the *2htdp/universe* library in a schematic and simplified way. The teachpack expects that a programmer develops a data definition that represents the state of the world and a function `render` that knows how to create an image for every possible state of the world. Depending on the needs of the program, the programmer must then design functions that respond to clock ticks, key strokes, and mouse events. Finally, an interactive program may need to stop when its current world belongs to a sub-class of states; `end?` recognizes these *final states*.

```
; WorldState : a data definition of your choice
; a collection of data that represents the state of the world

; render :
;   WorldState -> Image
; big-bang evaluates (render cw) to obtain image of
; current world cw when needed

; clock-tick-handler :
;   WorldState -> WorldState
; for each tick of the clock, big-bang evaluates
; (clock-tick-handler cw) for current world cw to obtain
; new world

; key-stroke-handler :
;   WorldState String -> WorldState
; for each key stroke, big-bang evaluates
; (key-stroke-handler cw ke) for current world cw and
; key stroke ke to obtain new world

; mouse-event-handler :
;   WorldState Number Number String -> WorldState
; for each manipulation of the mouse, big-bang evaluates
; (mouse-event-handler cw x y me) for current world cw,
; coordinates x and y, and mouse event me to
; obtain new world
```

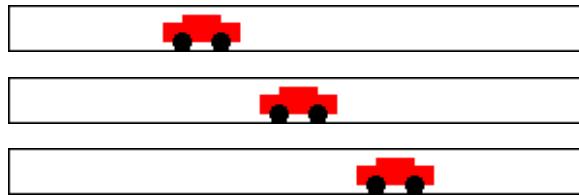
```
; end? :
;   WorldState -> Boolean
; after an event is processed, big-bang evaluates (end? cw)
; for current world cw to determine whether the program stops
```

Figure 13: Signatures for interaction functions

Assuming that you understand the rudimentary workings of `big-bang`, you can focus on the truly important problem of designing world programs. Let's construct a concrete example for the following design recipe:

Sample Problem: Design a program that moves a car across the world canvas, from left to right, at the rate of three pixels per clock tick.

For this problem statement, it is easy to imagine scenes that illustrate the domain:



In this book, we often refer to the domain of an interactive `big-bang` program as a “world,” and we speak of designing “world programs.”

The design recipe for world programs, like the one for functions, is a tool for systematically moving from a problem statement to a working program. It consists of three big steps and one small one:

1. For all those properties of the world that remain the same over time and are needed to render it as an `Image`, introduce constants. In BSL, we specify such constants via definitions. For the purpose of world programs, we distinguish between two kinds of constants:
 - a. “Physical” constants describe general attributes of objects in the world, such as the speed or velocity of an object, its color, its height, its width, its radius, and so forth. Of course these constants don't really refer to physical facts, but many are analogous to physical aspects of the real world.

In the context of our sample problem, the radius of the car's wheels and the distance between the wheels are such “physical” constants:

```
(define WIDTH-OF-WORLD 200)

(define WHEEL-RADIUS 5)
(define WHEEL-DISTANCE (* WHEEL-RADIUS 5))
```

Note how the second constant is computed from the first.

- b. Graphical constants are images of objects in the world. The program composes them into images that represent the complete state of the world.

Here are graphical constants for wheel images of our sample car:

We suggest you experiment in DrRacket's interaction area to develop graphical constants.

```
(define WHEEL (circle WHEEL-RADIUS "solid" "black"))
(define SPACE (rectangle ... WHEEL-RADIUS ... "white"))
(define BOTH-WHEELS (beside WHEEL SPACE WHEEL))
```

Graphical constants are usually computed, and the computations tend to involve the physical constants and other graphical images.

2. Those properties that change over time—in reaction to clock ticks, key strokes, or mouse actions—

give rise to the current state of the world. Your task is to develop a data representation for all possible states of the world. The development results in a data definition, which comes with a comment that tells readers how to represent world information as data and how to interpret data as information about the world.

Choose simple forms of data to represent the state of the world.

For the running example, it is the car's distance to the left margin that changes over time. While the distance to the right margin changes, too, it is obvious that we need only one or the other to create an image. A distance is measured in numbers, so the following is an adequate data definition:

```
| ; WorldState is a Number
| ; interpretation the number of pixels between the left border and the car
```

An alternative is to count the number of clock ticks that have passed and to use this number as the state of the world. We leave this design variant as an exercise.

3. Once you have a data representation for the state of the world, you need to design a number of functions so that you can form a valid **big-bang** expression.

To start with, you need a function that maps any given state into an image so that **big-bang** can render the sequence of states as images:

```
| ; render
```

Next you need to decide which kind of events should change which aspects of the world state. Depending on your decisions, you need to design some or all of the following three functions:

```
| ; clock-tick-handler
| ; key-stroke-handler
| ; mouse-event-handler
```

Finally, if the problem statement suggests that the program should stop if the world has certain properties, you must design

```
| ; end?
```

For the generic signatures and purpose statements of these functions, see [figure 13](#). You should reformulate these generic purpose statements so that you have a better idea of what your functions must compute.

In short, the desire to design an interactive program automatically creates several initial entries for your wish list. Work them off one by one and you get a complete world program.

Let us work through this step for the sample program. While **big-bang** dictates that we must design a rendering function, we still need to figure out whether we want any event handling functions. Since the car is supposed to move from left to right, we definitely need a function that deals with clock ticks. Thus, we get this wish list:

```
| ; WorldState -> Image
| ; places the image of the car x pixels from the left margin of
| ; the BACKGROUND image
| (define (render x)
|   BACKGROUND)

| ; WorldState -> WorldState
| ; adds 3 to x to move the car right
| (define (tock x)
|   x)
```

Note how we tailored the purpose statements to the problem at hand, with an understanding of how **big-bang** will use these functions.

4. Finally, you need a **main** function. Unlike all other functions, a **main** function for world programs

doesn't demand design. It doesn't even require testing because its sole reason for existing is that you can **launch** your world program conveniently from DrRacket's interaction area.

The one decision you must make concerns `main`'s arguments. For our sample problem we opt to apply `main` to the initial state of the world:

```
; WorldState -> WorldState
; launches the program from some initial state
(define (main ws)
  (big-bang ws
    [on-tick tock]
    [to-draw render]))
```

Hence, you can launch this interactive program with

```
> (main 13)
```

to watch the car drive off from 13 pixels to the right of the left margin. It will stop when you close `big-bang`'s window. Remember that `big-bang` returns the current state of the world when the evaluation stops.

Naturally, you don't have to use the name "WorldState" for the class of data that represents the states of the world. Any name will do as long as you use it consistently for the signatures of the event handling functions. Also, you don't have to use the names `tock`, `render`, or `end?`. You may name these functions whatever you like, as long as you use the same names when you write down the clauses of the `big-bang` expression. Lastly, you may have noticed that you may list the clauses of a `big-bang` expression in any order as long as you list the initial state first.

Let us now work through the rest of the program design process, using the design recipe for functions and other design concepts spelled out so far.

Exercise 41. Good programmers ensure that an image such as `CAR` can be enlarged or reduced via a single change to a constant definition. We started the development of our car image with a single plain definition:

```
(define WHEEL-RADIUS 5)
```

Good programmers establish a single point of control for all aspects of their programs, not just the graphical constants. Several chapters deal with this issue.

The definition of `WHEEL-DISTANCE` is based on the wheel's radius. Hence, changing `WHEEL-RADIUS` from `5` to `10` doubles of the car image. This kind of program organization is dubbed *single point of control*, and good design employs single point of control as much as possible.

Develop your favorite image of a car so that `WHEEL-RADIUS` remains the single point of control. Remember to experiment and make sure you can re-size the image easily. ■

Next we deal with the design of the clock tick handling function on the wish list:

```
; WorldState -> WorldState
; moves the car by three pixels every time the clock ticks
(define (tock ws) ws)
```

Since the state of the world represents the distance between the left margin of the canvas and the car, and since the car moves at three pixels per clock tick, a concise purpose statement combines these two facts into one. This makes it also easy to create examples and to define the function:

```
; WorldState -> WorldState
; moves the car by three pixels every time the clock ticks
; example:
; given: 20, expect 23
; given: 78, expect 81
(define (tock ws)
  (+ ws 3))
```

The last design step calls for tests to confirm that the examples work as expected. So we click the *RUN* button and test:

```
> (tock 20)
23
> (tock 78)
81
```

Since the results are as expected, the design of *tock* is finished.

Exercise 42. Formulate the examples as BSL tests. ▀

Our second entry on the wish list specifies a function that translates the state of the world into an image:

```
; WorldState -> Image
; places the car into a scene according to the given world state
(define (render ws)
  BACKGROUND)
```

To make examples for a rendering function, we suggest arranging a table like this:

ws =	50	
ws =	100	
ws =	150	
ws =	200	

It lists the given world states in the first column, and in the second column you see the desired scenes. For your first few rendering functions, you may just wish to draw these images by hand.

Even though this kind of image table is intuitive and explains what the running function is going to display—a moving car—it does not explain **how** the function creates this result. To get from here to there, we recommend writing down expressions that create the images in the table:

```
ws =      50      (place-image CAR 50 Y-CAR BACKGROUND)
ws =      100     (place-image CAR 100 Y-CAR BACKGROUND)
ws =      150     (place-image CAR 150 Y-CAR BACKGROUND)
ws =      200     (place-image CAR 200 Y-CAR BACKGROUND)
```

The capitalized names refer to the obvious constants: the image of a car, its fixed y-coordinate, and the background scene, which is currently empty.

This extended table suggests a pattern for the formula that goes into the body of the *render* function:

```
; WorldState -> Image
; places the car into a scene according to the given world state
(define (render ws)
  (place-image CAR ws Y-CAR BACKGROUND))
```

And that is mostly all there is to designing a simple world program.

Exercise 43. Finish the sample problem and get the program to run. That is, assuming that you have solved [exercise 41](#), define the constants *BACKGROUND* and *Y-CAR*. Then assemble all the function definitions, including their tests. When your program runs to your satisfaction, add a tree to scenery. We used

```
(define tree
  (underlay/xy (circle 10 "solid" "green")
               9 15
               (rectangle 2 20 "solid" "brown")))
```

to create a tree-like shape. Also add a clause to the *big-bang* expression that stops the animation when the car has disappeared on the right side of the canvas. ▀

After settling on a first data representation for world states, a careful programmer may have to revisit this fundamental design decision during the rest of the design process. For example, the data definition for the sample problem represents the car as a point. But (the image of) the car isn't just a mathematical point without width and height. Hence, the interpretation statement—the number of pixels from the left margin—is an ambiguous statement. Does this statement measure the distance between the left margin and the left end of the car? Its center point? Or even its right end? We ignored this issue here and leave it to BSL's image primitives to make the decision for us. If you don't like the result, revisit the data definition above and modify it or its interpretation statement to adjust to your taste.

Exercise 44. Modify the interpretation of the sample data definition so that a state denotes the x-coordinate of the right-most edge of the car. ▀

Exercise 45. Let's work through the same problem statement with a time-based data definition:

```
; AnimationState is a Number
; interpretation the number of clock ticks since the animation started
```

Like the original data definition, this one also equates the states of the world with the class of numbers. Its interpretation, however, explains that the number means something entirely different.

Design functions `tock` and `render` and develop a `big-bang` expression so that you get once again an animation of a car traveling from left to right across the world's canvas.

How do you think this program relates to the `animate` function from [Prologue: How to Program](#)?

Use the data definition to design a program that moves the car according to a sine wave. Don't try to drive like that. ▀

Dealing with mouse movements is occasionally tricky because it isn't exactly what it seems to be. For a first idea of why that is, read [On Mice and Keys](#).

Let us end the section with an illustration of mouse event handling. We also use the example as a first illustration of the advantages that a separation of view and model provide. Suppose we wish to allow people to move the car through hyperspace:

Sample Problem: Design a program that moves a car across the world canvas, from left to right, at the rate of three pixels per clock tick. **If the mouse is clicked anywhere on the canvas, the car is placed at the x-coordinate of that point.**

The bold part is the addition to the sample problem from above.

When we are confronted with a modified problem, we use the design process to guide us. If used properly, this process naturally determines what we need to add to our existing program to cope with the addition to the problem statement. So here are the four steps applied to this modified problem:

1. There are no new properties, meaning we do not need new constants.
2. Since the program is still concerned with only one property that changes over time—the x position of the car—the existing data representation suffices.
3. The revised problem statement calls for a mouse event handler, without giving up on the clock-based movement of the car. Hence, we state an appropriate wish:

```
; WorldState Number Number String -> WorldState
; places the car at the x-coordinate if me is "button-down"
(define (hyper x-position-of-car x-mouse y-mouse me)
  x-position-of-car)
```

4. Lastly, we need to modify `main` to take care of mouse events. All this requires is the addition of an `on-mouse` clause that defers to the new entry on our wish list:

```
(define (main ws)
  (big-bang ws
    [on-tick tock]
    [on-mouse hyper]
    [to-draw render]))
```

After all, the modified problem statement calls for dealing with mouse clicks and everything else remains the same.

The rest is a mere matter of designing one more function, and for that we use the design recipe for functions.

An entry on the wish list covers the first two steps of the design recipe for functions. Hence, our next step is to develop some functional examples:

```
; WorldState Number Number String -> WorldState
; places the car at the x-coordinate if me is "button-down"
; given: 21 10 20 "enter"
; wanted: 21
; given: 42 10 20 "button-down"
; wanted: 10
; given: 42 10 20 "move"
; wanted: 42
(define (hyper x-position-of-car x-mouse y-mouse me)
  x-position-of-car)
```

The examples say that if the string argument is equal to "button-down", the function returns x-mouse; otherwise it returns x-position-of-car.

Exercise 46. Formulate the examples as BSL tests. Click *RUN* and watch them fail. ▶

To complete the function definition, we must appeal to your fond memories from [Prologue: How to Program](#), specifically memories about the `conditional` form. Using `cond`, `hyper` is a two-line definition:

In the next chapter, we explain designing with `cond` in detail.

```
; WorldState Number Number String -> WorldState
; places the car at the x-coordinate if me is "button-down"
(define (hyper x-position-of-car x-mouse y-mouse me)
  (cond
    [(string=? "button-down" me) x-mouse]
    [else x-position-of-car]))
```

If you solved [exercise 46](#), re-run the program and watch all tests succeed. Assuming the tests do succeed, evaluate

```
(main 1)
```

in DrRacket's interactions area and transport your car through hyperspace.

You may wonder why this program modification is so straightforward. There are really two reasons. First, this book and its software strictly separate the data that a program tracks—the *model*—and the image that it shows—the *view*. In particular, functions that deal with events have nothing to do with how the state is rendered. If we wish to modify how a state is rendered, we can focus on the function specified in a `to-draw` clause. Second, the design recipes for programs and functions organize programs in the right way. If anything changes in a problem statement, following the design recipe a second time naturally points out where the original problem solution has to change. While this may look obvious for the simple kind of problems we are dealing with now, it is critical for the kind of problems that programmers encounter in the real world.

3.7 Virtual Pet Worlds

This exercise section introduces the first two elements of a virtual pet game. It starts with just a display of a cat that keeps walking across the canvas. Of course, all the walking makes the cat unhappy and its unhappiness shows. Like all pets, you can try petting, which helps some, or you can try feeding, which helps a lot more.

So let's start with an image of our favorite cat:



```
(define cat1 )
```

Copy the cat image and paste it into DrRacket, then give the image a name with `define`, just like above.

Exercise 47. Design a “virtual cat” world program that continuously moves the cat from left to right. Let's call it `cat-prog` and let's assume it consumes the starting position of the cat. Furthermore, make the cat move three pixels per clock tick. Whenever the cat disappears on the right it should re-appear on the left. You may wish to read up on the `modulo` function. ■

Exercise 48. Improve the cat animation with a second, slightly different image:



```
(define cat2 )
```

Adjust the rendering function from [exercise 47](#) so that it uses one cat image or the other based on whether x-coordinate is odd. Read up on `odd?` in the help desk, and use a `cond` expression to select cat images. ■

Exercise 49. Design a world program that maintains and displays a “happiness gauge.” Let's call it `gauge-prog`, and let's agree that the program consumes the maximum level of happiness. The gauge display starts with the maximum score, and with each clock tick, happiness decreases by `-0.1`; it never falls below `0`, the minimum happiness score. Every time the down arrow key is pressed, happiness increases by `1/5`; every time the up arrow is pressed, happiness jumps by `1/3`.

To show the level of happiness, we use a scene with a solid, red rectangle with a black frame. For a happiness level of 0, the red bar should be gone; for the maximum happiness level of 100, the bar should go all the way across the scene.

Note When you know enough, we will explain how to combine the gauge program with the solution of [exercise 47](#). Then we will be able to help the cat because as long as you ignore it, it becomes less happy. If you pet the cat, it becomes happier. If you feed the cat, it becomes much, much happier. So you can see why you want to know a lot more about designing world programs than these first three chapters can tell you. ■

4 Intervals, Enumerations, Itemizations

Thus far you have four choices to represent information as data: numbers, strings, images, and Boolean values. For many problems this is enough, but there are many more for which these four collections of data in BSL (or different ones in different programming languages) don't suffice. Put differently, programming with just the built-in collections of data is often clumsy and therefore error prone.

At a minimum, good programmers must learn to design programs with restrictions on these built-in collections. One way to restrict is to enumerate a bunch of elements from a collection and to say that these are the only ones that are going to be used for some problem. Enumerating elements works only when there is a finite number of them. To accommodate collections with “infinitely” many elements, we introduce intervals, which are collections of elements that satisfy a specific property.

Defining enumerations and intervals means distinguishing among different kinds of elements. To distinguish in code requires conditional functions, that is, functions that choose different ways of computing results depending on the value of some argument. Both [Many Ways To Compute](#) and [Mixing It Up with Booleans](#) illustrate with examples how to write such functions. In both, however, there is no design system; all you have is some new construct in your favorite programming language (that’s BSL) and some examples on how to use it.

Infinite may just mean “so large that enumerating the elements is entirely impractical.”

In this chapter, we introduce enumerations and intervals and discuss a general design strategy for these forms of input data. We start with a second look at the `cond` expression. Then we go through three different kinds of data descriptions: enumerations, intervals, and itemizations. An enumeration lists every single piece of data that belongs to it, while an interval specifies a range of data in one statement. The last one, itemizations, mixes the clauses of the first two, specifying ranges in one clause of its definition, and specific pieces of data in another. The chapter ends with a section on the general design strategy for such situations.

4.1 Programming with Conditionals

Recall the brief introduction to conditional expressions in [Prologue: How to Program](#). Since `cond` is the most complicated expression form in this book, let us take a close look at its general shape:

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

A `cond` expression starts with `(`, is marked with a *keyword*, and ends with `)`. Following the keyword, a programmer writes as many `cond` lines as needed; each `cond` line consists of **two** expressions, enclosed in opening and closing brackets: `[` and `]`.

The use of brackets is a convention to make `cond` lines stand out. It is perfectly acceptable to use `(...)` in place of `[...]`.

A `cond` line is also known as a *cond clause*.

Here is a function definition that uses a conditional expression:

```
(define (next traffic-light-state)
  (cond
    [(string=? "red" traffic-light-state) "green"]
    [(string=? "green" traffic-light-state) "yellow"]
    [(string=? "yellow" traffic-light-state) "red"]))
```

Like the mathematical example in [Prologue: How to Program](#), this example illustrates the convenience of using `cond` expressions. In many problem contexts, a function must distinguish several different situations. With a `cond` expression, you can use one line per possibility and thus remind the reader of the code of the different situations from the problem statement.

A note on pragmatics: Contrast `cond` expressions with `if` expressions from [Mixing It Up with Booleans](#). The latter distinguish one situation from all others. As such, `if` expressions are much less suited for multi-situation contexts; they are best used when all we wish to say is “one or the other.” We therefore

always use `cond` for situations when we wish to remind the reader of our code that some distinct situations come directly from data definitions. For other pieces of code, we use whatever construct is most convenient.

When the conditions get too complex in a `cond` expression, you occasionally wish to say something like "in all other cases." For these kinds of problems, `cond` expressions permit the use of the `else` keyword for the very last `cond` line:

```
(cond
  [ConditionExpression1 ResultExpression1]
  [ConditionExpression2 ResultExpression2]
  ...
  [else DefaultResultExpression])
```

If you make the mistake of using `else` in some other `cond` line, BSL in DrRacket signals an error:

```
> (cond
  [(> x 0) 10]
  [else 20]
  [(< x 10) 30])
cond: found an else clause that isn't the last clause in its cond expression
```

That is, BSL rejects grammatically incorrect phrases because it makes no sense to figure out what such a phrase might mean.

Imagine designing a function that, as part of a game-playing program, computes some good-bye sentence at the end of the game. Here is its header:

```
; PositiveNumber is a Number greater or equal to 0.
; PositiveNumber -> String
; computes the reward level from the given score s
```

And here are two variants for a side-by-side comparison:

<pre>(define (reward s) (cond [(<= 0 s 10) "bronze"] [(and (< 10 s) (<= s 20)) "silver"] [(< 20 s) "gold"]))</pre>	<pre>(define (reward s) (cond [(<= 0 s 10) "bronze"] [(and (< 10 s) (<= s 20)) "silver"] [else "gold"]))</pre>
---	--

On the left, this table displays a `cond` with three full-fledged conditions; on the right, the function comes with an `else` clause. To formulate the last condition for the function on the left, you must calculate that `(< 20 s)` holds if

- `s` is in `PositiveNumber` and
- `(<= 0 s 10)` as well as
- `(and (< 10 s) (<= s 20))` evaluates to `#false`.

While the calculation looks simple in this case, it is easy to make small mistakes and to introduce bugs into your program. It is therefore better to formulate the function definition as shown on the right, **if** you know that you want the exact opposite—called the *complement*—of all previous conditions in a `cond`.

4.2 Computing Conditionally

From reading the [Many Ways To Compute](#) and [Mixing It Up with Booleans](#), you roughly know how DrRacket evaluates conditional expressions. Let us go over the idea a bit more precisely for `cond`

expressions. Take another look at this definition:

```
(define (reward s)
  (cond
    [(<= 0 s 10) "bronze"]
    [(and (< 10 s) (<= s 20)) "silver"]
    [else "gold"]))
```

This function consumes a numeric score—a positive number—and produces a color.

Just looking at the `cond` expression you cannot predict which of the three `cond` clauses is going to be used. And that is the point of a function. The function deals with many different inputs, for example, 2, 3, 7, 18, 29. For each of these inputs, it may have to proceed in a different manner. Differentiating among the different classes of inputs is the purpose of the `cond` expression.

Take for example

```
(reward 3)
```

You know that DrRacket replaces function applications with the function's body after substituting the argument for the parameter:

```
==
(cond
  [(<= 0 3 10) "bronze"]
  [(and (< 10 3) (<= 3 20)) "silver"]
  [else "gold"]))
```

For a `cond` expression, DrRacket evaluates one condition at a time. For the `first` one that evaluates to `#true` it replaces the `cond` with the result expression:

```
(reward 3)
 ==
(cond
  [(<= 0 3 10) "bronze"]
  [(and (< 10 3) (<= 3 20)) "silver"]
  [else "gold"]))
 ==
(cond
  [#true "bronze"]
  [(and (< 10 3) (<= 3 20)) "silver"]
  [else "gold"])
 ==
"bronze"
```

In this particular example, the first condition—which the substitution turned into an ordinary expression in the arithmetic of Boolean values—evaluates to `#true` because 0 is less than 3 and 3 is less than 10.

Here is a second example:

```
(reward 21)
 ==
(cond
  [(<= 0 21 10) "bronze"]
  [(and (< 10 21) (<= 21 20)) "silver"]
  [else "gold"]))
 ==
(cond
  [#false "bronze"]
  [(and (< 10 21) (<= 21 20)) "silver"]
  [else "gold"])
 ==
```

```
(cond
  [(and (< 10 21) (≤ 21 20)) "silver"]
  [else "gold"])
```

Note how the first condition evaluated to `#false` this time, and as mentioned in [Many Ways To Compute](#) the entire `cond` clause is dropped. The rest of the calculation proceeds as expected:

```
...
 ==
(cond
  [(and (< 10 21) (≤ 21 20)) "silver"]
  [else "gold"])
 ==
(cond
  [(and #true (≤ 21 20)) "silver"]
  [else "gold"])
 ==
(cond
  [(and #true #false) "silver"]
  [else "gold"])
 ==
(cond
  [#false "silver"]
  [else "gold"])
 ==
(cond
  [else "gold"])
 ==
"gold"
```

Like the first condition, the second one also evaluates to `#false`, though in several steps, and so the calculation proceeds to the third `cond` line. The `else` tells DrRacket to replace the entire `cond` expression with the result expression from this clause.

Exercise 50. Enter the definition of `reward` and the application (`reward 18`) into the Definitions area of DrRacket and use the stepper to find out **how** DrRacket evaluates applications of the function. ▶

Exercise 51. A `cond` expression is really just an expression and may therefore show up in the middle of another expression:

```
(- 200 (cond
  [(> y 200) 0]
  [else y]))
```

Use the stepper to evaluate the expression for two distinct values of `y`: `100` and `210`.

Nesting `cond` expressions can eliminate common expressions. Recall the following function definition from [Prologue: How to Program](#):

```
(define (create-rocket-scene.v5 h)
  (cond
    [(≤ h ROCKET-CENTER-TO-BOTTOM)
     (place-image ROCKET 50 h MTSCN)]
    [(> h ROCKET-CENTER-TO-BOTTOM)
     (place-image ROCKET 50 ROCKET-CENTER-TO-BOTTOM MTSCN)]))
```

As you can see, both branches of the `cond` expression have the following shape:

```
(place-image ROCKET X ... MTSCN)
```

with `...` indicating where they differ.

Reformulate `create-rocket-scene.v5` to use a nested expression; the resulting function mentions `place-image` only once. ▶

4.3 Enumerations

Not all strings represent mouse events. If you looked in HelpDesk when the last section introduced the `on-mouse` clause for `big-bang`, you found out that only six strings are used to notify programs of mouse events:

```
; A MouseEvt is one of these strings:
; - "button-down"
; - "button-up"
; - "drag"
; - "move"
; - "enter"
; - "leave"
```

The interpretation of these strings is quite obvious. One of the first two strings shows up when the computer user clicks the mouse button or releases it. In contrast, the third and fourth are about moving the mouse and possibly holding down the mouse button at the same time. Finally, the last two strings represent the events of a mouse moving over the edge of the canvas: either going into the canvas from the outside or exiting the canvas.

We call it “simplistic” because it does not include the “off” state, the “blinking red” state, or the “blinking yellow” state.

More importantly, the data definition for representing mouse events as strings looks quite different from the data definitions we have seen so far. It is called an *enumeration*, and it is a data representation in which every possibility is listed. It should not come as a surprise that enumerations are common. Here is a simple one:

```
; A TrafficLight shows one of three colors:
; - "red"
; - "green"
; - "yellow"
; interpretation each element of TrafficLight represents which colored
; bulb is currently turned on
```

It is a simplistic representation of the states that a traffic light can take on. Unlike others, this data definition also uses a slightly different phrase to explain what the term `TrafficLight` means but this is an inessential difference.

Programming with enumerations is mostly straightforward. When a function’s input is a class of data whose description spells out its elements on a case-by-case basis, the function should distinguish just those cases and compute the result on a per-case basis. For example, if you wanted to define a function that computes the next state of a traffic light, given the current state as an element of `TrafficLight`, you would come up with a definition like this one:

```
; TrafficLight -> TrafficLight
; determines the next state of the traffic light from the given s

(check-expect (traffic-light-next "red") "green")

(define (traffic-light-next s)
  (cond
    [(string=? "red" s) "green"]
    [(string=? "green" s) "yellow"]
    [(string=? "yellow" s) "red"]))
```

Because the data definition for [TrafficLight](#) consists of three distinct elements, the `traffic-light-next` function naturally distinguishes between three different cases. For each case, the result expression is just another string, the one that corresponds to the next case.

Exercise 52. If you copy and paste the above function definition into the definitions area of DrRacket and click *RUN*, DrRacket highlights two of the three `cond` lines. This coloring tells you that your test cases do not cover the full `conditional`. Add enough tests to make DrRacket happy. ■

Exercise 53. Design a [big-bang](#) program that simulates a traffic light for a given duration. The program renders the state of a traffic light as a solid circle of the appropriate color, and it changes state on every clock tick. What is the most appropriate initial state? Ask your engineering friends. ■

The main idea of an enumeration is that it defines a collection of data as a **finite** number of pieces of data. Each item explicitly spells out which piece of data belongs to the class of data that we are defining. Usually, the piece of data is just shown as is; on some occasions, the item of an enumeration is an English sentence that describes a finite number of elements of pieces of data with a single phrase.

Here is an important example:

```
; A 1String is a string of length 1,
; including
; - "\\" (the backslash),
; - " " (the space bar),
; - "\t" (tab),
; - "\r" (return), and
; - "\b" (backspace).
; interpretation represents a single letter or keyboard character
```

You know that such a data definition is proper if you can describe all of its elements with a BSL test. In the case of `1String`, you can find out whether some string `s` belongs to the collection with

```
(= (string-length s) 1)
```

An alternative way to check that you have succeeded is to enumerate all the members of the collection of data that you wish to describe:

```
; A 1String is one of:
; - "q"
; - "w"
; - "e"
; - "r"
; - "t"
; - "y"
; ...
; - " "
; - "\\"
; - "\t"
; - "\r"
; - "\b"
```

If you look at your keyboard, you find ←, ↑, and such notations. Our chosen programming language, BSL, specifically the `2htdp/universe` library, uses its own data definition to represent this information: Here is an excerpt:

You know where to find the full definition.

```
; A KeyEvent is one of:
; - a single-character string, i.e., a string of length 1
; - "left"
; - "right"
; - "up"
; - "down"
; - ...
```

The first item in this enumeration describes the same bunch of strings that `1String` describes. The clauses that follow enumerate strings for special key events, such as pressing one of the four arrow keys or releasing a key.

At this point, we can actually design a key event handler systematically. Here is a sketch:

```
; WorldState KeyEvent -> ...
(define (handle-key-events w ke)
  (cond
    [(= (string-length ke) 1) ...]
    [(string=? "left" ke) ...]
    [(string=? "right" ke) ...]
    [(string=? "up" ke) ...]
    [(string=? "down" ke) ...]
    ...))
```

This event handling function uses a `cond` expression, and for each line in the enumeration of the data definition, there is one `cond` line. The condition in the first `cond` line identifies the `KeyEvents` identified in the first line of the enumeration, the second `cond` clause corresponds to the second data enumeration line, and so on.

```
; Position is a Number.
; interpretation distance between the left margin and the ball

; Position KeyEvent -> Position
; computes the next location of the ball

(check-expect (keh 13 "left") 8)
(check-expect (keh 13 "right") 18)
(check-expect (keh 13 "a") 13)

(define (keh p k)
  (cond
    [= (string-length k) 1]
    [p]
    [(string=? "left" k)
     (- p 5)]
    [(string=? "right" k)
     (+ p 5)]
    [else p]))
```

```
(define (keh p k)
  (cond
    [(string=? "left" k)
     (- p 5)]
    [(string=? "right" k)
     (+ p 5)]
    [else p]))
```

Figure 14: Conditional functions and special enumerations

When programs rely on data definitions that are defined by a programming language (such as BSL) or its libraries (such as the `2htdp/universe` library), it is common that they use only a part of the enumeration. To illustrate this point, let us look at a representative problem.

Sample Problem: Design a key-event handler that moves a red dot left or right on a horizontal line in response to pressing the left and right arrow keys.

Figure 14 presents two solutions to this problem. The function on the left is organized according to the basic idea of using one `cond` clause per line in the data definition of the input, `KeyEvent`. In contrast, the right-hand side displays a version that uses the three essential lines: two for the keys that matter and one for everything else. The re-ordering is appropriate because only two of the `cond`-lines are relevant, and they can be cleanly separated from other lines. Naturally, this kind of re-arrangement is done **after** the function is designed properly.

4.4 Intervals

Imagine yourself responding to the following sample design task:

Sample Problem: Design a program that simulates the descent of a UFO.

After a bit of thinking, you could come up with a program like the one [figure 15](#). Understand the data definition and the function definitions fully before you read on.

```
; WorldState is a Number
; interpretation height of UFO (from top)

; constants:
(define WIDTH 300)
(define HEIGHT 100)
(define CLOSE (/ HEIGHT 3))

; visual constants:
(define MT (empty-scene WIDTH HEIGHT))
(define UFO
  (overlay (circle 10 "solid" "green")
           (rectangle 40 2 "solid" "green")))

; WorldState -> WorldState
(define (main y0)
  (big-bang y0
    [on-tick nxt]
    [to-draw render]))

; WorldState -> WorldState
; computes next location of UFO

(check-expect (nxt 11) 14)

(define (nxt y)
  (+ y 3))

; WorldState -> Image
; place UFO at given height into the center of MT

(check-expect
  (render 11) (place-image UFO (/ WIDTH 2) 11 MT))

(define (render y)
  (place-image UFO (/ WIDTH 2) y MT))
```

Figure 15: UFO, descending

Before you release this "game" program, however, you may wish to add the display of status line to the canvas:

Sample Problem: Add a status line that says "descending" when the UFO's height is above one third of the height of the canvas. It switches to "closing in" below that. And finally, when the UFO has reached the bottom of the canvas, the status notifies the player that the UFO has "landed."

You are free to use appropriate colors for the status line.

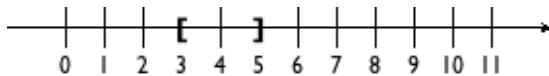
In this case, we don't have a finite enumeration of distinct elements or distinct subclasses of data. After all conceptually the interval between 0 and HEIGHT (for some number greater than 0) contains an infinite number of numbers and a large number of integers. Therefore we use intervals to superimpose some organization on the generic data definition, which just uses "numbers" to describe the class of coordinates.

An *interval* is a description of a class of (real or rational or integer) numbers via boundaries. The simplest interval has two boundaries: left and right. If the left boundary is to be included in the interval, we say it is a *closed* on the left. Similarly, a right-closed interval includes its right boundary.

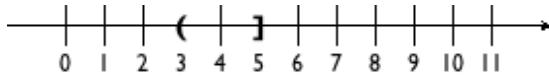
Finally, if an interval does not include a boundary, it is said to be *open* at that boundary.

Pictures of, and notations for, intervals use brackets for closed boundaries and parentheses for open boundaries. Here are four pictures of simple intervals:

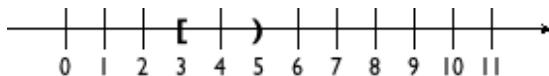
- $[3, 5]$ is a closed interval:



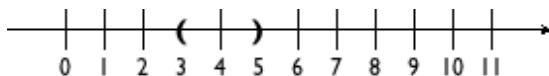
- $(3, 5]$ is a left-open interval:



- $[3, 5)$ is a right-open interval:



- and $(3, 5)$ is an open interval:



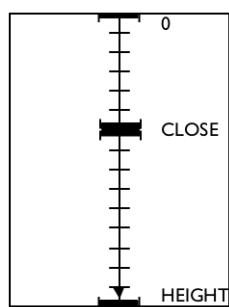
Exercise 54. Determine the integers that each of the four intervals contains. ■

The interval concept helps us formulate a data definition that captures the revised problem statement better than the "numbers" based definition:

```
; constants:
(define CLOSE (/ HEIGHT 3))

; A WorldState falls into one of three intervals:
; - between 0 and CLOSE
; - between CLOSE and HEIGHT
; - below HEIGHT
```

Specifically, there are three intervals, which we may picture as follows:



What you see is the standard number line, turned vertical and broken into intervals. Each interval starts with an angular downward-pointing bracket (\lceil) and ends with an upward-pointing bracket (\rceil).

The picture identifies three intervals in this manner:

- the upper interval goes from 0 to $CLOSE$;
- the middle one starts at $CLOSE$ and reaches $HEIGHT$; and
- the lower, invisible interval is just a single line at $HEIGHT$.

One can also think of the last interval as one that starts at $HEIGHT$ and goes on forever.

Visualizing the data definition in this manner helps with the design of functions in many ways. First, it immediately suggests how to pick examples. Clearly we want the function to work inside of all the

intervals and we want the function to work properly at the ends of each interval. Second, the image as well as the data definition tell us that we need to formulate a condition that determines whether or not some "point" is within one of the intervals.

Putting the two together also raises a question, namely, how exactly the function deals with the end points. In the context of our example, two points on the number line belong to two intervals: CLOSE belongs to both the upper interval and the middle one, while HEIGHT seems to fall into both the middle one and the lowest one. Such overlaps usually cause problems for programs, and they ought to be avoided.

BSL functions avoid them naturally due to the way `cond` expressions are evaluated. Consider this natural organization of a function that consumes elements of `WorldState`:

```
; WorldState -> WorldState
(define (f y)
  (cond
    [(<= 0 y CLOSE) ...]
    [(<= CLOSE y HEIGHT) ...]
    [(>= y HEIGHT) ...]))
```

The three `cond` lines correspond to the three intervals. Each condition identifies those values of `y` that are in between the limits of the intervals. Due to the way `cond` lines are checked one by one, however, a `y` value of CLOSE makes BSL pick the first `cond` line, and a `y` value of HEIGHT triggers the evaluation of the second *ResultExpression*.

If we wanted to make this choice obvious and immediate for every reader of our code, we would use different conditions:

```
; WorldState -> WorldState
(define (g y)
  (cond
    [(<= 0 y CLOSE) ...]
    [(and (< CLOSE y) (<= y HEIGHT)) ...]
    [(> y HEIGHT) ...]))
```

Note how the second `cond` line of `g` uses an `and` expression to combine a strictly-less check with a less-than-or-equal check instead of `f`'s `<=` with three arguments.

Given all that, we can complete the definition of the function that adds the requested status line to our UFO animation:

```
; WorldState -> Image
; adds a status line to the scene created by render

(check-expect (render/status 10)
              (place-image (text "descending" 11 "green")
                           10 10
                           (render 10)))

(define (render/status y)
  (cond
    [(<= 0 y CLOSE)
     (place-image (text "descending" 11 "green")
                  10 10
                  (render y))]
    [(and (< CLOSE y) (<= y HEIGHT))
     (place-image (text "closing in" 11 "orange")
                  10 10
                  (render y))]
    [(> y HEIGHT)
     (place-image (text "landed" 11 "red")
                  10 10
                  (render y))]))
```

```
(render y))))])
```

The function uses a `cond` expression to distinguish the three intervals. In each `cond` clause, the *ResultExpression* uses `render` (from [figure 15](#)) to create the image with the descending UFO and then places an appropriate text at position (10,10) with `place-image`.

To run this revised animation, you need to change `main` from [figure 15](#) just a tiny bit:

```
; WorldState -> WorldState
(define (main y0)
  (big-bang y0
    [on-tick nxt]
    [to-draw render/status]))
```

One aspect of this function definition might disturb you, and to clarify why, let us refine the sample problem from above just a tiny bit:

Sample Problem: Add a status line, positioned at (20,20), that says "descending" when the UFO's height is above one third of the height ...

This could be the response of a client who has watched your animation for a first time.

At this point, you have no choice but to change the function `render/status` at **three** distinct places because you have three copies of one external piece of information: the location of the status line. To avoid multiple changes for a single element, programmers try to avoid copies. You have two choices to fix this problem here. The first one is to use constant definitions, which you might recall from early chapters. The second one is to think of the `cond` expression as an expression that may appear anywhere in a function, including in the middle of some other expression:

```
; WorldState -> Image
; adds a status line to the scene create by render

(check-expect (render/status 42)
  (place-image (text "closing in" 11 "green")
    20 20
    (render 42)))

(define (render/status y)
  (place-image
    (cond
      [(<= 0 y CLOSE)
       (text "descending" 11 "green")]
      [(and (< CLOSE y) (<= y HEIGHT))
       (text "closing in" 11 "orange")]
      [(> y HEIGHT)
       (text "landed" 11 "red")])
    20 20
    (render y)))
```

In this revised definition of `render/status`, the `cond` expression is the first argument to `place-image`. As you can see, its result is always a `text` image that is placed at position (20,20) into the image created by `(render y)`.

4.5 Itemizations

An interval distinguishes different subclasses of numbers; an enumeration spells out item for item the useful elements of an existing class of data. Data definitions that use *itemizations* generalize intervals and enumerations. They allow the combination of any existing data classes (defined elsewhere) with each other and with individual pieces of data.

Here is an obvious example, a rewrite of an important example from [Enumerations](#):

```
; A KeyEvent is one of:
; - 1String
; - "left"
; - "right"
; - "up"
; - "down"
; - ...
```

In this case, the `KeyEvent` data definition refers to the `1String` data definition. Since functions that deal with `KeyEvents` often deal with `1Strings` separately from the rest and do so with auxiliary functions, we now have a convenient way to express signatures for these functions, too.

The description of the `string->number` primitive, which we used before, employs the idea of an itemization in a sophisticated way. Its signature is

```
; String -> NorF
; converts the given string into a number;
; produces #false if impossible
(define (string->number s) (... s ...))
```

meaning that the result signature names a simple class of data:

```
; A NorF is one of:
; - #false
; - a Number
```

This itemization combines one piece of data (`#false`) with a large class of data (`Number`).

Now imagine a function that consumes the result of `string->number` and adds `3`, dealing with `#false` as if it were `0`:

```
; NorF -> Number
; adds 3 to the given number; 3 otherwise

(check-expect (add3 #false) 3)
(check-expect (add3 0.12) 3.12)

(define (add3 x)
  (cond
    [(false? x) 3]
    [else (+ x 3)]))
```

As above, the function's body consists of a `cond` expression with as many clauses as there are items in the enumeration of the data definition. The first `cond` clause recognizes when the function is applied to `#false`; the corresponding result is `3` as requested. The second clause is about numbers and adds `3` as required.

Let's solve a somewhat more purposeful design task:

Sample Problem: Design a program that launches a rocket when the user of your program presses the space bar. The program first displays the rocket sitting at the bottom of the canvas. Once launched, it moves upward at three pixels per clock tick.

This revised version suggests a data definition that represents two classes of states:

```
; A LR (short for: launching rocket) is one of:
; - "resting"
; - non-negative number
; interpretation "resting" represents a rocket on the ground
; a number denotes the height of a rocket in flight
```

While the interpretation of "resting" is obvious, the interpretation of numbers is a bit ambiguous. Any given number could refer to at least two different notions of "height":

1. the word "height" could refer to the distance between the ground and the rocket's point of reference, say, its center; or
2. it could mean the distance between the top of the canvas and the reference point.

Either one works fine. The second one uses the conventional computer meaning of the word "height." It is thus slightly more convenient for functions that translate the state of the world into an image, and we therefore choose to interpret the number in that spirit.

To drive home this choice, [exercise 59](#) below asks you to solve the exercises of this section using the first interpretation of "height."

Exercise 55. The design recipe for world programs demands that you translate information into data and vice versa to ensure a complete understanding of the data definition. In some way it is best to draw some world scenarios and to represent them with data and, conversely, to pick some data examples and to draw pictures that match them. Do so for the [LR](#) definition, including at least HEIGHT and [0](#) as examples. ■

In reality, rocket launches come with count-downs:

Sample Problem: Design a program that launches a rocket when the user presses the space bar. At that point, the simulation starts a count-down for three ticks, before it displays the scenery of a rising rocket. The rocket should move upward at a rate of three pixels per clock tick.

Following the design recipe for world programs from [Designing World Programs](#), we first collect constants:

```
; physical constants
(define HEIGHT 300)
(define WIDTH 100)
(define YDELTA 3)

; graphical constants
(define BACKG (empty-scene WIDTH HEIGHT))
(define ROCKET (rectangle 5 30 "solid" "red"))
```

The YDELTA constant describes how fast the rocket moves along the y-axis, as specified in the problem statement; HEIGHT and WIDTH are arbitrary as is the graphical constant for the rocket.

Next we turn to the development of a data definition. This revision of the problem clearly calls for three distinct sub-classes of states:

```
; A LRCD (short for: launching rocket count down) is one of:
; - "resting"
; - a number in [-3, -1]
; - a non-negative number
; interpretation a rocket resting on the ground, in count-down mode,
; or the number of pixels from the top i.e. its height
```

The second, new sub-class of data—three negative numbers—represents the world after the user pressed the space bar and before the rocket lifts off.

At this point, we write down our wish list for a function that renders states as images and for any event-handling functions that we may need:

```
; LRCD -> Image
; renders the state as a resting or flying rocket
(define (show x)
  BACKG)
```

```
; LRCD KeyEvent -> LRCD
; starts the count-down when space bar is pressed,
; if the rocket is still resting
(define (launch x ke)
  x)

; LRCD -> LRCD
; raises the rocket by YDELTA,
; if it is moving already
(define (fly x)
  x)
```

Remember that the design recipe for world programs—specifically, the mandates of the *2htdp/universe* library—dictates these signatures, though the choice of names for the data collection and the event handlers are ours. Also, we have specialized the purpose statements to fit our problem statement.

From here, we use the design recipe for functions to create complete definitions for all three of them, starting with examples for the first one:

```
(check-expect
  (show "resting")
  (place-image ROCKET
    10 (- HEIGHT (/ (image-height ROCKET) 2))
    BACKG))

(check-expect
  (show -2)
  (place-image (text "-2" 20 "red")
    10 (* 3/4 WIDTH)
    (place-image ROCKET
      10 (- HEIGHT (/ (image-height ROCKET) 2))
      BACKG)))

(check-expect
  (show 53)
  (place-image ROCKET 10 53 BACKG))
```

As before in this chapter, we make one test per subclass in the data definition. The first example shows the resting state, the second the middle of a count down, and the last one the rocket in flight. Furthermore, we express the expected values as expressions that draw appropriate images. We used DrRacket's interaction area to create these images; what would you do?

A first look at the examples suggests the introduction of at least one additional constant:

```
(define ROCKET-CENTER (/ (image-height ROCKET) 2))
```

This value is used at least twice to place the rocket at the appropriate place, namely on the ground of the empty scene.

A second look at the examples reveals that making examples also means making choices. Nothing in the problem statement actually demands how exactly the rocket is displayed before it is launched but doing so is natural. Similarly, nothing says to display a number during the count down, but it adds a nice touch. Last but not least, if you solved [exercise 55](#) you also know that `HEIGHT` and `0` are special points for the third clause of the data definition.

In general, intervals deserve special attention when you make up examples, that is, they deserve at least three kinds of examples: one from each end and another one from inside. Since the second subclass of `LRCD` is a (finite) interval and the third one is a half-open interval, let us take a closer look at their end points:

- Clearly, `(show -3)` and `(show -1)` must produce images like the one for `(show -2)`. After all, the rocket still rests on the ground, even if the count down numbers differ.

- The case for `(show HEIGHT)` is different. According to our agreement, `HEIGHT` represents the state when the rocket has just been launched. Pictorially this means the rocket is still resting on the ground. Based on the last test case above, here is the test case that expresses this insight:

```
(check-expect
  (show HEIGHT)
  (place-image ROCKET 10 HEIGHT BACKG))
```

Except that if you evaluate the “expected value” expression by itself in DrRacket’s interaction area, you see that the rocket is half-way underground. This shouldn’t be the case of course, meaning we need to adjust this test case and the above:

```
(check-expect
  (show HEIGHT)
  (place-image ROCKET 10 (- HEIGHT ROCKET-CENTER) BACKG))

(check-expect
  (show 53)
  (place-image ROCKET 10 (- 53 ROCKET-CENTER) BACKG))
```

- Finally, determine the result you now expect from `(show 0)`. It is a simple but revealing exercise.

Following the precedents in this chapter, `show` uses a `cond` expression to deal with the three clauses of the data definition:

```
(define (show x)
  (cond
    [(string? x) ...]
    [(<= -3 x -1) ...]
    [(>= x 0) ...]))
```

Each clause identifies the corresponding subclass with a precise condition: `(string? x)` picks the first subclass, which consists of just one element, the string "resting"; `(<= -3 x -1)` completely describes the second subclass of data; and `(>= x 0)` is a test for all non-negative numbers.

Exercise 56. Why is `(string=? "resting" x)` **incorrect** as the first condition in `show`? Conversely, formulate a completely accurate condition, that is, a `Boolean` expression that evaluates to `#true` precisely when `x` belongs to the first subclass of `LRCD`. ■

Combining the examples and the above skeleton of the `show` function yields a complete definition in a reasonably straightforward manner:

```
(define (show x)
  (cond
    [(string? x)
     (place-image ROCKET 10 (- HEIGHT ROCKET-CENTER) BACKG)]
    [(<= -3 x -1)
     (place-image (text (number->string x) 20 "red")
                 10 (* 3/4 WIDTH)
                 (place-image ROCKET
                             10 (- HEIGHT ROCKET-CENTER)
                             BACKG))]
    [(>= x 0)
     (place-image ROCKET 10 (- x ROCKET-CENTER) BACKG)])))
```

Indeed, this way of defining functions is highly effective and is an essential element of the full-fledged design recipe that we are developing in this book.

Stop! Do you notice all the occurrences of `10` in the code? It always refers to the x-coordinate of the rocket. So go ahead, create a constant definition for this `10`. Now read on.

Exercise 57. Take a second look at `show`. An expression of the shape

```
(place-image ROCKET 10 (- ... ROCKET-CENTER) BACKG)
```

appears three different times in the function: twice to draw a resting rocket and once to draw a flying rocket. Define an auxiliary function that performs this work and thus shorten show. Why is this a good idea? We discussed this idea in [Prologue: How to Program](#).

Let us move on to the second function, which deals with the key event to launch the rocket. We have its header material, so we formulate examples as tests:

```
(check-expect (launch "resting" " ") -3)
(check-expect (launch "resting" "a") "resting")
(check-expect (launch -3 " ") -3)
(check-expect (launch -1 " ") -1)
(check-expect (launch 33 " ") 33)
(check-expect (launch 33 "a") 33)
```

An inspection of these six examples shows that the first two are about the first subclass of [LRCD](#), the third and fourth concern the count-down, and the last two are about key events when the rocket is already in the air.

Since writing down the sketch of a `cond` expression worked well for the design of the `show` function, we do it again:

```
(define (launch x ke)
  (cond
    [(string? x) ...]
    [(<= -3 x -1) ...]
    [(>= x 0) ...]))
```

Looking back at the examples suggests that nothing changes when the world is in a state that is represented by the second or third subclass of data. Put differently, it is obvious that `launch` should produce `x`—the current state of the world—when this happens:

```
(define (launch x ke)
  (cond
    [(string? x) ...]
    [(<= -3 x -1) x]
    [(>= x 0) x]))
```

Finally, the first example identifies the exact case when the `launch` function produces a new world state:

```
(define (launch x ke)
  (cond
    [(string? x) (if (string=? " " ke) -3 x)]
    [(<= -3 x -1) x]
    [(>= x 0) x]))
```

Specifically, when the state of the world is "`resting`" and the user presses the space bar, the function starts the count-down with `-3`.

Copy the code into the definitions area of DrRacket and ensure that the above definitions work. At that point, you may wish to add a function for running the program:

```
; LRCD -> LRCD
(define (main1 s)
  (big-bang s
    [to-draw show]
    [on-key launch]))
```

This function does **not** specify what to do when the clock ticks; after all, we haven't designed `fly` yet. Still, with `main1` it is possible to run this incomplete version of the program and to check that you can start the count-down. What would you provide as the argument in a call to `main1`?

```

; LRCD -> LRCD
; raises the rocket by YDELTA if it is moving already

(check-expect (fly "resting") "resting")
(check-expect (fly -3) -2)
(check-expect (fly -2) -1)
(check-expect (fly -1) HEIGHT)
(check-expect (fly 10) (- 10 YDELTA))
(check-expect (fly 22) (- 22 YDELTA))

(define (fly x)
  (cond
    [(string? x) x]
    [(<= -3 x -1) (if (= x -1) HEIGHT (+ x 1))]
    [(>= x 0) (- x YDELTA)])))

```

Figure 16: Launching a count-down and a lift-off

The design of `fly`—the clock-tick handler—proceeds just like the design of the preceding two functions, and the above figure displays the result of the design process. Once again the key is to cover the space of possible input data with a good bunch of examples, especially for the two intervals. These examples ensure that the count-down and the transition from the count-down to the lift-off work properly.

Exercise 58. Define `main2` so that you can launch the rocket and watch it lift off. Read up on the `on-tick` clause to determine the length of one tick and how to change it.

If you watch the entire launch, you will notice that once the rocket reaches the top, something curious happens. Explain. Add a `stop-when` clause to `main2` so that the simulation of the lift-off stops gracefully when the rocket is out of sight. ▀

In solving [exercise 58](#) you now have a complete, working program but one that behaves a bit strangely. Experienced programmers tell you that using negative numbers to represent the count-down phase is too “brittle.” The next chapter introduces the means to provide a good data definition for this problem. Before we go there, however, let us reiterate in the following section how to design programs that consume data described by itemizations.

Exercise 59. Recall that the word “height” forced us to choose one of two possible interpretation. Now that you have solved the exercises in this section, solve them again using the first interpretation of the word. Compare and contrast the solutions. ▀

4.6 Designing with Itemizations

What the preceding three sections have clarified is that the design of functions can and must exploit the organization of the data definition. Specifically, if a data definition singles out certain pieces of data or specifies ranges of data, then the creation of examples and the organization of the function reflects these cases and ranges.

In this section, we refine the design recipe of [From Functions to Programs](#) so that you can proceed in a systematic manner when you encounter problems concerning functions that consume itemizations, including enumerations and intervals. To keep the explanation grounded, we illustrate the six design steps with the following, somewhat simplistic example:

Sample Problem: The state of Tax Land has created a three-stage sales tax to cope with its budget deficit. Inexpensive items, those costing less than \$1,000, are not taxed. Luxury items, with a price of more than \$10,000, are taxed at the rate of eight percent (8.00%). Everything in between comes with a five percent (5%) mark up.

Design a function for a cash register that given the price of an item, computes the sales tax.

So keep this problem in mind as we reconsider the steps of our design recipe:

- When the problem statement distinguishes different classes of input information, you need carefully formulated data definitions.

A data definition must use distinct *clauses* for each different subclasses of data or in some cases just individual pieces of data. Each clause specifies a data representation for a particular subclass of information. The key is that each subclass of data is distinct from every other class so that our function can proceed by analyzing disjoint cases.

Our sample problem deals with prices and taxes, which are usually positive numbers. It also clearly distinguishes three ranges of positive numbers:

```
; A Price falls into one of three intervals:
; - 0 through 1000;
; - 1000 through 10000;
; - 10000 and above.
; interpretation the price of an item
```

Make sure you understand how these three ranges relate to the original problem.

- As far as the signature, purpose statement, and function header are concerned, you proceed as before.

Here is the material for our running example:

```
; Price -> Number
; computes the amount of tax charged for price p
(define (sales-tax p) 0)
```

- For functional examples, however, it is imperative that you pick at least one example from each subclass in the data definition. Also, if a subclass is a finite range, be sure to pick examples from the boundaries of the range and from its interior.

Since our sample data definition involves three distinct intervals, let us pick all boundary examples and one price from inside each interval and determine the amount of tax for each: 0, 537, 1000, 1282, 10000, and 12017.

Stop! Before you read on, try to calculate the tax for each of these prices.

Here is our first attempt:

0.00	537.00	1000.00	1282.00	10000.00	12017.00
0.00	0.00	????.???	64.10	?????.???	961.36

The question marks point out that the problem statement uses the somewhat vague phrase “those costing less than \$1,000” and “more than \$10,000” to specify the tax table. While a programmer may immediately jump to the conclusion that these words mean “strictly less” or “strictly more,” the lawmakers may have meant to say “less or equal” or “more or equal,” respectively. Being skeptical, we decide here that Tax Land legislators always want more money to spend, so the tax rate for \$1,000 is 5% and the rate for \$10,000 is 8%. A programmer in a tax company would have to ask the tax-law specialist in the company.

Now that we have figured out how the boundaries are to be interpreted in the domain, we could refine the data definition. We trust you can do this on your own.

Before we go, let us turn some of the examples into test cases:

```
(check-expect (sales-tax 537) 0)
(check-expect (sales-tax 1000) (* 0.05 1000))
(check-expect (sales-tax 12017) (* 0.08 12017))
```

Take a close look. Instead of just writing down the expected result, we write down how to compute the expected result. This makes it easier later to formulate the function definition.

Stop! Write down the remaining test cases. Think about why you may need more test cases than subclasses in the data definition.

4. The biggest novelty is the conditional template. In general,

the template mirrors the organization of subclasses with a `cond`.

This slogan means two concrete things. First, the function's body must be a conditional expression with as many clauses as there are distinct subclasses in the data definition. If the data definition mentions three distinct subclasses of input data, you need three `cond` clauses; if it has seventeen subclasses, the `cond` expression contains seventeen clauses. Second, you must formulate one condition expression per `cond` clause. Each expression involves the function parameter and identifies one of the subclasses of data in the data definition.

```
; Price -> Number
; computes the amount of tax charged for price p
(define (sales-tax p)
  (cond
    [(and (<= 0 p) (< p 1000)) ...]
    [(and (<= 1000 p) (< p 10000)) ...]
    [(>= p 10000) ...]))
```

5. When you have finished the template, you are ready to define the function. Given that the function body already contains a schematic `cond` expression, it is natural to start from the various `cond` lines. For each `cond` line, you may assume that the input parameter meets the condition and you exploit the corresponding test cases. To formulate the corresponding result expression, you write down the computation for this example as an expression that involves the function parameter. Ignore all other possible kinds of input data when you work on one line; the other `cond` clauses take care of those.

```
; Price -> Number
; computes the amount of tax charged for price p
(define (sales-tax p)
  (cond
    [(and (<= 0 p) (< p 1000)) 0]
    [(and (<= 1000 p) (< p 10000)) (* 0.05 p)]
    [(>= p 10000) (* 0.08 p)]))
```

5. Finally, you run the tests and make sure that the tests cover all `cond` clauses.

What do you do when one of your test cases fails? Review at the end of [Designing Functions](#) concerning test failures.

Exercise 60. Introduce constant definitions that separate the intervals for low prices and luxury prices from the others so that the legislator in Tax Land can easily raise the taxes even more. ■

4.7 Finite State Worlds

Let us exploit our knowledge to create a world that simulates a basic US traffic light. When the light is green and it is time to stop the traffic, the light turns yellow and after that turns red. When the light is red and it is time to get the traffic going, the light switches to green.

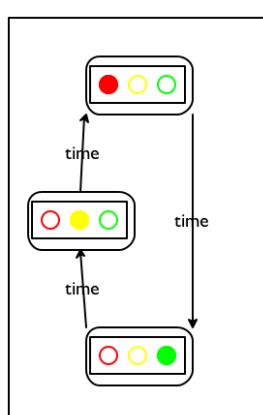


Figure 17: How a traffic light functions

Figure 17 summarizes this description as a *state transition diagram*. Such a diagram consists of *states* and arrows that connect these states. Each state depicts a traffic light in one particular configuration: red, yellow, or green. Each arrow shows how the world can change, from which state it can *transition* to another state. Our sample diagram contains three arrows, because there are three possible ways in which the traffic light can change. Labels on the arrows indicate the reason for changes; a traffic light transitions from one state to another as time passes.

In many situations, state transition diagrams have only a finite number of states and arrows. Computer scientists call such diagrams *finite state machines* or *automata*, for short: FSA. While FSAs look simple at first glance, they play an important role in computer science.

To create a world program for an FSA, we must first pick a data representation for the possible “states of the world,” which, according to [Designing World Programs](#), represents those aspects of the world that may change in some ways as opposed to those that remain the same. In the case of our traffic light, what changes is the color of the light, that is, which bulb is turned on. The size of the bulbs, their arrangement (horizontal or vertical), and other aspects don’t change. Since there are only three states—red, yellow, and green—we pick three strings, as shown in the data definition of [TrafficLight](#).

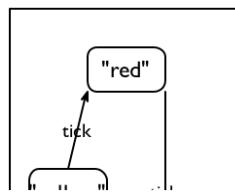


Figure 18: How to represent a traffic light

Figure 18 is a diagrammatic interpretation of the [TrafficLight](#) data definition. Like the diagram in [figure 17](#), it consists of three states, arranged in such a way that it is easy to view each data element as a representation of a concrete configuration. Also, the arrows are now labeled with `tick` to suggest that our world program uses the passing of time as the trigger that changes the state of the traffic light. If we wanted to simulate a manually operated pedestrian light, we might choose transitions based on key strokes.

Now that we know how to represent the states of our world, how to go from one to the next, and that the state changes at every tick of the clock, we can write down the signature, a purpose statement, and a stub for the two functions we must design:

```

; TrafficLight -> TrafficLight
; determines the next state of the traffic light, given current-state
(define (tl-next current-state) current-state)

; TrafficLight -> Image
; renders the current state of the traffic light as an image
(define (tl-render current-state) (empty-scene 100 30))
  
```

Preceding sections use the names `render` and `next` to name the functions that translate a state of the world into an image and that deal with clock ticks. Here we prefix these names with some syllable that suggests to which world the functions belong. Because the specific functions have appeared before, we leave them as exercises.

Exercise 61. Finish the design of a world program that simulates the traffic light FSA. Here is the main function:

```

; TrafficLight -> TrafficLight
  
```

```
; simulates a traffic light that changes with each clock tick
(define (traffic-light-simulation initial-state)
  (big-bang initial-state
    [to-draw tl-render]
    [on-tick tl-next 1]))
```

The function's argument is the initial state for the `big-bang` expression, which tells DrRacket to re-draw the state of the world with `tl-render` and to handle clock ticks with `tl-next`. Also note it informs the computer that the clock should tick once per second.

Complete the design of `tl-render` and `tl-next`.

Hint Copy the data definition for `TrafficLight` and the definitions of `tl-next` and `tl-render` into DrRacket's definition area.

Here are some test cases for the design of the latter:

```
(check-expect (tl-render "red") 
)
(check-expect (tl-render "yellow") 
)
(check-expect (tl-render "green") )
```

You may design a function that uses these images directly. If you decide to create images with the functions from the `2htdp/image` library, design an auxiliary function for creating the image of a one-color bulb. Then read up on the `place-image` function, which helps you place bulbs at the proper place.

Exercise 62. An alternative data representation for a traffic light program may use numbers instead of strings:

```
; A N-TrafficLight shows one of three colors:
; - 0
; - 1
; - 2
; interpretation 0 means the traffic light shows red,
; 1 green, and 2 yellow
```

Re-draw the transition diagram from [figure 18](#) for `N-TrafficLight`.

Study the following variant of `tl-next`:

```
; N-TrafficLight -> N-TrafficLight
; determines the next state of the traffic light, given current-state
(define (tl-next-numeric current-state)
  (modulo (+ current-state 1) 3))
```

Re-formulate the test cases for `tl-next` for `tl-next-numeric`.

Does the `tl-next` function convey its intention more clearly than the `tl-next-numeric` function? If so, why? If not, why not?

Exercise 63. As [From Functions to Programs](#) says, programs must define (unusual) constants and use names instead of actual constants. In this spirit, a proper data representation for traffic lights introduces three constant definitions and uses the constant's names:

```
(define RED 0)
(define GREEN 1)
(define YELLOW 2)

; A S-TrafficLight shows one of three colors:
; - RED
; - GREEN
; - YELLOW
```

If the names are chosen properly, such a data definition does not need an interpretation statement.

The `equal?` function compares two arbitrary values, regardless of what these values are. Equality is a complicated topic in the world of programming.

Given this specific data definition, you might define two different functions that switch the state of a traffic light in a simulation program:

```
; S-TrafficLight -> S-TrafficLight
; determines the next state of the traffic light, given cs

(define (tl-next-symbolic cs)      (define (tl-next-symbolic cs)
  (modulo (+ cs 1) 3))           (cond
                                    [(equal? cs RED) GREEN]
                                    [(equal? cs GREEN) YELLOW]
                                    [(equal? cs YELLOW) RED]))
```

Which of the two is properly designed using the recipe for itemization? Which of the two continues to work if you change the constants to

```
(define RED "red")
(define GREEN "green")
(define YELLOW "yellow")
```

Does this help you answer the questions?

Here is another finite state problem that introduces a few additional complications:

Sample Problem: Design a world program that simulates the working of a door with an automatic door closer. If this kind of door is locked, you can unlock it with a key. An unlocked door is closed but someone pushing at the door opens it. Once the person has passed through the door and lets go, the automatic door takes over and closes the door again. When a door is closed, it can be locked again.

To tease out the essential elements, we again draw a transition diagram; see the left-hand side of the figure. Like the traffic light, the door has three distinct states: locked, closed, and open. Locking and unlocking are the activities that cause the door to transition from the locked to the closed state and vice versa. As for opening an unlocked door, we say that one needs to **push** the door open. The remaining transition is unlike the others, because it doesn't require any activities by anyone or anything else. Instead, the door closes automatically over time. The corresponding transition arrow is labeled with ***time*** to emphasize this.

Figure 19: A transition diagram for a door with an automatic closer

Following our recipe, we start with a translation of the three real-world states into BSL data:

```
; A DoorState is one of:
; - "locked"
; - "closed"
```

```
; - "open"
```

That is, we use three strings to represent the three states, and the interpretation of these strings is natural.

The next step of a world design demands that we translate the actions in our domain—the arrows in the left-hand diagram—into interactions with the computer that the *2htdp/universe* library can deal with. Our pictorial representation of the door's states and transitions, specifically the arrow from `open` to `closed` suggests the use of a function that simulates time. For the other three arrows, we could use either keyboard events or mouse clicks or both. Let us use three keystrokes: `"u"` for unlocking the door, `"l"` for locking it, and the space bar `" "` for pushing it open. The right-hand side diagram expresses these choices graphically; it translates the above state-machine diagram from the world of information into the world of data and function suggestions.

Once we have decided to use the passing of time for one action and key presses for the others, we must design functions that transform the current state of the world—represented as `DoorState`—into the next state of the world. Put differently, we have just created a wish list with two handler functions that have the following signature and purpose statements:

- `door-closer`, which closes the door during one tick;
- `door-actions`, which manipulates the door in response to pressing a key; and
- `door-render`, which translates the current state of the door into an image.

The last wish comes from our desire to render states as images in our simulation.

We start with `door-closer`. Since `door-closer` acts as the `on-tick` handler, we get its signature from our choice of `DoorState` as the collection of world states:

```
; DoorState -> DoorState
; closes an open door over the period of one tick
(define (door-closer state-of-door) state-of-door)
```

Making up examples is trivial when the world can only be in one of three states. Here we use a table to express the basic idea, just like in some of the mathematical examples before:

given state	desired state
<code>"locked"</code>	<code>"locked"</code>
<code>"closed"</code>	<code>"closed"</code>
<code>"open"</code>	<code>"closed"</code>

This table can easily be expressed as BSL examples:

```
(check-expect (door-closer "locked") "locked")
(check-expect (door-closer "closed") "closed")
(check-expect (door-closer "open") "closed")
```

The template step demands a conditional with three clauses:

```
(define (door-closer state-of-door)
  (cond
    [(string=? "locked" state-of-door) ...]
    [(string=? "closed" state-of-door) ...]
    [(string=? "open" state-of-door) ...]))
```

and the process of turning this template into a function definition is dictated by the examples:

```
(define (door-closer state-of-door)
  (cond
    [(string=? "locked" state-of-door) "locked"]
    [(string=? "closed" state-of-door) "closed"]
    [(string=? "open" state-of-door) "closed"]))
```

Don't forget to run the example-tests.

The second function, `door-actions`, takes care of the remaining three arrows of the diagram. Functions that deal with keyboard events consume both a world and a key event, meaning the signature is as follows:

```
; DoorState KeyEvent -> DoorState
; simulates the actions on the door via three kinds of key events
(define (door-actions s k) s)
```

As for `door-closer`, we summarize the examples in a table:

given state	given key event	desired state
"locked"	"u"	"closed"
"closed"	"l"	"locked"
"closed"	" "	"open"
"open"	—	"open"

The examples combine information from our drawing with the choices we made about mapping actions to keyboard events. Unlike the table of examples for traffic light, this table is incomplete. Think of some other examples; then consider why our table suffices.

From here, it is straightforward to create a complete design:

```
(check-expect (door-actions "locked" "u") "closed")
(check-expect (door-actions "closed" "l") "locked")
(check-expect (door-actions "closed" " ") "open")
(check-expect (door-actions "open" "a") "open")
(check-expect (door-actions "closed" "a") "closed")

(define (door-actions s k)
  (cond
    [(and (string=? "locked" s) (string=? "u" k)) "closed"]
    [(and (string=? "closed" s) (string=? "l" k)) "locked"]
    [(and (string=? "closed" s) (string=? " " k)) "open"]
    [else s]))
```

Note the use of `and` to combine two conditions: one concerning the current state of the door and the other concerning the given key event.

Last but not least we need a function that renders the current state of the world as a scene. For simplicity, let us just use a large text for this purpose:

```
; DoorState -> Image
; translates the current state of the door into a large text

(check-expect (door-render "closed") (text "closed" 40 "red"))

(define (door-render s)
  (text s 40 "red"))
```

Here is how we run the program:

```
; DoorState -> DoorState
; simulates a door with an automatic door closer
(define (door-simulation initial-state)
  (big-bang initial-state
    [on-tick door-closer]
    [on-key door-actions]
    [to-draw door-render]))
```

Now it is time for you to collect the pieces and run them in DrRacket to see whether it all works.

Exercise 64. During a door simulation the “open” state is barely visible. Modify `door-simulation` so that the clock ticks once every three seconds. Re-run the simulation.

```

; A DoorState is one of:
; - "locked"
; - "closed"
; - "open"

; DoorState -> DoorState
; simulates a door with an automatic door closer
(define (door-simulation initial-state)
  (big-bang initial-state
    [on-tick door-closer]
    [on-key door-actions]
    [to-draw door-render])))

; DoorState -> DoorState
; closes an open door over the period of one tick

(check-expect (door-closer "locked") "locked")
(check-expect (door-closer "closed") "closed")
(check-expect (door-closer "open") "closed")

(define (door-closer state-of-door)
  (cond
    [(string=? "locked" state-of-door) "locked"]
    [(string=? "closed" state-of-door) "closed"]
    [(string=? "open" state-of-door) "closed"]))

; DoorState KeyEvent -> DoorState
; simulates actions on the door via three key events

(check-expect (door-actions "locked" "u") "closed")
(check-expect (door-actions "closed" "l") "locked")
(check-expect (door-actions "closed" " ") "open")
(check-expect (door-actions "open" "a") "open")
(check-expect (door-actions "closed" "a") "closed")

(define (door-actions s k)
  (cond
    [(and (string=? "locked" s) (string=? "u" k)) "closed"]
    [(and (string=? "closed" s) (string=? "l" k)) "locked"]
    [(and (string=? "closed" s) (string=? " " k)) "open"]
    [else s]))

; DoorState -> Image
; renders the current state of the door as a large red text

(check-expect (door-render "closed")
              (text "closed" 40 "red"))

(define (door-render s)
  (text s 40 "red"))

```

Figure 20: A door-simulation program

5 Adding Structure

Mathematicians know tricks that “merge” two numbers into a single number such that it is possible to retrieve the original ones. Programmers consider these kinds of tricks evil, because they obscure a program’s true intentions.

Suppose you want to design a world program that simulates a ball bouncing back and forth on a straight vertical line between the floor and ceiling of some imaginary, perfect room. Assume that it always moves two pixels per clock tick. If you follow the design recipe, your first goal is to develop a data representation for what changes over time. Here, the ball's location and its direction change over time, but that's **two** values while **big-bang** keeps track of just one. Thus the question arises how one piece of data can represent two changing quantities of information.

Here is another scenario that raises the same question. Your cell phone is mostly a few million lines of software wrapped in some plastic. Among other things, it administrates your list of contacts. Each contact comes with a name, a phone number, an email address, and perhaps some other information. When you have lots of contacts, each single contact is best represented as a single piece of data; otherwise the various pieces could get mixed up by accident.

Because of such programming problems, every programming language provides some mechanism for combining several pieces of data into a single piece of *compound data* and ways to retrieve the constituent values when needed. This chapter introduces BSL's mechanics, so-called structure type definitions, and how to design programs that work on compound data.

5.1 posn Structures

A location on a world canvas is uniquely identified by two pieces of data: the distance from the left margin and the distance from the top margin. The first is called an *x-coordinate* and the second one is the *y-coordinate*.

DrRacket, which is basically a BSL program, represents such locations with `posn` structures. A `posn` structure combines two numbers into a single value. We can create a `posn` structure with the operation `make-posn`, which consumes two numbers and makes a `posn`. For example,

```
(make-posn 3 4)
```

is an expression that creates a `posn` structure whose x-coordinate is `3` and whose y-coordinate is `4`.

A `posn` structure has the same status as a number or a Boolean or a string. In particular, both primitive operations and functions may consume and produce structures. Also, a program can name a `posn` structure:

```
(define one-posn (make-posn 8 6))
```

Stop! Describe `one-posn` in terms of coordinates.

Two built-in primitives are important for dealing with `posn`s. One is `posn-x`, the other is `posn-y`. Both consume `posn` structures and produce numbers. We say that `posn-x` selects the x-coordinate of a `posn` structure; for example, `(posn-x one-posn)` produces `8`. Before we discuss anything else though, let's take a look at the laws of computation for `posn` structures. That way, we can both create functions that process `posn` structures and predict what they compute.

5.2 Computing with `posns`

While functions and the laws of functions are completely familiar from pre-algebra, `posn` structures appear to be a new idea. At the same time, the concept of a `posn` ought to look like something you might have encountered before. Indeed, they are just data representations of Cartesian points or positions in the plane.

Figure 21: A Cartesian point

Selecting a Cartesian point's pieces is also a familiar process. For example, when a teacher says “take a look at the graph nearby and tell me what p_x and p_y are,” you are likely to answer 31 and 26, respectively, because you know that you need to read off the values where the vertical and horizontal lines that radiate out from p hit the axes.

We can express this idea in BSL. Assume you add

```
(define p (make-posn 31 26))
```

to the definitions area and click *RUN*. Then you can perform the following interactions

```
> (posn-x p)
31
> (posn-y p)
26
```

in the interactions area. Defining p is like drawing the point in a Cartesian plane; using `posn-x` and `posn-y` is like subscripting p with an index: p_x and p_y .

Computationally speaking, `posn` structures come with two equations:

```
(posn-x (make-posn x0 y0))      ==      x0
(posn-y (make-posn x0 y0))      ==      y0
```

DrRacket uses these equations during computations. Here is an example of a computation involving `posn` structures:

```
(posn-x p)
== ; DrRacket replaces p with its value (make-posn 31 26)
(posn-x (make-posn 31 26))
== ; DrRacket uses the law for posn-x
31
```

Stop! Confirm the second interaction above with your own computation.

5.3 Programming with `posn`

Now consider designing a function that computes the distance of some location to the origin of the canvas:

The picture clarifies that “distance” means the length of the most direct path from the location to the top-left corner of the canvas—“as the crow flies.”

Here are the purpose statement and the header:

```
; computes the distance of a-posn to the origin
(define (distance-to-0 a-posn)
  0)
```

The key is that `distance-to-0` consumes a single value, some `posn`. It produces a single value, the distance of the location to the origin.

In order to make up examples, we need to know how to compute this distance. For points with `0` as one of the coordinates, the result is the other coordinate:

```
(check-expect (distance-to-0 (make-posn 0 5)) 5)
(check-expect (distance-to-0 (make-posn 7 0)) 7)
```

For the general case, we could try to figure out the formula on our own, or we may recall the formula from your geometry courses. As you know, this is domain knowledge that you might have but in case you don’t, we supply it; after all, this domain knowledge isn’t computer science. So, here is the distance formula for (x,y) again:

Given this formula, we can easily make up some more functional examples:

```
(check-expect (distance-to-0 (make-posn 3 4)) 5)
(check-expect (distance-to-0 (make-posn 8 6)) 10)
(check-expect (distance-to-0 (make-posn 5 12)) 13)
```

Just in case you’re wondering, we rigged the examples so that the results would be easy to figure out. This isn’t the case for all `posn` structures.

Stop! Plug the x- and y-coordinates from the examples into the formula. Confirm the expected results for all five examples.

Next we can turn our attention to the definition of the function. The examples imply that the design of `distance-to-0` does not need to distinguish between different situations; it always just computes the distance from the x- and y-coordinates inside the given `posn` structure. But the function must select these coordinates from the given `posn` structure. And for that, it uses the `posn-x` and `posn-y` primitives. Specifically, the function needs to compute `(posn-x a-posn)` and `(posn-y a-posn)` because `a-posn` is the name of the given, unknown `posn` structure:

```
(define (distance-to-0 a-posn)
  (... (posn-x a-posn) ...
    ... (posn-y a-posn) ...))
```

Using this template and the examples, the rest is easy:

```
(define (distance-to-0 a-posn)
  (sqrt
    (+ (sqr (posn-x a-posn))
      (sqr (posn-y a-posn))))))
```

The function `squares` (`(posn-x a-posn)` and `(posn-y a-posn)`), which represent the x- and y-coordinates, sums up the results, and takes the square root. With DrRacket, we can also quickly check that our new function produces the proper results for our examples.

Exercise 65. Evaluate the following expressions:

- `(distance-to-0 (make-posn 3 4))`
- `(distance-to-0 (make-posn 6 (* 2 4)))`
- `(+ (distance-to-0 (make-posn 12 5)) 10)`

by hand. Show all steps. Assume that `sqr` performs its computation in a single step. Check the results with DrRacket's stepper.

Exercise 66. The Manhattan distance of a point to the origin considers a path that follows the rectangular grid of streets found in Manhattan. Here are two examples:



The left one shows a “direct” strategy, going as far as needed leftwards, followed by as many steps as needed in the upwards direction. In comparison, the right one shows a “random walk” strategy, going some blocks leftwards, some upwards, and so on until the destination—here, the origin—is reached.

Stop! Does it matter which strategy you follow?

Design the function `manhattan-distance`, which measures the Manhattan distance of the given posn to the origin.

5.4 Defining Structure Types

Unlike numbers or Boolean values, structures such as `posn` usually don't come with a programming language. Only the mechanism to define structure types is provided; the rest is left up to the programmer. This is also true for BSL.

The use of brackets in a structure type definition is a convention, not a necessity. It makes the field names stand out. Replacing brackets with parentheses is perfectly acceptable.

A *structure type definition* is another form of definition, distinct from constant and function definitions. Here is how the creator of DrRacket defined the `posn` structure type in BSL:

```
⋮ (define-struct posn [x y])
```

In general, a structure type definition has this shape:

```
⋮ (define-struct StructureName [FieldName ... FieldName])
```

The keyword `define-struct` signals the introduction of a new structure type. It is followed by the name of the structure. The third part of a structure type definition is a sequence of names enclosed in brackets; these names are the *fields*.

A structure type definition actually **defines** functions. But, unlike an ordinary function definition, a

structure type definition defines many functions simultaneously. Specifically, it defines three kinds of functions:

- one *constructor*, a function that creates *structure instances* from as many values as there are fields; as mentioned, *structure* is short for structure instance. The phrase *structure type* is a generic name for the collection of all possible instances.
- a *selector* per field, which extracts the value of the field from a structure instance;
- and one *structure predicate*, which, like ordinary predicates, distinguishes instances from all other kinds of values.

A program can use these as if they were functions or built-in primitives.

One curious aspect of structure type definitions is that it makes up names for the various new operations it creates. Specifically, for the name of the constructor, it prefixes the structure name with “make-” and for the names of the selectors it postfixes the structure name with the field names. Finally, the predicate is just the structure name with “?” added, pronounced “huh” when read aloud.

This naming convention looks complicated but, with a little bit of practice, you get the hang of it. It also immediately explains the functions that come with *posn* structures: `make-posn` is the constructor, `posn-x` and `posn-y` are selectors. While we haven’t encountered `posn?` yet, we now know that it exists; the next chapter explains the role of these predicates in detail.

Enough with *posn* structures for a while. Let’s look at a structure type definition that we might use to keep track of contacts such as those in your cell phone:

```
(define-struct entry [name phone email])
```

Here are the names of the functions that this definition introduces:

- `make-entry`, which consumes three values and constructs an instance of `entry`;
- `entry-name`, `entry-phone`, and `entry-email`, which all consume one instance of `entry` and selects one of the three field values;
- and `entry?`, the predicate.

Since each `entry` combines three values, the expression

```
(make-entry "Sarah Lee" "666-7771" "lee@classy-university.edu")
```

creates an `entry` structure with `"Sarah Lee"` in the `name` field, `"666-7771"` in the `phone` field, and `"lee@classy-university.edu"` in the `email` field.

Once you have an `entry` structure, you can use selectors to find out what the value in a specific field is. Let’s make this concrete again. Enter these definitions

```
(define pl
  (make-entry "Sarah Lee" "666-7771" "lee@classy-university.edu"))

(define bh
  (make-entry "Tara Harper" "666-7770" "harper@small-college.edu"))
```

into the definitions area and click *RUN*. You can then conduct experiments like these:

```
> (entry-name pl)
"Sarah Lee"
> (entry-name bh)
"Tara Harper"
```

When `entry-name` is applied to the above instances of `entry`, it extracts the value in the `name` field.

The other selectors work just fine, too:

```
> (entry-email pl)
```

```
"lee@classy-university.edu"
> (entry-phone pl)
"666-7771"
```

Stop! Try `entry-email` on `bh`.

Exercise 67. Write down the names of the functions (constructors, selectors, and predicates) that the following structure type definitions introduce:

- `(define-struct movie [title producer year])`
- `(define-struct person [name hair eyes phone])`
- `(define-struct pet [name number])`
- `(define-struct CD [artist title price])`
- `(define-struct sweater [material size producer])`

Make sensible guesses as to what kind of values go with which fields. Then create at least one instance per structure type definition.

Every structure type definition introduces a new kind of structure, distinct from all others.

Programmers want this kind of expressive power because they wish to convey an **intention** with the structure name. Wherever a structure is created, selected, or tested, the text of the program explicitly reminds the reader of this intention. If it weren't for these future readers of code, programmers could use one structure definition for structures with one field, another for structures with two fields, a third for structures with three, and so on.

Remember that for computer scientists, the positive direction of the y-axis is down.

In this context consider the problem of representing the bouncing ball mentioned at the very beginning of this chapter. We can pinpoint the ball's location with a single number, namely the distance of pixels from the top. Its *speed* is the number of pixels it moves per clock tick. Its *velocity* is the speed plus the direction in which it moves. Since the ball moves along a straight, vertical line, a number is a perfectly adequate data representation for its velocity:

- A positive number means the ball moves down.
- A negative number means it moves up.

We can use this domain knowledge to formulate a structure type definition:

```
(define-struct ball [location velocity])
```

Both fields are going to contain numbers, so `(make-ball 10 -3)` is a good data example. It represents a ball that is `10` pixels from the top and moves up at `3` pixels per clock tick.

Notice how, in principle, a `ball` structure merely combines two numbers, just like a `posn` structure. When a program contains the expression `(ball-velocity a-ball)`, it immediately conveys that this programs deals with the representation of a ball and its velocity. In contrast, if the program used `posn` structures instead, `(posn-y a-ball)` might mislead a reader that the expression is about a y-coordinate.

Exercise 68. There are other ways to represent bouncing balls. Here is one:

```
(define SPEED 3)
(define-struct balld [location direction])
(make-balld 10 "up")
```

Interpret this program fragment and then create other instances of `balld`.

Since structures are values just like numbers or Booleans or strings, it makes sense that one instance of

a structure occurs inside another instance. Consider game objects. They don't always move along vertical lines. They move in some "oblique" manner across the canvas. Describing both the location and the velocity of a ball moving across a 2-dimensional world canvas demands two numbers: one per direction. For the location part, the two numbers represent the x- and y-coordinates. Velocity describes the *changes* in the horizontal and vertical direction; in other words, these "change numbers" must be added to the respective coordinates to find out where the object is next.

Those of you who know complex numbers may wish to contemplate a data representation that uses complex numbers for both location and velocity. BSL supports those, too.

Let us use `posn` structures to represent locations and let us introduce a `vel` structure type for velocities:

```
(define-struct vel [deltax deltay])
```

In case you are wondering, the word "delta" is commonly used to speak of change when it comes to simulations of physical activities.

Now we can use instances of `ball` to combine a `posn` structure with a `vel` structure to represent balls that move in straight lines but not necessarily along only vertical (or horizontal) lines:

```
(define ball1 (make-ball (make-posn 30 40) (make-vel -10 5)))
```

One way to interpret this instance is to think of a ball that is `30` pixels from the left, `40` pixels from the top. It moves `10` pixels toward the left per clock tick, because subtracting 10 pixels from the x-coordinate brings it closer to the left. As for the vertical direction the ball drops at `5` pixels per clock tick, because adding positive numbers to a y-coordinate increases the distance from the top.

Exercise 69. Here is an alternative to the nested data representation of balls:

```
(define-struct ballf [x y deltax deltay])
```

Programmers often call this a *flat representation*. Create an instance of `ballf` that has the same interpretation as `ball1`.

For a second example of nested structures, let us briefly look at the example of contact lists. Many cell phones support contact lists that allow several phone numbers per name: one for a home line, one for the office, and one for a cell phone number. For phone numbers, we wish to include both the area code and the local number. Since this nests the information, it's best to create a nested data representation, too:

```
(define-struct centry [name home office cell])
(define-struct phone [area number])
(make-centry "Shriram Fisler"
             (make-phone 207 "363-2421")
             (make-phone 101 "776-1099")
             (make-phone 208 "112-9981"))
```

The intention here is that an entry on a contact list has four fields: a name and three phone records. The latter are represented with instances of `phone`, which separates the area code from the local phone number.

In sum, nesting information is natural. The best way to represent such information with data is to mirror the nesting with nested structure instances. Doing so makes it easy to interpret the data in the application domain of the program, and it is also straightforward to go from examples of information to data. Of course, it is really the task of data definitions to specify how to go back and forth between information and data. Before we study data definitions for structure type definitions, however, we first take a systematic look at computing with, and thinking about, structures.

5.5 Computing with Structures

A programming language usually also supports structure-like data that use numeric field names.

Structure types generalizes Cartesian points in two ways. First, a structure type may specify an arbitrary number of fields: zero, one, two, three, and so forth. Second, structure types name fields, they don't number them. Programmers often want to bundle several pieces of information into one, and it is much easier to remember that a family name is available in a field called `last-name` than in the 7th field.

In the same spirit, computing with structure instances generalizes the manipulation of Cartesian points. To appreciate this idea, let us first look at a diagrammatic way to think about structure instances as lock boxes with as many compartments as there are fields. Here is a representation of

```
(make-entry "Sarah Lee" "666-7771" "lee@classy-university.edu")
```

as such a diagram:

The box itself carries a label, identifying it as an instance of a specific structure type definition. Each compartment, too, is labeled. We italicize all labels in such diagrams. The other instance of `entry` from the preceding section,

```
(make-entry "Tara Harper" "666-7770" "harper@small-college.edu")
```

corresponds to a similar box diagram, though the content of the compartments differs:

Not surprisingly, nested structures instances have a diagram of boxes nested in boxes. Thus, `ball1` from above is equivalent to this diagram:

In this case, the outer box contains two boxes, one per field. Generally speaking, structure instances and their boxes can be nested arbitrarily deep. The next part of this book is going to study examples of this kind.

Exercise 70. Draw box representations for the examples you made up for [exercise 67](#).

In the context of this imagery, a selector is like a key. It opens a specific compartment for a certain kind of box and thus enables the holder to extract its content. Hence, applying `entry-name` to `pl` yields a string:

```
> (entry-name pl)
"Sarah Lee"
> (entry-name (make-posn 42 5))
entry-name: expects an entry, given (posn 42 5)
```

But `entry-name` applied to a `posn` structure signals an error. If a compartment contains a box, selectors work just fine, too:

```
> (ball-velocity ball1)
(vel -10 5)
> (vel-deltax (ball-velocity ball1))
-10
```

Applying `ball-velocity` to `ball1` extracts the value of the velocity field, which is an instance of `vel`. We can also apply a selector to the result of a selection, as the second interaction shows. Since the inner expression extracts the velocity from `ball1`, the outer expression extracts the value of the `deltax` field, which in this case is `-10`.

The first interaction also shows that structure instances are values. DrRacket prints them exactly like we enter them, exactly as it acts for other values:

```
> (make-vel -10 5)
(vel -10 5)
> (make-entry "Tara Harper" "666-7770" "harper@small-college.edu")
(entry "Tara Harper" "666-7770" "harper@small-college.edu")
> (make-centry
  "Shriram Fisler"
  (make-phone 207 "363-2421")
  (make-phone 101 "776-1099")
  (make-phone 208 "112-9981"))

(centry
 "Shriram Fisler"
 (phone 207 "363-2421")
 (phone 101 "776-1099")
 (phone 208 "112-9981"))
```

Generally speaking, each structure type definition creates not only new functions and new ways to create values, but it also adds new laws of computation to DrRacket's knowledge. These laws are roughly analogous to those for subscripting `posn` structures in [Computing with posns](#). They are most easily understood by example. When DrRacket encounters a structure type definition with two fields, say

```
(define-struct ball [location velocity])
```

it introduces two laws, one per selector:

```
(ball-location (make-ball l0 v0)) == l0
(ball-velocity (make-ball l0 v0)) == v0
```

For different structure type definitions, it introduces different laws, even if they have the same number of fields. Thus, if a program contains the definition

```
(define-struct vel [deltax deltay])
```

DrRacket uses

```
(vel-deltax (make-vel dx0 dy0)) == dx0
(vel-deltay (make-vel dx0 dy0)) == dy0
```

for future computations.

Using these specific laws, we can now explain the interaction from above:

```
(vel-deltax (ball-velocity ball1))
== ; DrRacket replaces ball1 with its value
(vel-deltax (ball-velocity (make-ball (make-posn 30 40) (make-vel -10 5))))
== ; DrRacket uses the law for ball-velocity
(vel-deltax (make-vel -10 5))
== ; DrRacket uses the law for vel-deltax
-10
```

Stop! What would be the result if you were to use `vel-deltay` instead?

Exercise 71. Spell out the laws that the following structure type definitions introduce:

http://www.ccs.neu.edu/home/matthias/HtDP2e/Draft/part_one.html

```
(define-struct centry [name home office cell])
(define-struct phone [area number])
```

Use these laws to explain how DrRacket arrives at 101 as the value of this expression:

```
(phone-area
  (centry-office
    (make-centry
      "Shriram Fisler"
      (make-phone 207 "363-2421")
      (make-phone 101 "776-1099")
      (make-phone 208 "112-9981"))))
```

Show every step of the computation. Confirm them with DrRacket's stepper.

Predicates are the last idea that we must discuss to completely understand structure type definitions. As mentioned, every structure type definition introduces one new predicate. DrRacket uses these predicates to discover whether a selector is applied to the proper kind of value; the next chapter explains this idea in detail. Here we just want to bring across that these predicates are just like the predicates from “arithmetic.” While `number?` recognizes numbers and `string?` recognizes strings, predicates such as `posn?` and `entry?` recognize `posn` structures and `entry` structures. We can confirm our ideas of how they work with experiments in the interactions area. Assume that the definitions area contains these definitions:

```
(define a-posn (make-posn 7 0))

(define pl
  (make-entry "Sarah Lee" "666-7771" "lee@classy-university.edu"))
```

If `posn?` is a predicate that distinguishes `posn` structures from all other values, then we should expect that it yields `#false` for numbers and `#true` for `a-posn`:

```
> (posn? a-posn)
#true
> (posn? 42)
#false
> (posn? #true)
#false
> (posn? (make-posn 3 4))
#true
```

Naturally, for Boolean values it also produces `#false`, and when applied to other instances of `posn`, it returns `#true`. Similarly, `entry?` distinguishes `entry` structures from all other values:

```
> (entry? pl)
#true
> (entry? 42)
#false
> (entry? #true)
#false
```

In general, a predicate recognizes exactly those values constructed with a constructor of the same name. [Intermezzo: BSL](#) explains this law in detail, and it also collects the laws of computing for BSL in one place.

Exercise 72. Place the following definitions in DrRacket’s definition area:

```
(define HEIGHT 200)
(define MIDDLE (quotient HEIGHT 2))
(define WIDTH 400)
(define CENTER (quotient WIDTH 2))

(define-struct game [left-player right-player ball])
```

```
(define game0
  (make-game MIDDLE MIDDLE (make-posn CENTER CENTER)))
```

Click *RUN* and evaluate the following expressions:

```
(game-ball game0)
(posn? (game-ball game0))
(game-left-player game0)
```

Explain the results with step-by-step computations. Double-check your computations with DrRacket's stepper.

5.6 Programming with Structures

Proper programming calls for data definitions. With the introduction of structure type definitions, data definitions become interesting. Remember that a data definition provides a way of representing information into data and interpreting data as information. For structure types, this calls for a description of what kind of data goes into which field. For some structure type definitions formulating such descriptions is easy and obvious:

```
(define-struct posn [x y])
; A Posn is a structure: (make-posn Number Number)
; interpretation the number of pixels from left and from top
```

It doesn't make any sense to use other kinds of data to create a *posn*. Similarly, all instances of *entry*—our structure type definition for entries on a contact list—are clearly supposed to be strings according to our usage in the preceding section:

```
(define-struct entry [name phone email])
; An Entry is a structure: (make-entry String String String)
; interpretation a contact's name, 7-digit phone#, and email address
```

For both of them, it is also straightforward to describe how a reader is to interpret instances of these structures in the application domain.

Contrast this simplicity with the structure type definition for *ball*, which obviously allows at least two distinct interpretations:

```
(define-struct ball [location velocity])
; A Ball-1d is a structure: (make-ball Number Number)
; interpretation 1 the position from top and the velocity
; interpretation 2 the position from left and the velocity
```

Whichever one we use in a program, we must stick to it consistently. As [Defining Structure Types](#) shows, however, it is also possible to use *ball* structures in an entirely different manner:

```
; A Ball-2d is a structure: (make-ball Posn Vel)
; interpretation 2-dimensional position with a 2-dimensional velocity

(define-struct vel [deltax deltay])
; A Vel is a structure: (make-vel Number Number)
; interpretation (make-vel dx dy) means a velocity of dx pixels [per tick]
; along the horizontal and dy pixels along the vertical direction
```

Here we name a second collection of data, *Ball-2d*, distinct from *Ball-1d*, to describe data representations for balls that move in straight lines across a world canvas. In short, it is possible to use **one and the same** structure type in **two different ways**. Of course, within one program, it is best to stick

To do so, we use distinct names for the two data collections. As long as we use the names consistently, we can call data collections whatever we want.

to one and only one use; otherwise you are setting yourself up for problems.

Also [Ball-2d](#) refers to another one of our data definitions, namely, the one for [Vel](#). While all other data definitions have thus far referred to built-in data collections ([Number](#), [Boolean](#), [String](#)), it is perfectly acceptable, and indeed common, that one of your data definition refers to another.

Exercise 73. Formulate a data definition for the above phone structure type definition that accommodates the given examples.

Next formulate a data definition for phone numbers using this structure type definition:

```
| (define-struct phone# [area switch num])
```

Historically, the first three digits make up the area code, the next three the code for the phone switch (exchange) of your neighborhood, and the last four represent the phone with respect to the neighborhood. Describe the content of the three fields as precisely as possible with intervals.

At this point, you might be wondering what data definitions really mean. This question, and its answer, is the topic of the next section. For now, we indicate how to use data definitions for program design.

Here is a problem statement to set up some context:

Sample Problem: Your team is designing an interactive game program that moves a red dot across a 100 x 100 canvas and allows players to use the mouse to place the dot. Together you chose [Posn](#) as the data representation for the game state. Here is how far you got:

```
; visual constants
(define MTS (empty-scene 100 100))
(define DOT (circle 3 "solid" "red"))

; The state of the world is represented by a Posn.

; Posn -> Posn
(define (main p0)
  (big-bang p0
    [on-tick x+]
    [on-mouse reset-dot]
    [to-draw scene+dot]))
```

Your task is to design `scene+dot`, the function that consumes the state of the world and adds a red dot to the empty canvas at the specified position.

The problem context dictates the signature of your function:

```
; Posn -> Image
; adds a red spot to MTS at p
(define (scene+dot p)
  MTS)
```

Adding a purpose statement is straightforward. As suggested in [Designing Functions](#), it uses the function's parameter to express what the function computes.

Next we work out a couple of examples:

```
(check-expect (scene+dot (make-posn 10 20))
              (place-image DOT 10 20 MTS))
(check-expect (scene+dot (make-posn 88 73))
              (place-image DOT 88 73 MTS))
```

Given that the function consumes a [Posn](#), we know that the function can extract the values of the `x` and `y` fields:

```
(define (scene+dot p)
  (... (posn-x p) ... (posn-y p) ...))
```

Once we see these additional pieces in the body of the function, the rest of the definition is straightforward. Using `place-image`, the function puts DOT into MTS at the coordinates contained in p:

```
(define (scene+dot p)
  (place-image DOT (posn-x p) (posn-y p) MTS))
```

A function may produce structures, not just consume them. Let's resume our sample problem from above because it includes just such a task:

Sample Problem: Another colleague is asked to define `x+`, a function that consumes a `Posn` and increases the x-coordinate by 3.

Recall that the `x+` function handles clock ticks.

We can adapt the first few steps of the design of `scene+dot`:

```
; Posn -> Posn
; increases the x-coordinate of p by 3
(check-expect (x+ (make-posn 10 0)) (make-posn 13 0))
(define (x+ p)
  (... (posn-x p) ... (posn-y p) ...))
```

The signature, the purpose, and the example all come out of the problem statement. Instead of a header—a function with a default result—our sketch contains the two selector expressions for `Posns`. After all, the information for the result must come from the inputs, and the input is a structure that contains two values.

Finishing the definition is a small step now. Since the desired result is a `Posn`, which are created with `make-posn`, the function can use it to combine the pieces in the skeleton:

```
; Posn -> Posn
; increases the x-coordinate of p by 3
(check-expect (x+ (make-posn 10 0)) (make-posn 13 0))
(define (x+ p)
  (make-posn (+ (posn-x p) 3) (posn-y p)))
```

Exercise 74. Design the function `posn-up-x`, which consumes a `Posn` p and a Number n. It produces a `Posn` like p with n in the x field.

A neat observation is that we can define `x+` using `posn-up-x`:

```
(define (x+ p)
  (posn-up-x p (+ (posn-x p) 3)))
```

Note Functions such as `posn-up-x` are called *updaters functional setters*. They are extremely useful when you write large programs.

A function may also produce structure instances from atomic data. While `make-posn` is built-in primitive that does so, our running problem provides another fitting illustration:

Sample Problem: A third colleague is tasked to design the `reset-dot` function, which resets the dot when a mouse click occurs.

To tackle this problem, you need to recall from [Designing World Programs](#) that mouse-event handlers consume four values: the current state of the world, the x- and y-coordinate of the mouse click, and a `MouseEvt`.

By combining the knowledge from the sample problem with the general world-design recipe, we get a preliminary definition, including a signature and a purpose statement:

```
; Posn Number Number MouseEvt -> Posn
; for mouse clicks, (make-posn x y); otherwise p
(define (reset-dot p x y me)
  p)
```

In BSL, it is possible to make up examples for functions that deal with mouse events. We need a `Posn`, two numbers, and a `MouseEvt`, which is just a special kind of `String`. A mouse click, for example, is represented with two strings: `"button-down"` and `"button-up"`. The first one signals that a user pressed the mouse button, the latter signals the release of the button. With this in mind, here are two examples:

```
(check-expect (reset-dot (make-posn 10 20) 29 31 "button-down")
               (make-posn 29 31))
(check-expect (reset-dot (make-posn 10 20) 29 31 "button-up")
               (make-posn 10 20))
```

Stop! Interpret the two examples.

Although the function consumes only atomic forms of data, its purpose statement and the examples suggest that it differentiates between two kinds of `MouseEvt`s: `"button-down"` and all others. Such a case split suggests a `cond` expression:

```
(define (reset-dot p x y me)
  (cond
    [(mouse=? "button-down" me) (... p ... x y ...)]
    [else (... p ... x y ...)]))
```

Following the design recipe, this skeleton mentions the parameters to remind you of what data is available.

The rest is straightforward again because the purpose statement itself dictates what the function computes in each of the two cases:

```
(define (reset-dot p x y me)
  (cond
    [(mouse=? me "button-down") (make-posn x y)]
    [else p]))
```

As above, we could have mentioned that `make-posn` is useful to create instances of `Posn` but you know this, and we don't need to remind you constantly.

Exercise 75. Copy all relevant constant and function definitions to DrRacket's definition area. Add the tests and make sure they pass. Then run the program and use the mouse to place the red dot.

Physics tells you that, when you are given an object's location and velocity, you can determine the object's location after one unit of time by "adding" the velocity to the location.

Many programs deal with nested structures. We illustrate this point with another small excerpt from a world program:

Sample Problem: Your team is designing a game program that keeps track of an object that moves across the canvas at changing speed. The chosen data representation requires two data definitions:

```
(define-struct ufo [loc vel])
; A UFO is a structure: (make-ufo Posn Vel)
; interpretation (make-ufo p v) is at location p moving at velocity v
; For Vel, see above.
```

It is your task to develop the `ufo-move-1` function, which computes the location of a given `UFO` after one clock tick passes.

The order of these definitions matters. See [Intermezzo: BSL](#).

Let us start with some examples that explore the data definitions a bit:

```
(define v1 (make-vel 8 -3))
(define v2 (make-vel -5 -3))

(define p1 (make-posn 22 80))
(define p2 (make-posn 30 77))

(define u1 (make-ufo p1 v1))
(define u2 (make-ufo p1 v2))
(define u3 (make-ufo p2 v1))
(define u4 (make-ufo p2 v2))
```

The first four are elements of `Vel` and `Posn`. The last four combine the first four in all possible combinations.

Next we write down a signature, a purpose, some examples and a function header:

```
; UFO -> UFO
; determines where u moves in one clock tick;
; leaves the velocity as is

(check-expect (ufo-move-1 u1) u3)
(check-expect (ufo-move-1 u2) (make-ufo (make-posn 17 77) v2))

(define (ufo-move-1 u) u)
```

For the function examples, we use the data examples **and** our domain knowledge of positions and velocities. Specifically, we know that a vehicle that is moving north at 60 miles per hour and west at 10 miles per hour is going to end up 60 miles north from its starting point and 10 miles west, after one hour of driving. After two hours, it will be 120 miles north from the starting point and 20 miles to its west.

As before, we decide that a function that consumes a structure instance can (and probably must) extract information from the structure to compute its result. So once again we add selector expressions to the function definition:

```
(define (ufo-move-1 u)
  (... (ufo-loc u) ... (ufo-vel u) ...))
```

Note The addition of the selector expressions raises the question whether we need to refine this sketch even more. After all, the two expressions extract instances of `Posn` and `Vel`, respectively. These two are also structure instances, and we could extract values from them in turn. Here is what the resulting skeleton would look like:

```
; UFO -> UFO
(define (ufo-move-1 u)
  (... (posn-x (ufo-loc u)) ... (posn-y (ufo-loc u)) ...
    ... (vel-deltax (ufo-vel u)) ... (vel-deltay (ufo-vel u)) ...))
```

Doing so obviously makes the sketch look quite complex, however. For truly realistic programs, following this idea to its logical end would create incredibly complex program outlines. More generally,

If a function deals with nested structures, develop one function per level.

In the second part of the book, this guideline becomes even more important and we refine it a bit. **End**

Here we focus on how to combine the given `Posn` and the given `Vel` in order to obtain the next location of the `UFO`. After all, that's what our domain knowledge says. Specifically, it says that we should "add" the two together, where "adding" can't mean the operation we usually apply to numbers. So let us imagine that we have a function for adding a `Vel` to a `Posn`:

```
; Posn Vel -> Posn
; adds v to p
(define (posn+ p v) p)
```

Writing down the signature, purpose, and header like this is a legitimate way of programming. It is called “making a wish” and is a part of “making a wish list” as described in [From Functions to Programs](#).

The key is to make wishes in such a way that we can complete the function that we are working on. In this manner, we can split difficult programming tasks into different tasks, a technique that helps us solve problems in reasonably small steps. For the sample problem, we get a complete definition for `ufo-move-1`:

```
(define (ufo-move-1 u)
  (make-ufo (posn+ (ufo-loc u) (ufo-vel u)) (ufo-vel u)))
```

Because `ufo-move-1` and `posn+` are complete definitions, we can even click *RUN*, which checks that DrRacket doesn’t complain about grammatical problems with our work so far. Naturally, the tests fail because `posn+` is just a wish, not the function we need.

In geometry courses, the operation corresponding to `posn+` is called a *translation*.

Now it is time to focus on `posn+`. We have completed the first two steps of the design (data definitions, signature/purpose/header), so we must create examples. One easy way to create functional examples for a “wish” is to use the examples for the original function and to turn them into examples for the new function:

```
(check-expect (posn+ p1 v1) p2)
(check-expect (posn+ p1 v2) (make-posn 17 77))
```

For this problem, we know that `(ufo-move-1 (make-ufo p1 v1))` is to produce `p2`. At the same time, we know that `ufo-move-1` applies `posn+` to `p1` and `v1`, implying that `posn+` must produce `p2` for these inputs. Stop! Check our manual calculations to insure you are following what we are doing.

We are now able to add selector expressions to our design sketch:

```
(define (posn+ p v)
  (... (posn-x p) ... (posn-y p) ...)
  ... (vel-deltax v) ... (vel-deltay v) ...))
```

Because `posn+` consumes instances of `Posn` and `Vel` and because each piece of data is an instance of a two-field structure, we get four expressions. In contrast to the nested selector expressions from above, these are simple applications of a selector to a parameter.

If we remind ourselves what these four expressions represent, or if we recall how we computed the desired results from the two structures, our completion of the definition of `posn+` is straightforward:

```
(define (posn+ p v)
  (make-posn (+ (posn-x p) (vel-deltax v))
             (+ (posn-y p) (vel-deltay v)))))
```

The first step is to add the velocity in the horizontal direction to the x-coordinate and the velocity in the vertical direction to the y-coordinate. Doing so yields two expressions that compute the two new coordinates. With `make-posn` we can combine the two coordinates into a single `Posn` again, as desired.

Exercise 76. Enter these definitions and their test cases into the definitions area of DrRacket and make sure they work. It is the first time that we made a “wish” and you need to make sure you understand how the two functions work together.

5.7 The Universe of Data

Every language comes with a universe of data. This data represents information from and about the external world; it is what

In mathematics such collections are called sets.

programs manipulate. This universe of data is a collection that consists of collections of built-in data but also of classes of data that programs create.

Figure 22: The universe of data

[Figure 22](#) shows one way to imagine the universe of BSL. Since there are an infinitely many numbers and strings, the collection of all data is infinite. We indicate “infinity” in the figure with “...” but a real definition would have to avoid this imprecision.

Neither programs nor individual functions in programs deal with the entire universe of data. It is the purpose of a data definition to describe parts of this universe and to name these parts so that we can refer to them concisely. Put differently, a named data definition is a description of a collection of data, and that name is usable in other data definitions and in function signatures. In a function signature, the name specifies what data a function will deal with and, implicitly, which part of the universe of data it won’t deal with.

Figure 23: The meaning of a data definition

Practically the data definitions of the first four chapters restrict built-in collections of data. They do so via an explicit or implicit itemization of all included values. For example, the region shaded with gray stripes in [figure 23](#) depicts the following data definition:

```
; A BS is one of:  
; - "hello",  
; - "world", or  
; - pi.
```

While a data definition picking out two strings and one number is silly, note the stylized mix of English and BSL that is used. Its meaning is precise and unambiguous, clarifying exactly which elements belong to `BS` and which don’t.

The introduction of structure types creates an entirely new picture. When a programmer defines a structure type, the universe expands with all possible structure instances. For example, the addition of a `posn` structure type means that instances of `posn` with all possible values in the two fields appear. The middle bubble in [figure 24](#) depicts the addition of those values, showing things such as `(make-posn "hello" 0)` and even `(make-posn (make-posn 0 1) 2)`. And yes, even though some of these instances of `posn` make no sense to us, it is indeed possible to construct all of them in a BSL program.

Figure 24: Adding structure to a universe

The addition of another structure type definition mixes and matches everything, too. Say we add a structure type definition for `ball`, also with two fields. As the third bubble in [figure 24](#) shows, this addition creates instances of `ball` that contain numbers, `posn` structures, and so on as well as instances of `posn` that contain instances of `ball`. Try it out in DrRacket! Add

```
| (define-struct ball [location velocity])
```

to the definitions area, hit *RUN*, and create some structure instances like this:

```
> (make-posn (make-ball "hello" 1) #false)
(posn (ball "hello" 1) #false)
> (make-posn (make-ball (make-ball (make-posn 1 2) 3) 4) 5)
(posn (ball (ball (posn 1 2) 3) 4) 5)
```

Of course, there are no limits on how far you can nest these structure, as the second expression illustrates. Play with structures and learn.

Figure 25: Imposing a data definition on structures

As far as the pragmatics of data definitions is concerned, a data definition for structure types describes large collections of data via combinations of existing data definitions with instances. When we write

In mathematics and physics courses, you encounter one such kind of data: Cartesian coordinates, which also combine two numbers into one item.

```
| ; Posn is (make-posn Number Number)
```

we are describing an infinite number of possible instances of `posn`. Like above, the data definitions use combinations of natural language, data collections defined elsewhere and data constructors. Nothing else should show up in a data definition at the moment.

A data definition for structures specifies a new collection of data made up of those instances to be used by our functions. For example, the data definition for `Posns` identifies the region shaded with gray stripes in [figure 25](#), which includes all those `posn` structures whose two fields contain numbers. At the same time, it is perfectly possible to construct an instance of `posn` that doesn't satisfy the requirement that both fields contain numbers:

```
| (make-posn (make-posn 1 1) "hello")
```

this structure contains a `posn` in the `x` field and a string in the `y` field.

Exercise 77. Formulate data definitions for the following structure type definitions:

- (`(define-struct movie [title producer year])`)
- (`(define-struct person [name hair eyes phone])`)
- (`(define-struct pet [name number])`)
- (`(define-struct CD [artist title price])`)
- (`(define-struct sweater [material size producer])`)

Make sensible assumptions as to what kind of values go into each field.

Exercise 78. Provide a structure type definition and a data definition for representing points in time since midnight. A point in time consists of three numbers: hours, minutes, and seconds.

Exercise 79. Provide a structure type definition and a data definition for representing lower-case three-letter words. A word consists of letters, represented with the one-letter strings "a" through "z" plus `#false`. **Note** This exercise is a small part of the design of a Hangman game; see [exercise 399](#).

Programmers not only write data definitions, they also read them in order to understand programs, to expand the kind of data they can deal with, to eliminate errors, and so on. We read a data definition to understand how to create data that belongs to the designated collection and to determine whether some piece of data belongs to some specified class.

Since data definitions play such a central and important role in the design process, it is often best to illustrate data definitions with examples just like we illustrate the behavior of functions with examples. And indeed, creating data examples from a data definition is straightforward:

- for a built-in collection of data (number, string, Boolean, images), choose your favorite examples;

Note On occasion, people use descriptive names to qualify built-in data collections, such as `NegativeNumber` or `OneLetterString`. They are no replacement for a well-written data definition.

- for an enumeration, use several of the items of the enumeration;
- for intervals, use the end points (if they are included) and at least one interior point;
- for itemizations, deal with each part separately; and
- for data definitions for structures, follow the natural language description, that is, use the constructor and pick an example from the data collection named for each field.

That's all there is to constructing examples from data definitions for most of this book, though data definitions are going to become much more complex than what you have seen so far.

Exercise 80. Create examples for the following data definitions:

- ; A Color is one of:
 ; – "white"
 ; – "yellow"
 ; – "orange"
 ; – "green"
 ; – "red"
 ; – "blue"
 ; – "black"

Note DrRacket recognizes many more strings as colors.

- ; H (a “happiness scale value”) is a number in [0,100],
 ; i.e., a number between 0 and 100
- (`(define-struct person [fstname lstname male?])`)
 ; Person is (make-person String String Boolean)

Is it a good idea to use a field name that looks like the name of a predicate?

- ```
(define-struct dog [owner name age happiness])
; Dog is (make-dog Person String PositiveInteger H)
```

Add an interpretation to this data definition, too.

- ```
; Weapon is one of:
; - #false
; - Posn
; interpretation #false means the missile hasn't been fired yet;
; an instance of Posn means the missile is in flight
```

The last definition is an unusual itemization, using both built-in data and a structure type definition. The next chapter deals with this kind of data definition in depth.

5.8 Designing with Structures

The introduction of structure types reinforces that the process of creating functions has (at least) six steps, something already discussed in [Designing with Itemizations](#). It no longer suffices to rely on built-in data collections to represent information; it is now clear that programmers must create data definitions for all but the simplest problems.

Sample Problem: Design a function that computes the distance of objects in a 3-dimensional space to the origin of the coordinate system.

1. When a problem calls for the representation of pieces of information that belong together or describe a natural whole, you need a structure type definition. It requires as many fields as there are relevant properties. An instance of this structure type corresponds to the whole, and the values in the fields to its attributes.

A data definition for a structure type introduces a name for the collection of instances that are legitimate. Furthermore it must describe which kind of data goes with which field. **Use only names of built-in data collections or previously defined data definitions.**

In the end, we (and others) must be able to use the data definition to create sample structure instances. Otherwise, something is wrong with our data definition. To ensure that we can create instances, our data definitions should come with **data examples**.

Here is how we apply this idea to the sample problem:

```
(define-struct r3 [x y z])
; R3 is (make-r3 Number Number Number)

(define ex1 (make-r3 1 2 13))
(define ex2 (make-r3 -1 0 3))
```

The structure type definition introduces a new kind of structure, `r3`, and the data definition introduces `R3` as the name for all instances of `r3` that contain only numbers.

2. You still need a signature, a purpose statement, and a function header but there is nothing new here. We leave it to you to apply this idea to the sample problem.
3. Use the examples from the first step to create functional examples. For each field associated with intervals or enumerations, make sure to pick end points and intermediate points to create functional examples. We expect you to continue working on the sample problem.
4. A function that consumes structures usually—though not always—extracts the values from the various fields in the structure. To remind yourself of this possibility, write templates for such functions containing a selector for each field. Furthermore, you may want to write down next to each selector expression what kind of data it extracts from the given structure; this information is found in the data definition.

Do not create selector expressions if a field value is itself a structure. It is better to wish for an auxiliary function that processes the extracted field values.

Here is what we have so far for the sample problem, including the template:

```
; R3 -> Number
; determines the distance of p to the origin
(define (r3-distance-to-0 p)
  (... (r3-x p) ... (r3-y p) ... (r3-z p) ...))
```

5. Use the selector expressions from the template when you finally define the function, keeping in mind that you may not need (some of) them.
5. Test. Test as soon as the function header is written. Test until all expressions have been covered. And test again when you make changes.

Finish the sample problem. If you cannot remember the distance of a 3-dimensional point to the origin, look it up in a geometry book.

Exercise 81. Create templates for functions that consume instances of the following structure types:

There you will find a formula such as

- (`define-struct movie [title director year]`)
- (`define-struct person [name hair eyes phone]`)
- (`define-struct pet [name number]`)
- (`define-struct CD [artist title price]`)
- (`define-struct sweater [material size color]`)

No, you do not need data definitions for this task.

Exercise 82. Design the function `time->seconds`, which consumes instances of `time structures` and produces the number of seconds that have passed since midnight. For example, if you are representing 12 hours, 30 minutes, and 2 seconds with one of these structures and if you then apply `time->seconds` to this instance, the correct result is **45002**.

Exercise 83. Design the function `compare-word`. The function consumes two (`representations of` three-letter words. It produces a word that indicates where the given ones agree and disagree. The function retains the content of the structure fields if the two agree; otherwise it places `#false` in the field of the resulting word. **Hint** The exercises mentions two tasks: the comparison of words and the comparison of “letters.”

5.9 Structure in the World

When a world program must track two different and independent pieces of information, we must use a collection of structures to represent the world state data. One field keeps track of one piece of information and the other field the second piece of information. Naturally, if the domain world contains more than two independent pieces of information, the structure type definition must specify as many fields as there are distinct pieces of information.

Consider a space invader game, in which a UFO descends along a straight vertical line and some tank moves horizontally at the bottom of a scene. If both objects move at known constant speeds, all that's needed to describe these two objects is one piece of information per object: the y-coordinate for the UFO and the x-coordinate for the tank. Putting those together requires a structure with two fields:

```
(define-struct space-game [ufo tank])
```

We leave it to you to formulate an adequate data definition for this structure type definition including an interpretation. Ponder the hyphen in the name of the structure. BSL really allows the use of all kinds

of characters in the names of variables, functions, structures, and field names. What are the selector names for this structure, the name of the predicate?

Every time we say piece of information, we don't necessarily mean a single number or a single word. A piece of information may itself combine several components. Thus, creating a data representation for world states that consist of two (or more) complex pieces of information leads to nested structures.

Let us return to our imaginary space invader game for a moment. Clearly, a UFO that descends only along a vertical line is boring. To turn this idea into an interesting game where the tank attacks the UFO with some weapon, the UFO must be able to descend in non-trivial lines, perhaps jumping randomly. An implementation of this idea means that we need two coordinates to describe the location of the UFO, so that our revised data definition for the space game becomes:

```
; SpaceGame is (make-space-game Posn Number).
; interpretation (make-space-game (make-posn ux uy) tx) means that the
; UFO is currently at (ux,uy) and the tank's x-coordinate is tx
```

Understanding what kind of data representations is needed for world programs takes practice. To this end, the following two sections introduce several reasonably complex problem statements. We recommend you solve them before moving on to the kind of games that you might like to design on your own.

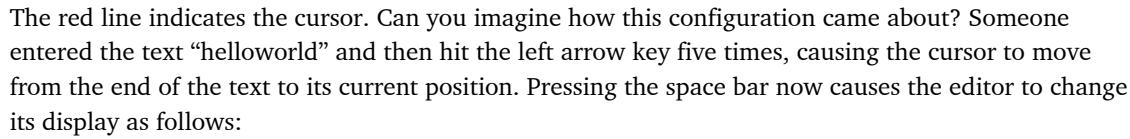
5.10 A Graphical Editor

One step of the programming process is to create a text document. To program in BSL, you open DrRacket, type on the keyboard, and watch text appear. Pressing the left arrow on the keyboard moves the cursor to the left; pressing the backspace (or delete) key erases a single letter to the left of the cursor—if there is a letter to start with.

This process is called “editing” though its precise name should be “text editing of programs” because we will use “editing” for a more demanding task than typing on a keyboard. When you write and revise other kinds of documents, say, an English assignment, you are likely to use other software applications, called word processors, though computer scientists dub all of them editor or even graphical editor.

You are now in a position to design a world program that acts as a one-line editor for plain text. Editing here includes entering letters and somehow changing the already existing text, including the deletion and the insertion of letters. This implies some notion of position within the text. People call this position a *cursor*; most graphical editors display it in such a way that it can easily be spotted.

Take a look at the following editor configuration:



The red line indicates the cursor. Can you imagine how this configuration came about? Someone entered the text “helloworld” and then hit the left arrow key five times, causing the cursor to move from the end of the text to its current position. Pressing the space bar now causes the editor to change its display as follows:

In short, it shows the phrase “hello world” with a cursor between the newly inserted space and the “w” from the original text.

Given this background, an editor world program must keep track of two pieces of information:

1. the text entered so far
2. the current location of the cursor.

And this suggests a structure type with two fields.

We can imagine several different ways of going from the information to data and back. For example, one field in the structure may contain the entire text entered and the other the number of characters

between the first character (counting from the left) and the cursor. Another data representation is to use two strings in the two fields: the part of the text to the left of the cursor and the part of the text to its right. Here is our preferred approach to representing the state of an editor:

```
(define-struct editor [pre post])
; Editor = (make-editor String String)
; interpretation (make-editor s t) means the text in the editor is
; (string-append s t) with the cursor displayed between s and t
```

Solve the next few exercises based on this data representation.

Exercise 84. Design the function `render`, which consumes an `Editor` and produces an image.

The purpose of the function is to render the text within an empty scene of 200 x 20 pixels. For the cursor, use a 1 x 20 red rectangle and for the strings, black text of size 16.

Develop the image for a sample string in DrRacket's interaction area. We started with this expression:

```
(overlay/align "left" "center"
  (text "hello world" 11 "black")
  (empty-scene 200 20))
```

You may wish to read up on `beside`, `above`, and such functions. When you are happy with the looks of the image, use the expression as a test and as a guide to the design of `render`.

Exercise 85. Design `edit`. The function consumes two inputs, an editor `ed` and a `KeyEvent` `ke`, and it produces another editor. Its task is to add a single-character `KeyEvent` `ke` to the end of the `pre` field of `ed`, unless `ke` denotes the backspace ("\"b") key. In that case, it deletes the character immediately to the left of the cursor (if there are any). The function ignores the tab key ("\"t") and the return key ("\"r").

The function pays attention to only two `KeyEvents` longer than one letter: "left" and "right". The left arrow moves the cursor one character to the left (if any), and the right arrow moves it one character to the right (if any). All other such `KeyEvents` are ignored.

Develop a good number of examples for `edit`, paying attention to special cases. When we solved this exercise, we created 20 examples and turned all of them into tests.

Hint Think of this function as consuming `KeyEvents`, a collection that is specified as an enumeration. It uses auxiliary functions to deal with the `Editor` structure. Keep a wish list handy; you will need to design additional functions for most of these auxiliary functions, such as `string-first`, `string-rest`, `string-last`, and `string-remove-last`. If you haven't done so, solve the exercises in [Functions](#).

Exercise 86. Define the function `run`. It consumes a string—the `pre` field of an editor—and then launches an interactive editor, using `render` and `edit` from the preceding two exercises for the `to-draw` and `on-key` clauses.

Exercise 87. Notice that if you type a lot, your editor program does not display all of the text. Instead the text is cut off at the right margin. Modify your function `edit` from [exercise 85](#) so that it ignores a keystroke if adding it to the end of the `pre` field would mean the rendered text is too wide for your canvas.

Exercise 88. Develop a data representation based on our first idea, using a string and an index. Then solve exercises [exercise 84](#) through [exercise 87](#) again.

Follow the design recipe.

Note on Design Choices The exercise is a first study of making design choices. It shows that the very first design choice concerns the data representation. Making the right choice requires planning ahead and weighing the complexity of each. Of course, getting good at this is a question of gaining experience.

And again, if you haven't done so, solve the exercises in [Functions](#).

5.11 More Virtual Pets

In this section we continue our virtual zoo project from [Virtual Pet Worlds](#). Specifically, the goal of the exercise is to combine the cat world program with the program for managing its happiness gauge. When the combined program runs, you see the cat walking across the canvas and, with each step, its happiness goes down. The only way to make the cat happy is to feed it (down arrow) or to pet it (up arrow). Finally, the goal of the last exercise is create another virtual, happy pet.

Exercise 89. Define a structure type that keeps track of the cat's x-coordinate and its happiness. Then formulate a data definition for cats, dubbed *VCat*, including an interpretation with respect to a combined world.

Exercise 90. Design the happy-cat world program, which presents an walking cat and that manages and displays its happiness level. It consumes the starting x-coordinate for the cat's walk; let's assume that the cat starts out with perfect happiness.

Hints (1) Reuse the functions from the world programs in [Virtual Pet Worlds](#). (2) Use structure type from the preceding exercise to represent the state of the world.

Exercise 91. Modify the happy-cat program from the preceding exercises so that it stops when the cat's happiness ever falls to 0.

Exercise 92. Extend your structure type definition and data definition from [exercise 89](#) to include a direction field. Adjust your happy-cat program so that the cat moves in the specified direction. The program should move the cat in the current direction, and it should turn the cat around when it reaches either end of the scene.

```
(define cham )
```

This drawing of a chameleon is a **transparent** image. To insert it into DrRacket, insert it with the “Insert Image” menu item. Using this instruction preserves the transparency of the drawing’s pixels.

When a partly transparent image is combined with a colored shape, say a rectangle, the image takes on the underlying color. In the chameleon drawing, it is actually the inside of the animal that is transparent; the area outside is solid white. Try out this expression in your DrRacket, using the *2htdp/image* library:

```
(overlay
  cham
  (rectangle (image-width cham) (image-height cham) "solid" "red"))
```

Exercise 93. Design the *cham* program, which has the chameleon continuously walking across the canvas, from left to right. When it reaches the right end of the canvas, it disappears and immediately reappears on the left. Like the cat, the chameleon gets hungry from all the walking and, as time passes by, this hunger expresses itself as unhappiness.

For managing the chameleon's happiness gauge, you may reuse the happiness gauge from the virtual cat. To make the chameleon happy, you feed it (down arrow, two points only); petting isn't allowed. Of course, like all chameleons, ours can change color, too: "r" turns it red, "b" blue, and "g" green. Add the chameleon world program to the virtual cat game and reuse functions from the latter when possible.

Start with a data definition, *VCham*, for representing chameleons.

Exercise 94. Copy your solution to [exercise 93](#) and modify the copy so that the chameleon walks across a tricolor background. Our solution uses these colors:

Have some Italian pizza when you've solved the problem.

```
(define BACKGROUND
  (beside (empty-scene WIDTH HEIGHT "green")
          (empty-scene WIDTH HEIGHT "white")
          (empty-scene WIDTH HEIGHT "red")))
```

but you may use any colors you wish. Observe how the chameleon changes colors to blend in as it crosses the border between two colors.

Note When you watch the animation carefully, you will see the chameleon riding on a white rectangle. If you know how to use image editing software, modify the picture so that the white rectangle is invisible. Then the chameleon will really blend in.

6 Itemizations and Structures

In the preceding two chapters, we have encountered two new kinds of data definitions. Those that employ itemization (enumeration and intervals) are used to create small collections from large ones. Those that use structures combine multiple collections. Since this book keeps emphasizing that the development of data representations is the starting point for proper program design, it cannot surprise you that programmers frequently want to itemize data definitions that involve structures or use structures to combine itemized data.

Recall the imaginary space invader game from [Structure in the World](#) in the preceding chapter. Thus far, it involves one UFO, descending from space, and one tank on the ground, moving horizontally. Our data representation uses a structure with two fields: one for the data representation of the UFO and another one for the data representation of the tank. Naturally players will want a tank that can fire off a missile. All of a sudden, we can think of a second kind of state that contains three independently moving objects: the UFO, the tank, and the missile. Thus we have two distinct structures: one for representing two independently moving objects and another one for the third. Since a world state may now be one of these two structures, it is natural to use an itemization to describe all possible states:

1. the state of the world is a structure with **two** fields, or
2. the state of the world is a structure with **three** fields.

As far as our domain is concerned—the actual game—the first kind of state represents the time before the tank has launched its sole missile and the second one after the missile has been fired.

No need to worry, the next part is about firing as many missiles as you want, without reloading.

This chapter introduces the basic idea of itemizing data definitions that involve structures. Because we have all the other ingredients we need, we start straight with itemizing structures. After that, we discuss some examples, including world programs that benefit from our new power. The last section is about errors in programming.

6.1 Designing with Itemizations, Again

Let us start with a refined problem statement for our space invader game from [Programming with Structures](#).

Sample Problem: Design a game program using the *2htdp/universe* library for playing a simple space invader game. The player is in control of a tank (a small rectangle) that must defend our planet (the bottom of the canvas) from a UFO (see [Intervals](#) for one possibility) that

descends from the top of the canvas to the bottom. In order to stop the UFO from landing, the player may fire a single missile (a triangle smaller than the tank) by hitting the space bar. In response, the missile emerges from the tank. If the UFO collides with the missile, the player wins; otherwise the UFO lands and the player loses.

Here are some details concerning the three game objects and their movements. First, the tank moves a constant speed along the bottom of the canvas though the player may use the left arrow key and the right arrow key to change directions. Second, the UFO descends at a constant velocity but makes small random jumps to the left or right. Third, once fired the missile ascends along a straight vertical line at a constant speed at least twice as fast as the UFO descends. Finally, the UFO and the missile collide if their reference points are close enough, for whatever you think “close enough” means.

The following two subsections use this sample problem as a running example, so study it well and solve the following exercise before you continue. Doing so will help you understand the problem in enough depth.

Exercise 95. Draw some sketches of what the game scenery looks like at various stages. Use the sketches to determine the constant and the variable pieces of the game. For the former, develop physical constants that describe the dimensions of the world (canvas), its objects, and the graphical constants used to render these objects. Then develop graphical constants for the tank, the UFO, the missile, and some background scenery. Finally, create your initial scene from the constants for the tank, the UFO, and the background.

Defining Itemizations The first step in our design recipe calls for the development of data definitions. One purpose of a data definition is to describe the construction of data that represent the state of the world; another is to describe all possible pieces of data that the functions of the world program may consume. Since we haven’t seen itemizations that include structures, this first subsection introduces the basic idea via example. While this is straightforward and probably won’t surprise you, pay close attention.

As argued in the introduction to this chapter, the space invader game with a missile-firing tank requires a data representation for two different kinds of game states. We choose two structure type definitions:

For this space invader game, we could get away with one structure type definition of three fields where the third field contains `#false` until the missile is fired and a `Posn` for the missile’s coordinates thereafter. See below.

```
(define-struct aim [ufo tank])
(define-struct fired [ufo tank missile])
```

The first one is for the time period when the player is trying to get the tank in position for a shot, and the second one is for representing states after the missile is fired. Before we can formulate a data definition for the complete game state, however, we need data representations for the tank, the UFO, and the missile.

Assuming constant definitions for such physical constants as `WIDTH` and `HEIGHT`, which are the subject of [exercise 95](#), we formulate the data definitions like this:

```
; A UFO is Posn.
; interpretation (make-posn x y) is the UFO's current location

(define-struct tank [loc vel])
; A Tank is (make-tank Number Number).
; interpretation (make-tank x dx) means the tank is at position
; (x, HEIGHT) and that it moves dx pixels per clock tick

; A Missile is Posn.
; interpretation (make-posn x y) is the missile's current location
```

Each of these data definitions describes nothing but a structure, either a newly defined one, `tank`, or a built-in data collection, `Posn`. Concerning the latter, it may surprise you a little bit that `Posns` are used

to represent two distinct aspects of the world. Then again we have used numbers (and strings and Boolean values) to represent many different kinds of information in the real world, so reusing a collection of structures such as `Posn` isn't a big deal.

Now we are in a position to formulate the data definitions for the state of the space invader game:

```
| ; A SIGS is one of:
| ; - (make-aim UFO Tank)
| ; - (make-fired UFO Tank Missile)
| ; interpretation represents the state of the space invader game
```

The shape of the data definition is that of an itemization. Each clause, however, describes the content of a structure type, just like the data definition for structure types we have seen so far. Still, this data definition shows that not every data definition comes with exactly one structure type definition; here one data definition involves two distinct structure type definitions.

The meaning of such a data definition is also straightforward. It introduces the name `SIGS` for a collection of data, namely all those structure instances that you can create according to the definition. So let us create some:

- Here is an instance that describes the tank maneuvering into position to fire the missile:

```
| | (make-aim (make-posn 20 10) (make-tank 28 -3))
```

- This one is just like the previous one but the missile has been fired:

```
| | | (make-fired (make-posn 20 10)
| | |   (make-tank 28 -3)
| | |   (make-posn 28 (- HEIGHT TANK-HEIGHT)))
```

Of course the capitalized names refer to the physical constants that you defined.

- Finally, here is one where the missile is close enough to the UFO for a collision:

```
| | | (make-fired (make-posn 20 100)
| | |   (make-tank 100 3)
| | |   (make-posn 22 103))
```

This example assumes that the canvas is more than 100 pixels tall.

Notice that the first instance of `SIGS` is generated according to the first clause of the data definition, and the second and third follow the second clause. Naturally the numbers in each field depend on your choices for global game constants.

Exercise 96. Explain why the three instances are generated according to the first or second clause of the data definition.

Exercise 97. Sketch how each of the three game states could be rendered assuming a 200 by 200 canvas.

The Design Recipe With a new way of formulating data definitions comes an inspection of the design recipe. This chapter introduces a way to combine two or more means of describing data, and the revised design recipe reflects this, especially the first step:

1. When do you need this new way of defining data? You already know that the need for itemizations is due to distinctions among different classes of information in the problem statement. Similarly, the need for structure-based data definitions is due to the demand to group several different pieces of information.

An itemization of different forms of data—including collections of structures—is required when your problem statement distinguishes different kinds of information and when at least some of these pieces of information consist of several different pieces.

One thing to keep in mind is that data definitions can be split. That is, if a particular clause in a data definition looks overly complex, it is acceptable to write down a separate data definition for this

clause and then just refer to this auxiliary definition via the name that it introduces.

And, as always, formulate data examples using the data definitions.

2. The second step remains the same. Formulate a function signature that mentions only the names of defined or built-in data collections, add a purpose statement, and create a function header.
3. Nothing changes for the third step. You still need to formulate functional examples that illustrate the purpose statement from the second step, and you still need one example per item in the itemization of the data definition.
4. The development of the template now exploits two different dimensions: the itemization itself and the use of structures in some of its clauses.

By the first, the body of the template consists of a `cond` expression that has as many `cond` clauses as the itemizations has items. Furthermore, you must add a condition to each `cond` clause that identifies the subclass of data in the corresponding item.

By the second, if an item deals with a structure, the template contains the selector expressions—in the `cond` clause that deals with the subclass of data described in the item.

When, however, you choose to describe the data with a separate data definition, then you do **not** add selector expressions. Instead, specialize the template for the other data definition to the task at hand and refer to that template. That is, indicate with a function call to this separate template that this subclass of data is being processed separately.

Before going through the work of developing a template, briefly reflect on the nature of the function. If the problem statement suggests that there are several tasks to be performed, it is likely that a composition of several, separately designed functions is needed instead of a template. In that case, skip the template step.

5. Fill the gaps in the template. It is easy to say, but the more complex you make your data definitions, the more complex this step becomes. The good news is that this design recipe helps us along, and there are many ways in which it can do so.

If you are stuck, fill the easy cases first and use default values for the others. While this makes some of the test cases fail, you are making progress and you can visualize this progress.

If you are stuck on some cases of the itemization, analyze the examples that correspond to those cases. Determine what the pieces of the template compute from the given inputs. Then consider how to combine these pieces, possibly with some global constants, to compute the desired output. Keep in mind that you might end up wishing for an auxiliary function.

Also if your template “calls” another template because the data definitions refer to each other, assume that the other function delivers what its purpose statement and its examples promise—even if this other function’s definition isn’t finished yet.

5. Test. If tests fail, determine what’s wrong: the function, the tests, or both. Go back to the appropriate step.

Go back to [Designing Functions](#), re-read the description of the simple design recipe, and compare it to the above. Also before you read on, try to solve the following exercise.

Exercise 98. A programmer has chosen to represent locations on the Cartesian plane as pairs (x, y) or as single numbers if the point lies on one of the axes:

Recall that the distance of a Cartesian point to the origin is the square root of the sum of the squares of its coordinates. For a location on either axis, the distance to the origin is the absolute value of the number.

```
; Location is one of:
; - Posn
; - Number
```

```
; interpretation Posn are positions on the Cartesian plane,
; Numbers are positions on either the x- or the y-axis.
```

Design the function `in-reach?`, which determines whether a given location's distance to the origin is strictly less than some constant R.

Note This function has no connection to any other material in this chapter.

Let us illustrate the design recipe with the design of a rendering function for our space invader game. Recall that a `big-bang` expression needs such a rendering function to turn the state of the world into an image after every clock tick, mouse click, or key stroke.

The signature of a rendering function looks like this

```
; SIGS -> Image
; adds TANK, UFO, and possibly the MISSILE to BACKGROUND
(define (si-render s) BACKGROUND)
```

where we assume that TANK, UFO, MISSILE and BACKGROUND are the requested image constants from [exercise 95](#). Recall that this signature is just an instance of the general signature for rendering functions, which always consume the collections of world states and produce some image.

s =	(si-render s) =
(make-aim (make-posn 10 20) (make-tank 28 -3))	
(make-fired (make-posn 10 20) (make-tank 28 -3) (make-posn 32 (- HEIGHT TANK-HEIGHT 10)))	
(make-fired (make-posn 20 100) (make-tank 100 3) (make-posn 22 103))	

Figure 26: Rendering space invader game states

We used a triangle that isn't available in BSL graphical library. No big deal.

Since the itemization in the data definition consists of two items, let us make three examples, using the data examples from above: see [figure 26](#). (Unlike the function tables found in mathematics books, this table is rendered vertically.) The left column contains sample inputs for our desired function, the right column lists the corresponding desired results. As you can see, we used the data examples from the first step of the design recipe, and they cover both items of the itemization.

Next we turn to the development of the template, the most important step of the design process. First, we know that the body of `si-render` must be a `cond` expression with two `cond` clauses. Following the design recipe, the two conditions are `(aim? s)` and `(fired? s)`, and they distinguish the two possible kinds of data that `si-render` may consume:

```
(define (si-render s)
  (cond
    [(aim? s) ...])
```

```
[(fired? s) ...]))
```

Second, we add selector expressions to each of the `cond` clauses that deals with structured input data. In this case, both clauses concern the processing of structures: `aim` and `fired`. The former comes with two fields and thus requires two selector expressions for the first `cond` clause, and the latter kind of structures consists of three values and thus demands three selector expressions:

```
(define (si-render s)
  (cond
    [(aim? s) (... (aim-tank s) ... (aim-ufo s) ...)]
    [(fired? s) (... (fired-tank s) ... (fired-ufo s)
                      ... (fired-missile s) ...))]))
```

And this is all there is to the template now.

The template contains nearly everything we need to finish our task. To complete the definition, we figure out for each `cond` line how to combine the values we have to compute the expected result. Beyond the pieces of the input, we may also use globally defined constants, for example, `BACKGROUND`, which is obviously of help here; primitive or built-in operations; and, if all else fails, wish-list functions, that is, we describe functions we wish we had.

Consider the first `cond` clause, where we have a data representation of a tank, that is, `(aim-tank s)`, and the data representation of a UFO, that is, `(aim-ufo s)`. From the first example in [figure 26](#), we know that we need to add the two objects to the background scenery. In addition, the design recipe suggests that if these pieces of data come with their own data definition, we are to consider defining helper (auxiliary) functions and to use those to compute the result:

```
... (tank-render (aim-tank s))
     (ufo-render (aim-ufo s) BACKGROUND) ...
```

Here `tank-render` and `ufo-render` are wish-list functions that deal with tanks and ufos, respectively:

```
; Tank Image -> Image
; adds t to the given image im
(define (tank-render t im) im)

; UFO Image -> Image
; adds u to the given image im
(define (ufo-render u im) im)
```

As you can see, the wish list entries arise from the second step of the design recipe.

With a bit of analogy, we can deal with the second `cond` clause in the same way:

```
(define (si-render s)
  (cond
    [(aim? s)
     (tank-render (aim-tank s)
                 (ufo-render (aim-ufo s) BACKGROUND))]
    [(fired? s)
     (tank-render (fired-tank s)
                 (ufo-render (fired-ufo s)
                             (missile-render (fired-missile s)
                                            BACKGROUND))))]))
```

Best of all, we can immediately re-use our wish list functions, `tank-render` and `ufo-render`, and all we need to add is a function for including a missile in the scene. The appropriate wish list entry is:

```
; Missile Image -> Image
; adds m to the given image im
(define (missile-render m im) im)
```

As above, the comment describes in sufficient detail what we want.

Exercise 99. Design the functions `tank-render`, `ufo-render`, and `missile-render`. Is the result of this expression

```
... (tank-render (fired-tank s)
    (ufo-render (fired-ufo s)
        (missile-render (fired-missile s)
            BACKGROUND))) ...
```

the same as the result of

```
... (ufo-render (fired-ufo s)
    (tank-render (fired-tank s)
        (missile-render (fired-missile s)
            BACKGROUND))) ...
```

When do the two expressions produce the same result?

Exercise 100. Design the function `si-game-over?` for use as the `stop-when` handler. The game stops if the UFO lands or if the missile hits the UFO. For both conditions, we recommend that you check for proximity of one object to another.

The `stop-when` clause allows for an optional second sub-expression, namely a function that renders the final state of the game. Design `si-render-final` and use it as the second part for your `stop-when` clause in the main function of [exercise 102](#).

Exercise 101. Design the function `si-move`, which is called for every clock tick. Accordingly it consumes an element of `SIGS` and produces another one. Its purpose is to move all objects according to their velocity.

Note that `random` is the first BSL primitive that is not a mathematical function. While functions in programming languages are inspired by mathematical functions, the two concepts are not identical.

For the random horizontal jumps of the UFO, use the BSL function `random`:

```
; N -> N
; produces a number in [0, n), possibly a different one
; each time the function is called
(define (random n) ...)
```

To test functions that employ `random`, use the following:

```
(define (si-move w)
  (si-move-proper w (create-random-number w)))
```

With this definition you separate the creation of a random number from the act of moving the game objects. While `create-random-number` produces possibly different results every time it is called on some fixed state of the world, `si-move-proper` is still guaranteed to return the same result when given the same input. In short, most of the code remains testable with the techniques you already know.

For `create-random-number`, consider using `check-random`:

```
; SIGS -> Number
; create a random number in case a UFO should perform a horizontal jump

(check-random (create-random-number a-sigs) (random DELTA-X))

(define (create-random-number w)
  ... (random DELTA-X) ...)
```

This test assumes that the function calls `random` once for `a-sigs`, in which case the expression in

`check-random`'s second sub-expression produces the same number.

Exercise 102. Design the function `si-control`, which plays the role of the key event handler. As such it consumes a game state and a `KeyEvent` and produces a new game state. This specific function reacts to three different key events:

- pressing the left arrow ensures that the tank moves left;
- pressing the right arrow ensures that the tank moves right; and
- pressing the space bar fires the missile if it hasn't been launched yet.

Once you have this function, you can define the `si-main` function, which uses `big-bang` to spawn the game-playing window.

Enjoy the game.

```
; SIGS.v2 -> Image
; renders the given game state and added it to BACKGROUND
(define (si-render.v2 s)
  (tank-render (fired-tank s)
    (ufo-render (fired-ufo s)
      (missile-render.v2 (fired-missile s)
        BACKGROUND))))
```

Figure 27: Rendering game states again

Data representations are rarely unique. For example, we could use a single structure type to represent the states of a space invader game—if we are willing to change the representations of missiles:

```
(define-struct sigs [ufo tank missile])
; SIGS.v2 (short for version 2)
; is (make-sigs UFO Tank MissileOrNot)
; interpretation represents the state of the space invader game

; A MissileOrNot is one of:
; - #false
; - Posn
; interpretation #false means the missile hasn't been fired yet;
; Posn says the missile has been fired and is at the specified location.
```

Unlike the first data representation for game states, this second version does not distinguish between before and after the missile launch. Instead, each state contains some data about the missile though this piece of data may just be `#false`, indicating that the missile hasn't been fired yet.

Of course, the functions for this second data representation of states differ from the functions for the first one. In particular, `SIGS.v2` consuming functions do not use a `cond` expression because there is only one kind of element in the collection. Put differently, the design recipe for structures from [Designing with Structures](#) suffices. [Figure 27](#) shows the (partial) result.

In contrast, the design of functions using `MissileOrNot` requires the recipe from this section. Let us look at the design of `missile-render2`, whose job it is to add a missile to an image. Here is the header material:

```
; MissileOrNot Image -> Image
; adds the missile image to sc for m
(define (missile-render.v2 m scene)
  scene)
```

As for examples, we must consider at least two cases: one when `m` is `#false` and another one when `m` is a `Posn`. In the first case, the missile hasn't been fired, which means that no image of a missile is to be added to the given scene. In the second case, the missile's position is specified and that is where the image of the missile must show up:

<code>m =</code>	<code>scene =</code>	<code>(missile-render.v2 m scene) =</code>
<code>false</code>		
<code>(make-posn</code>		
<code>32</code>		
<code>(- HEIGHT</code>		
<code>TANK-HEIGHT</code>		
<code>10))</code>		

This table's first two columns specify the inputs to `missile-render.v2`, while the last column shows us what the expected outcome is.

Exercise 103. Turn the examples into test cases.

Now we are ready to develop the template. Because of the data definition for the major argument (`m`) is an itemization with two items, the function body is likely to consist of a `cond` expression with two clauses:

```
(define (missile-render.v2 m scene)
  (cond
    [(boolean? m) ...]
    [(posn? m) ...]))
```

Following again the data definition, the first `cond` clause checks whether `m` is a `Boolean` value and the second one checks whether it is an element of `Posn`. And yes, if someone were to accidentally apply `missile-render.v2` to `#true` and some image, the function would use the first `cond` clause to compute the result. We have more to say on such errors below.

The second template step requests selector expressions in all those `cond` clauses that deal with structures. In our example, this is true for the second clause and the selector expressions extract the x- and y-coordinates from the given `Posn`:

```
(define (missile-render.v2 m scene)
  (cond
    [(boolean? m) ...]
    [(posn? m) (... (posn-x m) ... (posn-y m) ...)]))
```

Compare this template with the one for `si-render` above. The data definition for the latter deals with two distinct structure types, and therefore the function template for `si-render` contains selector expressions in both `cond` clauses. The data definition for `MissileOrNot`, however, mixes items that are plain values with items that describe structures. Both kinds of definitions are perfectly fine; the key for you is to follow the recipe and to find a code organization that matches the data definition.

Here is the complete function definition:

```
(define (missile-render.v2 m scene)
  (cond
    [(boolean? m) scene]
    [(posn? m) (place-image MISSILE (posn-x m) (posn-y m) scene)]))
```

Doing this step by step, you first work on the easy clauses, in this function that's the first one. Since it says the missile hasn't been fired, the function returns the given `scene`. For the second clause, you need to remember that `(posn-x m)` and `(posn-y m)` select the coordinates for the image of the missile. This function must add `MISSILE` to `scene`, so you have to figure out the best combination of primitive operations and your own functions to combine the four values. Choosing this combining operation is where your creative insight as a programmer comes into play.

Exercise 104. Design the functions `si-move.v2`, `si-game-over.v2?`, and `si-control.v2` to

complete the game for this second data definition.

Exercise 105. Develop a data representation for the kinds of people you find at a university: student (first and last name, gpa), professor (first and last name, tenure status), and staff (first and last name, salary group). Then develop a template for functions that consume the representation of a person.

Exercise 106. Develop a data representation for the following four kinds of zoo animals:

- **spiders**, whose relevant attributes are the number of remaining legs (we assume that spiders can lose legs in accidents) and the space they need in case of transport;
- **elephants**, whose only attributes are the space they need in case of transport;
- **boa constrictor**, whose attributes include length and girth; and
- **armadillo**, for whom you must determine appropriate attributes; they need to include attributes that determine space needed for transportation.

Develop a template for functions that consume representations of zoo animals.

Design the function `fits?`. The function consumes a zoo animal and the volume of a cage. It determines whether the cage is large enough for the animal.

Exercise 107. Your home town manages a fleet of vehicles: automobiles, vans, buses, SUVs, and trucks. Develop a data representation for vehicles. The representation of each vehicle must describe the number of passengers that it can comfortably accommodate, its license plate, and its fuel consumption (miles per gallon).

Develop a template for functions that consume representations of vehicles.

Note In reality, each of these structure type definitions contains additional fields that are specific to the kind of vehicle it describes. We ignore this idea here.

Exercise 108. Some program contains the following data definition:

```
; A Coordinate is one of:
; - a negative number
;   interpretation a point on the Y axis, distance from top
; - a positive number
;   interpretation a point on the X axis, distance from left
; - a Posn
;   interpretation a point in a scene, usual interpretation
```

Make up at least two data examples per clause in the data definition. For each of the examples, explain its meaning with a sketch of a canvas.

6.2 Mixing up Worlds

This section suggests several design problems for world program, starting with simple extension exercises concerning our virtual pets.

Exercise 109. In [More Virtual Pets](#) we discuss the creation of virtual pets that come with happiness gauges. One of the virtual pets is a cat, the other one is a chameleon. Each program is dedicated to a single pet, however.

Design the `cat-cham` world program, which “walks” either cats or chameleons across the canvas. That is, `cat-cham` consumes both a starting location and an animal:

```
; A VAnimal is either
; - a VCat
; - a VCham
```

Here `VCat` and `VCham` are your data definitions for exercises [exercise 89](#) and [exercise 93](#), respectively.

It then walks the given animal from left to right.

Given that `VAnimal` is the collection of world states, you need to design

- a rendering function from `VAnimal` to `Image`;
- a function for handling clock ticks, from `VAnimal` to `VAnimal`;
- and a function for dealing with key events so that you can feed and pet and colorize your pet—as applicable.

It remains impossible to change the color of a cat or to pet a chameleon.

Exercise 110. Design the `cham-and-cat` program, which deals with both a virtual cat **and** a virtual chameleon. You need a data definition for a “zoo” containing both animals and functions for dealing with it.

The problem statement leaves open how keys manipulate the two animals. Here are two possible interpretations:

1. Each key event goes to both animals.
2. Each key event applies to only one of the two animals.

For this alternative, you need to extend the data definition of a two-animal zoo to include a focus. The *focus* determines which of the two animals can currently be manipulated. To switch focus, have the key-handling function interpret "`k`" for “kitty” and "`l`" for lizard. Once a player hits "`k`", the following keystrokes apply to the cat only—until the player hits "`l`".

Choose one of the alternatives and design the appropriate program.

Exercise 111. In its default state, a pedestrian crossing light shows an orange person standing on a red background. When it is time to allow pedestrian to cross the street, the light receives a signal and switches to a green, walking person. This phase lasts for 10 seconds. After that the light displays the digits `9`, `8`, ..., `0` with odd numbers colored orange and even numbers colored green. When the countdown reaches 0, the light switches back to its default state.

Design a world program that implements such a pedestrian traffic light. The light switches from its default state when you press the space bar on your keyboard. All other transitions must be reactions to clock ticks. You may wish to use the following images

or you can make up your own stick figures with the image library.

Figure 28: A finite state machine as a diagram

Exercise 112. Design a world program that recognizes a pattern in a sequence of [KeyEvents](#). Specifically, it must accept any sequence that starts with "[a](#)", is followed by an arbitrarily long mix of "[b](#)" and "[c](#)", and ends in "[d](#)". As soon as it encounters this "[d](#)", the program stops running. If these four keys are out of order or if any other key is hit, the program must also shut down.

Clearly, "[acbd](#)" is one example of an acceptable string; "[ad](#)" and "[abcd](#)" are two others. Of course, "[da](#)", "[aa](#)", or "[d](#)" do not match.

Initially your program shows a 100 by 100 white rectangle. Once your program has seen an "[a](#)", it displays a yellow rectangle of the same size. After encountering the final "[d](#)", the color of the rectangle turns green. If any "bad" key event occurs, the program displays a red rectangle.

Hint Your solution implements a so-called finite state machine (FSM), an idea briefly mentioned in [Finite State Worlds](#) as one principle underneath the design of world programs. As the name says, a FSM program may be in one of a finite number of states. The first state is called an *initial state*. Each key event causes the machine to re-consider its current state; it may transition to the same state or to another one. When your program recognizes a proper sequence of key events, it transitions to a *final state*.

While the left column displays the conventional definition, the right one show how a seasoned designer would use constants to introduce these strings.

For a sequence-recognition problem, states typically represent the letters that the machine expects to see next:

conventional	defined abbreviations
<code>; ExpectsToSee is one of:</code>	<code>; ExpectsToSee is one of:</code>
<code>; - "start, expect to see an 'a' next"</code>	<code>; - AA</code>
<code>; - "expect to see: 'b', 'c', or 'd'"</code>	<code>; - BC</code>
<code>; - "encountered a 'd', finished"</code>	<code>; - DD</code>
<code>; - "error, user pressed illegal key"</code>	<code>; - ER</code>

```
(define AA "start, ...")
(define BC "expect ...")
(define DD "encountered ...")
(define ER "error, ...")
```

Note the state that says an illegal input has been encountered. [Figure 28](#) shows how to think of these states and their relationships in a diagrammatic manner. Each node corresponds to one of the four finite states; each arrow specifies which `KeyEvent` causes the program to transition from one state to another.

History In the 1950s, Stephen C. Kleene, whom we would call a computer scientist, invented *regular expressions* as a notation for the problem of recognizing text patterns. For the above problem, Kleene would write

$$a \ (b|c)^* \ d$$

which really just means `a` followed by either `b` or `c` arbitrarily often until `d` is encountered.

6.3 Input Errors

One central point of this chapter concerns the role of predicates. They are critical when you must design functions that process distinct kinds of data, mixes of data. Such mixes come up naturally when your problem statement mentions many different kinds of information, but they also come up when you hand your functions and programs to others. After all, you know and respect your data definitions and function signatures. You never know, however, what your friends and colleagues do and you especially don't know how someone without knowledge of BSL and programming uses your programs. This section therefore presents one way of protecting programs from inappropriate inputs.

Let us demonstrate this point with a simple program, a function for computing the area of a disk:

It is a form of self-delusion to think that we always respect our own function signatures. Calling a function on the wrong kind of data happens to the best of us, and you will make this mistake, too. While many languages are like BSL and expect the programmer to live up to such high standards, others support some way of checking function signatures at the cost of some additional complexity, especially for beginners. We ignore these complications for a while.

```
; Number -> Number
; computes the area of a disk with radius r
(define (area-of-disk r)
  (* 3.14 (* r r)))
```

Our friends may wish to use this function, especially for some of their geometry homework. Unfortunately, when our friends use this function, they may accidentally apply it to a string rather than a number. When that happens, the function stops the program execution with a mysterious error message:

```
> (area-of-disk "my-disk")
*: expects a number as 1st argument, given "my-disk"
```

With predicates, you can prevent this kind of cryptic error message and signal an informative error of our own.

Specifically, we can define checked versions of our functions, when we wish to hand them to our friends. Because our friends may not know BSL or may ignore our data definitions and signatures, we must expect that they apply this *checked function* to arbitrary BSL values: numbers, Boolean values, strings, images, `Posns`, and so on. Although we cannot anticipate which structure types will be defined in BSL, we know the rough shape of the data definition for the collection of all BSL values:

```
; Any BSL value is one of:
; - Number
```

```
; - Boolean
; - String
; - Image
; - (make-posn Any Any)
; ...
; - (make-tank Any Any)
; ...
```

As discussed in [The Universe of Data](#), the data definition for [Any](#) must be open-ended and any structure instance may contain an arbitrary BSL value in each of its fields. The second aspect implies that the sketch of the data definition of [Any](#) must refer to itself; but don't worry about such self-references yet. For now, understand that the data definition is a mixed itemization.

Based on this itemization, the template for a checked function has the following rough shape:

```
; Any -> ???
(define (checked-f v)
  (cond
    [(number? v) ...]
    [(boolean? v) ...]
    [(string? v) ...]
    [(image? v) ...]
    [(posn? v) (...(posn-x v) ... (posn-y v) ...)]
    ...
    [(tank? v) ...] ; which selectors are needed here?
    ...))
```

Of course, nobody can list all clauses of this definition; fortunately, that's not necessary. What we do know is that for all those values in the class of values for which the original function is defined, the checked version must produce the same results; for all others, it must signal an error.

Concretely, our sample function `checked-area-of-disk` consumes an arbitrary BSL value and uses `area-of-disk` to compute the area of a disk if the input is a number. It must stop with with an error message otherwise; in BSL we use the function `error` to accomplish this. The `error` function consumes a string and stops the program:

```
(error "area-of-disk: number expected")
```

Hence the rough definition of `checked-area-of-disk` looks like this:

```
(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [(boolean? v) (error "area-of-disk: number expected")]
    [(string? v) (error "area-of-disk: number expected")]
    [(image? v) (error "area-of-disk: number expected")]
    [(posn? v) (error "area-of-disk: number expected")]
    ...
    [(tank? v) (error "area-of-disk: number expected")]
    ...))
```

The use of `else` helps us to finish this function definition in the natural way:

```
; Any -> Number
; computes the area of a disk with radius v,
; if v is a number
(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [else (error "area-of-disk: number expected"))])
```

And just to make sure we get what we want, here is an interaction with this function:

```
> (checked-area-of-disk "my-disk")
area-of-disk: number expected
```

Compare this result with the one at the beginning of this section.

Writing checked functions is important if we distribute our programs for others to use. Designing programs that work properly, however, is far more important. This book focuses on the design process for program proper design and, to do this without distraction, we agree that we always adhere to data definitions and signatures. At least, we almost always do so, and on rare occasions we may ask you to design checked versions of a function or a program.

Exercise 113. A checked version of `area-of-disk` can also enforce that the arguments to the function are positive numbers, not just arbitrary numbers. Modify `checked-area-of-disk` in this way.

Exercise 114. Take a look at these definitions:

```
(define-struct vec [x y])

; A vec is
; (make-vec PositiveNumber PositiveNumber)
; interpretation represents a velocity vector
```

Develop the function `checked-make-vec`, which is to be understood as a checked version of the primitive operation `make-vec`. It ensures that the arguments to `make-vec` are positive numbers, and not just arbitrary numbers. In other words, `checked-make-vec` enforces our informal data definition.

Predicates You might wonder how you can design your own predicates. After all, checked functions really seem to have this general shape:

```
; Any -> ...
; checks that a is a proper input for function g
(define (checked-g a)
  (cond
    [(XYZ? a) (g a)]
    [else (error "g: bad input")]))
```

where `g` itself is defined like this:

```
; XYZ -> ...
(define (g some-x) ...)
```

We are assuming that there is a data definition labeled `XYZ`, and that `(XYZ? a)` produces `#true` precisely when `a` is an element of `XYZ` and `#false` otherwise.

For `area-of-disk`, which consumes `Numbers`, the appropriate predicate is clearly `number?`. In contrast, for some functions like `missile-render` from above, we clearly need to define our own predicate because `MissileOrNot` is a made-up, not a built-in data collection. So let us design a predicate for `MissileOrNot`.

We recall the signature for predicates:

```
; Any -> Boolean
; is a an element of the MissileOrNot collection
(define (missile-or-not? a) #false)
```

It is a good practice to use questions as purpose statements for predicates, because applying a predicate is like asking a question about a value. The question mark “?” at the end of the name also reinforces this idea; some people may tack on “huh” to pronounce the name of such functions.

Making up examples is also straightforward:

```
(check-expect (missile-or-not? #false) #true)
(check-expect (missile-or-not? (make-posn 10 2)) #true)
(check-expect (missile-or-not? "yellow") #false)
```

The first two examples recall that every element of `MissileOrNot` is either `#false` or some `Posn`. The third test says that strings aren't elements of the collection. Here are three more tests:

```
(check-expect (missile-or-not? #true) #false)
(check-expect (missile-or-not? 10) #false)
(check-expect (missile-or-not? (circle 3 "solid" "red")) #false)
```

Explain the expected answers!

Since predicates consume all possible BSL values, their templates are just like the templates for checked functions:

```
(define (missile-or-not? v)
  (cond
    [(number? v) ...]
    [(boolean? v) ...]
    [(string? v) ...]
    [(image? v) ...]
    [(posn? v) (... (posn-x v) ... (posn-y v) ...)]
    ...
    [(tank? v) ...] ; which selectors are needed here?
    ...))
```

And as with checked functions, a predicate doesn't need all possible `cond` lines. Only those that might produce `#true` are required:

```
(define (missile-or-not? v)
  (cond
    [(boolean? v) ...]
    [(posn? v) (... (posn-x v) ... (posn-y v) ...)]
    [else #false]))
```

All other cases are summarized via an `else` line that produces `#false`.

Given the template, the definition of `missile-or-not?` is a simple matter of thinking through each case:

```
(define (missile-or-not? v)
  (cond
    [(boolean? v) (boolean=? #false v)]
    [(posn? v) #true]
    [else #false]))
```

Only `#false` is a legitimate `MissileOrNot`, `#true` isn't. We express this idea with `(boolean=? #false v)`. An alternative is to use `(false? v)` as the predicate:

```
(define (missile-or-not? v)
  (cond
    [(false? v) #true]
    [(posn? v) #true]
    [else #false]))
```

Naturally all elements of `Posn` are also members of `MissileOrNot`, which explains the `#true` in the second line.

Exercise 115. Design predicates for the following data definitions from the preceding section: `SIGS`, `Coordinate` (exercise 108), and `VAnimal`.

To wrap up this chapter, let us mention two important predicates that you may wish to use in your world programs:

- `key?`, which determines whether the given value is a `KeyEvent`; and
- `mouse?`, which finds out whether some piece of data is an element of `MouseEvt`.

Check out their documentation.

Checking the World A world program is a place where things can easily go wrong. Even though we just agreed to trust that our functions are always applied to the proper kind of data, in a world program we may juggle too many things at once to have that much trust in ourselves. When we design a world program that takes care of clock ticks, mouse clicks, key strokes, and rendering, it is just too easy to get one of those interplays wrong. Of course, going wrong doesn't mean that BSL recognizes the mistake immediately. For example, one of our functions may produce a result that isn't quite an element of your data representation for world states. Nevertheless, **big-bang** accepts this piece of data and holds on to it, until the next event takes place. Then the next event handler receives this not-quite correct piece of data and it may fail because it really works only for proper world state representations. A worst-case scenario is that even the second and third and fourth event handling step copes with not-quite proper worlds, and it all blows up much later in the process.

To help with this kind of problem, **big-bang** comes with an optional **check-with** clause which accepts a predicate for world states. If, for example, we chose to represent all world states with **Number**, we could express this fact easily like this:

```
(define (main s0)
  (big-bang s0 ... [check-with number?] ...))
```

As soon as any event handling function produces something other than a number, the world stops with an appropriate error message.

A **check-with** clause is even more useful when the data definition is not just a class of data with a built-in predicate like **number?** but something subtle such as this interval definition:

```
; A UnitWorld is a number
; between 0 (inclusive) and 1 (exclusive).
```

In that case you want to formulate a predicate for this interval:

```
; Any -> Boolean
; is x between 0 (inclusive) and 1 (exclusive)

(check-expect (between-0-and-1? "a") #false)
(check-expect (between-0-and-1? 1.2) #false)
(check-expect (between-0-and-1? 0.2) #true)
(check-expect (between-0-and-1? 0.0) #true)
(check-expect (between-0-and-1? 1.0) #false)

(define (between-0-and-1? x)
  (and (number? x) (≤ 0 x) (< x 1)))
```

With this predicate you can now monitor every single transition in your world program:

```
(define (main s0)
  (big-bang s0 ... [check-with between-0-and-1?] ...))
```

If any of the world-producing handlers creates a number outside of the interval, or worse, a non-numeric-value, our program discovers this mistake immediately and gives us a chance to fix the mistake.

Exercise 116. Use the predicates from [exercise 115](#) to check

1. the space invader world program,
2. the virtual pet program ([exercise 109](#)),
3. the editor program ([A Graphical Editor](#)).

Equality Predicates Let us wrap up this section with a look at *equality predicates*, that is, functions that compare two elements of the same collection of data. Recall the definition of **TrafficLight**, which is the

http://www.ccs.neu.edu/home/matthias/HtDP2e/Draft/part_one.html

collection of three strings: "red", "green", and "yellow". Here is one way to define light=?

```
; TrafficLight TrafficLight -> Boolean
; are the two (states of) traffic lights equal

(check-expect (light=? "red" "red") #true)
(check-expect (light=? "red" "green") #false)
(check-expect (light=? "green" "green") #true)
(check-expect (light=? "yellow" "yellow") #true)

(define (light=? a-value another-value)
  (string=? a-value another-value))
```

When we click *RUN*, all tests succeed but unfortunately other interactions fail:

```
> (light=? "salad" "greens")
#false
> (light=? "beans" 10)
string=?: expects a string as 2nd argument, given 10
```

Compare these interactions with other equality predicates that are built in:

```
> (boolean=? "#true" 10)
boolean=?: expects a boolean as 1st argument, given "#true"
```

Try (`string=? 10 #true`) and (`= 20 "help"`) on your own. All of them signal an error about being applied to the wrong kind of argument.

A checked version of light=? enforces that both arguments belong to `TrafficLight`; if not, it signals an error like those that built-in equality predicates issue. Adding this checks in turn demands a predicate for `TrafficLight`, whose name we abbreviate to `light?` for simplicity:

Keep in mind that "red" is different from "Red" or "RED". The case of characters in strings matter.

```
; Any -> Boolean
; is the given value an element of TrafficLight
(define (light? x)
  (cond
    [(string? x) (or (string=? "red" x)
                      (string=? "green" x)
                      (string=? "yellow" x))]
    [else #false]))
```

Now we can wrap up the revision of light=? by just following our original analysis. First, the function determines that the two inputs are elements of `TrafficLight`; if not it uses `error` to signal the mistake:

```
; Any Any -> Boolean
; are the two values elements of TrafficLight and,
; if so, are they equal

(check-expect (light=? "red" "red") #true)
(check-expect (light=? "red" "green") #false)
(check-expect (light=? "green" "green") #true)
(check-expect (light=? "yellow" "yellow") #true)

(define (light=? a-value another-value)
  (if (and (light? a-value) (light? another-value))
      (string=? a-value another-value)
      (error "traffic light expected, given: some other value")))
```

Exercise 117. Revise light=? so that the error message specifies which of the two arguments aren't

elements of [TrafficLight](#).

While it is unlikely that your programs will use `light=?`, your world programs ought to use `key=?` and `mouse=?`, two equality predicates that we briefly mentioned at the end of the last subsection. Naturally, `key=?` is an operation for comparing two `KeyEvents`; similarly, `mouse=?` compares two `MouseEvt`s. While both kinds of events are represented as strings, it is important to realize that not all strings represent key events or mouse events.

We recommend using `key=?` in key event handlers and `mouse=?` in mouse event handlers from now on. The use of `key=?` in a key event handler ensures that the function really compares strings that represent key events and not arbitrary strings. As soon as, say, the function is accidentally applied to `"hello\n world"`, `key=?` signals an error and thus informs us that something is wrong.

7 Summary

In this first part of the book, you learned a bunch of simple but important lessons. Here is a summary:

1. A **good programmer** designs programs. A bad programmer tinkers until the program seems to work.
2. The **design recipe** has two important dimensions. One concerns the process of design, that is, the sequence of steps to be taken. The other explains how the chosen data representation influences the design process.
3. Every well-designed program consists of many constant definitions, structure type definitions, data definitions, and function definitions. For **batch programs**, one function is the “main” function, and it typically composes several other functions to perform its computation. For **interactive programs**, the `big-bang` function plays the role of the main function; it specifies the initial state of the program, an image-producing output function, and at most three event handlers: one for clock ticks, one for mouse clicks, one for key events. In both kinds of programs, function definitions are presented “top down,” starting with the main function, followed by those functions mentioned in the main function, and so on.
4. Like all programming languages, *Beginning Student Language* comes with a **vocabulary and a grammar**. Programmers must be able to determine the **meaning** of each sentence in a language so that they can anticipate how the program performs its computation when given an input.
5. Programming languages, including BSL, come with a rich set of libraries so that programmers don’t have to re-invent the wheel all the time. A programmer should become comfortable with the functions that a library provides, especially their signatures and purposes statements. Doing so simplifies life.
6. A programmer must get to know the “tools” that a chosen programming language offers. These tools are either part of the language—such as `cond` or `max`—or they are “imported” from a library. In this spirit, make sure you understand the following terms: **structure type** definition; **function** definition; **constant** definition; **structure instance**; **data definition**; **big-bang**, and event handling function.

v.6.3.0.2

Intermezzo: BSL

Fixed-Size Data deals with BSL as if it were a natural language. It introduces the “basic words” of the language, suggests how to compose the “words” into “sentences,” and appeals to your knowledge of algebra for an intuitive understanding of these “sentences.” While this kind of introduction works to some extent, truly effective communication requires some formal study of a language.

A programming language is in many ways like a natural language. It has a vocabulary and a grammar, though programmers use the word *syntax* for these. The vocabulary is the collection of those words from which we compose sentences in our language. A sentence in a programming language is an expression or a definition; the language’s grammar dictates how to form these phrases.

Not all grammatical sentences are meaningful—neither in English nor in a programming language. For example, the English sentence “the cat is round” is a meaningful sentence, but “the brick is a car” makes no sense even though it is completely grammatical. To determine whether a sentence is meaningful, we must know the *meaning* of a language; programmers call this *semantics*.

In this intermezzo, we discuss the meaning of BSL programs as if it were an extension of the familiar laws of arithmetic and algebra. After all, computation starts with this form of simple mathematics, and we should understand the connection between this mathematics and computing. The

The intermezzos of this book introduce ideas from programming languages that programmers must eventually understand and appreciate but are secondary to program design.

first three sections present the syntax and semantics of a good portion of BSL. Based on this new understanding of BSL, the fourth resumes our discussion of errors. The remaining sections expand this understanding to the complete language; the last one expands the tools for expressing tests.

BSL Vocabulary

Figure 29 introduces and defines BSL’s basic vocabulary. It consists of literal constants, such as numbers or Boolean values; names that have meaning according to BSL, for example, `cond` or `+`; and names to which programs can give meaning via `define` or function parameters.

A *name* or a *variable* is a sequence of characters not including a space or one of the following: `"`, `'`, ```, `(`, `)`, `[`, `]`, `{`, `}`, `|`, `:`, `#`:

- A *primitive* is a name to which BSL assigns meaning, for example, `+` or `sqrt`.
- A *variable* is a name without preassigned meaning.

A *value* is one of:

- A *number* is one of: `1`, `-1`, `3/5`, `1.22`, `#i1.22`, `0+1i`, and so on. The syntax for BSL numbers is complicated because it accommodates a range of formats: positive and negative numbers, fractions and decimal numbers, exact and inexact numbers, real and complex numbers, numbers in bases other than `10`, and more. Understanding the precise notation for numbers requires a thorough understanding of grammars and parsing, which is out of scope for this intermezzo.
- A *boolean* is one of: `#true` or `#false`.
- A *string* is one of: `" "`, `"a"`, `"doll"`, `"he says \"hello world\" ok"`, and so on. In general, it is a sequence of characters enclosed by a pair of `"`.
- An *image* is a png, jpg, tiff, and various other formats. We intentionally omit a precise definition.

We use `v`, `v-1`, `v-2` and the like when we wish to say “any possible value.”

Figure 29: BSL core vocabulary

Each of the explanations defines a set via a suggestive iteration of its elements. Although it is possible to specify these collections in their entirety and in a formal manner, we consider this superfluous here and trust in your intuition. Keep in mind that each of these sets may come with some extra elements.

BSL Grammar

Figure 30 shows a large part of the BSL grammar, which—in comparison to other languages—is extremely simple. As to BSL’s expressive power, don’t let the looks deceive you. The first action item is to discuss how to read such grammars. Each line with a `=` introduces a *syntactic category*; the best way to pronounce `=` as “is one of” and `|` as “or.” Wherever you see three dots, imagine as many repetitions of what precedes the dots as you wish. This means, for example, that a *program* is either nothing or a single occurrence of *def-or-expr* or a sequence of two of them:

Reading a grammar aloud makes it sound like a data definition. One could indeed use grammars to write down many of our data definitions.

```
def-or-expr
def-or-expr
```

or three, four, five, or however many. Since this example is not particularly illuminating, let us look at the second syntactic category. It says that *definition* is either

```
(define (variable variable) expr)
```

because “as many as you wish” includes zero, or

```
(define (variable variable variable) expr)
```

which is one repetition, or

```
(define (variable variable variable variable) expr)
```

which uses two.

```

program = def-or-expr ...

def-or-expr = definition
| expr

definition = (define (variable variable variable ...) expr)

expr = variable
| value
| (primitive expr expr ...)
| (variable expr expr ...)
| (cond [expr expr] ... [expr expr])
| (cond [expr expr] ... [else expr])

```

Figure 30: BSL core grammar

The final point about grammars concerns the three “words” that come in a distinct font: `define`, `cond`, and `else`. According to the definition of BSL vocabulary, these three words are names. What the vocabulary definition does not tell us is that these names have a pre-defined meaning. In BSL, these words serve as markers that differentiate some compound sentences from others, and in acknowledgment of their role, such words are called *keywords*.

Now we are ready to state the purpose of a grammar. The grammar of a programming language dictates how to form sentences from the vocabulary of the grammar. Some sentences are just elements of vocabulary. For example, according to figure 30 42 is a sentence of BSL:

- The first syntactic category says that a program is a *def-or-expr*. Expressions may refer to the definitions.
- The second definition tells us that a *def-or-expr* is either a *definition* or an *expr*.
- The last definition lists all ways of forming an *expr*, and the second one is *value*.

In DrRacket, a program really consists of two distinct parts: the definitions area and the expressions in the interactions area.

Since we know from figure 29 that 42 is a value, we have a complete confirmation.

The interesting parts of the grammar show how to form compound sentences, which are sentences built from other sentences. For example, the *definition* part tells us that a function definition is formed by using “(”, followed by the keyword `define`, followed by another “(”, followed by a sequence of at least two variables, followed by “)”, followed by an *expr*, and closed by a right parenthesis “)” that matches the very first one. You can see from this example how the leading keyword `define` distinguishes definitions from expressions.

Expressions, called *expr*, come in six flavors: variables, constants, primitive applications, (function) applications, and two varieties of `conditionals`. While the first two are atomic sentences, the last four are compound sentences. Note how the keyword `cond` distinguishes conditional expressions from primitive and function applications.

Here are three examples of expressions: `"all"`, `x`, and `(f x)`. The first one belongs to the class of strings and is therefore an expression. The second is a variable, and every variable is an expression. The third is a function application, because `f` and `x` are variables.

In contrast, the following parenthesized sentences are not expressions: `(f define)`, `(cond x)`, and `((f 2) 10)`. The first one partially matches the shape of a function application but it uses `define` as if it were a variable. The second one fails to be a correct `cond` expression because it contains a variable as the second item and not a pair of expressions surrounded by parentheses. The last one is neither a conditional nor an application because the first part is an expression.

Finally, you may notice that the grammar does not mention white space: blank spaces, tabs, and newlines. BSL is a permissive language. As long as there is some white space between the elements of any sequence in a program, DrRacket can understand your BSL programs.

Keep in mind that two kinds of readers study your BSL programs: people and DrRacket.

Good programmers, however, may not like what you write. These programmers use white space to make their programs easily readable. Most importantly, they adopt a style that favors human readers over the software applications that process programs (such as DrRacket). They pick up this style from careful reading code examples in books, paying attention to how it is formatted.

Exercise 118. Explain why the following sentences are syntactically legal expressions

1. `x`
2. `(= y z)`
3. `(= (= y z) 0)`

|

Exercise 119. Explain why the following sentences are syntactically illegal

1. `(3 + 4)`
2. `number?`
3. `(l)`
4. `(x)`

|

Exercise 120. Explain why the following sentences are syntactically legal definitions

1. `(define (f x) x)`
2. `(define (f x) y)`
3. `(define (f x y) 3)`

|

Exercise 121. Explain why the following sentences are syntactically illegal

1. `(define (f "x") x)`
2. `(define (f x y z) (x))`
3. `(define (f) 10)`

|

Exercise 122. Discriminate the legal from the illegal sentences in the following list:

1. (x)
2. (+ 1 (not x))
3. (+ 1 2 3)

Explain why the sentences are legal or illegal. Determine whether they belong the category *expr* or *definition*. ■

Note on Grammatical Terminology The components of compound sentences have names. We have introduced some of these names on an informal basis. [Figure 31](#) provides a summary of the conventions.

```

; function application:
(function argument ... argument)

; function definition:
(define (function-name parameter ... parameter)
  function-body)

; conditional expression:
(cond
  cond-clause
  ...
  cond-clause)

; cond clause
[condition answer]

```

Figure 31: Syntactic naming conventions

In addition to the terminology of [figure 31](#), we say *function header* for second component of a definition. Accordingly, the expression component is called *function body*. People who consider programming languages as a form of mathematics use *left-hand side* for a header and *right-hand side* for the body. On occasion, you also hear or read the term *actual arguments* for the arguments in a function application.

BSL Meaning

When you hit the return key on your keyboard and ask DrRacket to evaluate an expression, it uses the laws of arithmetic and algebra to obtain a *value*. For the variant of BSL treated so far, [figure 29](#) defines grammatically what a value is—the set of values is just a subset of all expressions. The set includes [Booleans](#), [String](#), and [Images](#).

The rules of evaluation come in two categories. First, we need an infinite number of rules like those of arithmetic to evaluate applications of primitives:

```

(+ 1 1) == 2
(- 2 1) == 1
...

```

But BSL arithmetic is more general than just number crunching. It also includes rules for dealing with Boolean values, strings, and so on:

```
(not #true) == #false
(string=? "a" "a") == #true
...
```

Remember `==` says that two expressions are equal according to the laws of computation in BSL.

Like in algebra, you can always replace equals with equals in expressions:

```
(boolean? (= (string-length (string-append "hello" "world")) (+ 1 3)))
==
(boolean? (= (string-length (string-append "hello" "world")) 4))
==
(boolean? (= (string-length "helloworld") 4))
==
(boolean? (= 10 4))
==
(boolean? #false)
==
true
```

Second, we need a rule from algebra to understand the application of a functions to arguments. Suppose the program contains the definition

```
(define (f x-1 ... x-n)
  f-body)
```

Then an application of a function is governed by the law:

```
(f v-1 ... v-n) == f-body
; with all occurrences of x-1 ... x-n
; replaced with v-1 ... v-n, respectively
```

Due to the history of languages such as BSL, we refer to this rule as the *beta* or *beta-value* rule.

This rule is as general as possible, so it is best to look at a concrete example. Say the definition is

```
(define (poly x y)
  (+ (expt 2 x) y))
```

Then the first step in an evaluation of `(poly 3 5)` uses beta:

```
(poly 3 5) == (+ (expt 2 3) 5) ... == (+ 8 5) == 13
```

For the rest, DrRacket uses plain arithmetic.

In addition to beta, we need a couple of rules that help us determine the value of `cond` expressions. These rules are algebraic rules but are not a part of the standard algebra curriculum. When the first condition is `#false`, then the first `cond`-line disappears:

<code>(cond</code> <code>[#false ...]</code> <code>[condition-1 answer-1]</code> <code>...)</code>	<code>==</code> <code>(cond</code> <code>; the first line disappeared</code> <code>[condition-1 answer-1]</code> <code>...)</code>
---	--

This rule has the name `condfalse`. When the first condition is `#true`, DrRacket picks the

right-hand side of this `cond` clause:

```
(cond                ==      answer
  [#true answer]
  [condition-1 answer-1]
  ...)
```

We call this rule `condtrue`. It also applies when the condition is `else`.

Consider the following evaluation:

```
(cond
  [(zero? 3) 1]
  [(= 3 3) (+ 1 1)]
  [else 3])
== ; by plain arithmetic and “equals for equals”
(cond
  [#false 1]
  [= 3 3] (+ 1 1)
  [else 3])
== ; by rule condfalse
(cond
  [= 3 3] (+ 1 1)
  [else 3])
== ; by plain arithmetic and “equals for equals”
(cond
  [#true (+ 1 1)]
  [else 3])
== ; by rule condtrue
(+ 1 1)
```

The calculation illustrates plain arithmetic, the replacement of equals by equals, and the two `cond` rules.

Exercise 123. Evaluate the following expressions step by step:

1. $(+ (* (/ 12 8) 2/3) (- 20 (\sqrt{4})))$
2. `(cond`
 $[(= 0 0) \#false]$
 $[(> 0 1) (\text{string=? "a" "a"})]$
 $[\text{else} (= (/ 1 0) 9)])$
3. `(cond`
 $[(= 2 0) \#false]$
 $[(> 2 1) (\text{string=? "a" "a"})]$
 $[\text{else} (= (/ 1 2) 9)])$

Use DrRacket’s stepper to confirm your computations. ▀

Exercise 124. Suppose the program contains these definitions:

```
(define (f x y)
  (+ (* 3 x) (* y y)))
```

Show how DrRacket evaluates the following expressions, step by step:

1. $(+ (f 1 2) (f 2 1))$

```
2. (f 1 (* 2 3))
```

```
3. (f (f 1 (* 2 3)) 19)
```

Use DrRacket's stepper to confirm your computations. ▶

Meaning and Computing

Scientists would say the stepper is a **model** of DrRacket's actual evaluation mechanism.

The interpreters from [Refining Interpreters](#) is another model.

The stepper tool in DrRacket mimics a student in a pre-algebra course. In a sense, the stepper is like a mechanized student from a pre-algebra course, and it is extremely good at, and also extremely fast, applying the laws of arithmetic and algebra as spelled out here.

You can, and you ought to, use the stepper when you don't understand how a new language construct works. The sections on **Computing** suggest exercises for this purpose, but you can make up your own examples, run them through the stepper, and ponder why it takes certain steps.

Finally, you may also wish to use the stepper when you are surprised by the result that a program computes. Using the stepper effectively in this way requires practice. For example, it often means copying the program and pruning unnecessary pieces. But once you understand how to use the stepper well this way, you will find that this procedure clearly explains run-time errors and logical mistakes in your programs.

BSL Errors

For a nearly full sample of error messages, see [BSL Error Messages](#) at the end of this intermezzo.

When DrRacket discovers that some parenthesized phrase does not belong to BSL, it signals a *syntax error*. To determine whether a fully-parenthesized program is syntactically legal, DrRacket uses the grammar in [figure 30](#) and reasons along the lines explained above. Not all syntactically legal programs have meaning, however.

When DrRacket evaluates a syntactically legal program and discovers that some operation is used on the wrong kind of value, it raises a *run-time error*. Consider the simple example of `(/ 1 0)`. It is a syntactically legal sentence, but you know from mathematics that $1/0$ does not have a value. Since BSL's calculations must be consistent with mathematics, DrRacket signals an error:

```
> (/ 1 0)
/: division by zero
```

Naturally it also signals an error when an expression such as `(/ 1 0)` is nested deeply inside of another expression:

```
> (+ (* 20 2) (/ 1 (- 10 10)))
/: division by zero
```

DrRacket's behavior translates into our calculations as follows. When we find an expression that is not a value and when the evaluation rules allow no further simplification, we say the computation is *stuck*. This notion of stuck corresponds to a run-time error. For example, computing the value of the above expression leads to a stuck state:

```
(+ (* 20 2) (/ 1 (- 10 10)))
==
(+ (* 20 2) (/ 1 0))
==
(+ 40 (/ 1 0))
```

What this calculation also shows is that DrRacket eliminates the context of a stuck expression as it signals an error. In this concrete example, it eliminates the addition of 40 to the stuck expression (/ 1 0).

Not all nested stuck expressions end up signaling errors. Suppose a program contains this definition:

```
(define (my-divide n)
  (cond
    [(= n 0) "inf"]
    [else (/ 1 n)]))
```

If you now apply my-divide to 0, DrRacket calculates as follows:

```
(my-divide 0)
==
(cond
  [(= 0 0) "inf"]
  [else (/ 1 0)])
```

It would obviously be wrong to say that the function signals the division-by-zero error now, even though an evaluation of the shaded sub-expression may suggest it. The reason is that (= 0 0) evaluates to #true and therefore the second cond clause does not play any role:

```
(my-divide 0)
==
(cond
  [(= 0 0) "inf"]
  [else (/ 1 0)])
==
(cond
  [#true "inf"]
  [else (/ 1 0)])
==
"inf"
```

Fortunately, our laws of evaluation take care of these situations automatically. We just need to keep in mind when the laws apply. For example, in

```
(+ (* 20 2) (/ 20 2))
```

the addition cannot take place before the multiplication or division. Similarly, the shaded division in

```
(cond
  [(= 0 0) "inf"]
  [else (/ 1 0)])
```

cannot be substituted for the complete `cond` expression until the corresponding line is the first condition in the `cond`.

As a rule of thumb, it is best to keep the following in mind:

Simplify the outermost and left-most sub-expression that is ready for evaluation.

While this guideline is a simplification, it always explains BSL's results.

In some cases, programmers also want to define functions that raise errors. Recall the checked version of `area-of-disk` from [Input Errors](#):

```
(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [else (error "number expected")]))
```

Now imagine applying `checked-area-of-disk` to a string:

```
(- (checked-area-of-disk "a")
  (checked-area-of-disk 10))
==
(- (cond
  [(number? "a") (area-of-disk "a")]
  [else (error "number expected")])
  (checked-area-of-disk 10))
==
(- (cond
  [#false (area-of-disk "a")]
  [else (error "number expected")])
  (checked-area-of-disk 10))
==
(- (error "number expected")
  (checked-area-of-disk 10))
```

At this point you might try to evaluate the second expression, but even if you do find out that its result is roughly [314](#), your calculation must eventually deal with the `error` expression, which is just like a stuck expression. In short, the calculation ends in

```
==  
(error "number expected")
```

Boolean Expressions

Our current definition of BSL omits two forms of expressions: `and` and `or` expressions. Adding them provides a case study of how to study new language constructs. We must first understand their syntax and then their semantics.

Here is the revised grammar of expressions:

```
expr = ...
  | (and expr expr)
```

```
| (or expr expr)
```

The grammar says that `and` and `or` are keywords, each followed by two expressions. At first glance, the two expressions look like function applications.

To understand why `and` and `or` are not BSL-defined functions, we must look at their pragmatics first. Suppose we need to formulate a condition that determines whether $(/ 1 n)$ is `r`:

```
(define (check n r)
  (and (not (= n 0)) (= (/ 1 n) r)))
```

We formulate the condition as an `and` combination of two Boolean expressions because we don't wish to divide by `0` accidentally. Now let us apply `check` to `0` and `1/5`:

```
(check 0 1/5)
==
(and (not (= 0 0)) (= (/ 1 0) 1/5))
```

If `and` were an ordinary operation, we would have to evaluate both sub-expressions, and doing so would trigger an error. Instead `and` is not a primitive operation, and its evaluation throws away the second expression when the first one is `#false`. In short an `and` expression short-circuits evaluation.

Now it would be easy to formulate evaluation rules for `and` and `or`. Another way to explain their meaning is to show how to translate them into equivalent expressions:

```
(and exp-1 exp-2) is short for (cond
  [exp-1 (if exp-2 #true #false)]
  [else #false])
```

and

```
(or exp-1 exp-2) is short for (cond
  [exp-1 #true]
  [else (if exp-2 #true #false)])
```

So if you are ever in doubt about how to evaluate an `and` or `or` expressions, use the above equivalences to calculate. But we trust that you understand these operations intuitively, and that is almost always enough.

Note You may be wondering why the above abbreviations use

```
(if exp-2 #true #false)
```

on the right-hand side instead of just

```
exp-2
```

And indeed, if `exp-2` evaluates to `#true` or `#false`, writing `exp-2` suffices and is shorter than the `if` expression. An expression in BSL may evaluate to all kinds of values, however. If it does evaluate to something other than a Boolean, `(if exp-2 #true #false)` would signal an error while `exp-2` would not. Since `(and exp-1 exp-2)` also signals an error when `exp-2` evaluates to something other than a Boolean, we use the above right-hand side and not the inaccurate short-cut.

Exercise 125. The use of `if` may have surprised you in another way because this intermezzo does not mention this form elsewhere. In short, the intermezzo appears to

explain `and` with a form that has no explanation either. At this point, we are relying on your intuitive understanding of `if` as a short-hand for `cond`. Write down a rule that shows how to reformulate

```
| (if exp-test exp-then exp-else)
```

as a `cond` expression.

Constant Definitions

Programs consist not only of function definitions but also constant definitions, but these weren't included in our first grammar. So here is an extended grammar that includes constant definitions:

```
definition = ...
| (define name expr)
```

The shape of a constant definition is similar to that of a function definition. It starts with a “(”, followed by the keyword `define`, followed by a variable, followed by an expression, and closed by a right parenthesis “)” that matches the very first one. While the keyword `define` distinguishes constant definitions from expressions, it does not differentiate from function definitions. For that, a human reader must look at the second component of the definition.

Next we must understand what a constant definition means. For a constant definition with a literal constant on the right hand side, such as

As it turns out DrRacket has another way of dealing with function definitions; see [Nameless Functions](#).

```
| (define RADIUS 5)
```

it is easy to see that the variable is just a short hand for the value. That is, wherever DrRacket encounters `RADIUS` during an evaluation, it replaces it with `5`.

For a constant definition with a proper expression on the right-hand side, say,

```
| (define DIAMETER (* 2 RADIUS))
```

we must immediately determine the value of the expression. This process makes use whatever definitions precede this constant definition. Hence,

```
| (define RADIUS 5)
| (define DIAMETER (* 2 RADIUS))
```

is equivalent to

```
| (define RADIUS 5)
| (define DIAMETER 10)
```

This recipe even works when function definitions are involved:

```
| (define RADIUS 10)
| (define DIAMETER (* 2 RADIUS))
| (define (area r) (* 3.14 (* r r)))
```

```
(define AREA-OF-RADIUS (area RADIUS))
```

As DrRacket steps through this sequence of definitions, it first determines that RADIUS stands for 10, DIAMETER for 20, and area is the name of a function. Finally, it evaluates (area RADIUS) to 314 and associates AREA-OF-RADIUS with that value.

Mixing constant and function definitions gives rise to a new kind of run-time error, too. Take a look at this program:

```
(define RADIUS 10)

(define DIAMETER (* 2 RADIUS))

(define AREA-OF-RADIUS (area RADIUS))

(define (area r) (* 3.14 (* r r)))
```

It is like the one above with the last two definitions swapped. For the first two definitions, evaluation proceeds as before. For the third one, however, evaluation goes wrong. The recipe calls for the evaluation of (area RADIUS). While the definition of RADIUS precedes this expression, the definition of area has not yet been encountered. If you were to evaluate the program with DrRacket, you would therefore get an error, explaining that ``this function is not defined.'' So be carefully about using functions in constant definitions only when you know they are defined.

Exercise 126. Evaluate the following program, step by step:

```
(define PRICE 5)
(define SALES-TAX (* 0.08 PRICE))
(define TOTAL (+ PRICE SALES-TAX))
```

Does the evaluation of the following program signal an error?

```
(define COLD-F 32)
(define COLD-C (fahrenheit->celsius COLD-F))
(define (fahrenheit->celsius f)
  (* 5/9 (- f 32)))
```

How about the next one?

```
(define LEFT -100)
(define RIGHT 100)
(define (f x) (+ (* 5 (expt x 2)) 10))
(define f@LEFT (f LEFT))
(define f@RIGHT (f RIGHT))
```

Check your computations with DrRacket's stepper. ■

Structure Type Definitions

As you can imagine, **define-struct** is by far the most complex BSL construct. We have therefore saved its explanation for last. Here is the addition to the grammar:

```
definition = ...
| (define-struct name [name ...])
```

A structure type definition is a third form of definition. The keyword `define-struct` distinguishes this form of definition from function and constant definitions.

Here is a simple example:

```
| (define-struct point [x y z])
```

Since `point`, `x`, `y`, and `z` are variables and the parentheses are placed according to the grammatical pattern, it is a proper definition of a structure type. In contrast, these two parenthesized sentences

```
| (define-struct [point x y z])
```

```
| (define-struct point x y z)
```

are illegal definitions, because `define-struct` is not followed by a single variable name and a sequence of variables in parentheses.

While the syntax of `define-struct` is straightforward, its meaning is difficult to spell out with evaluation rules. As mentioned several times, a `define-struct` definition defines several functions at once: a constructor, several selectors, and a predicate. Thus the evaluation of

```
| (define-struct c [s-1 ... s-n])
```

introduces into the following functions into the program:

1. `make-c`: a *constructor*;
2. `c-s-1`... `c-s-n`: a series of *selectors*; and
3. `c?:` a *predicate*.

These functions have the same status as `+`, `-`, or `*`. Before we can understand the rules that govern these new functions, however, we must return to the definition of values. After all, one purpose of a `define-struct` is to introduce a class of values that is distinct from all existing values.

Simply put, the use of `define-struct` extends the universe of values. To start with, it now also contains structures, which compound several values into one. When a program contains a `define-struct` definition, its evaluation modifies the definition of values:

A *value* is one of: a *number*, a *boolean*, a *string*, an *image*,

- or a structure value:

```
| (make-c _value-1 ... _value-n)
```

assuming a structure type `c` is defined.

For example, the definition of `point` means the addition of values of this shape:

```
| (make-point 1 2 -1)
| (make-point "one" "hello" "world")
| (make-point 1 "one" (make-point 1 2 -1))
|
| ...
```

Now we are in a position to understand the evaluation rules of the new functions. If `c-s-1` is applied to a `c` structure, it returns the first component of the value. Similarly, the second

selector extracts the second component, the third selector the third component, and so on. The relationship between the new data constructor and the selectors is best characterized with n equations added to BSL's rules:

$$\begin{aligned} (\text{c-s-1 } (\text{make-c } V-1 \dots V-n)) &== V-1 \\ (\text{c-s-n } (\text{make-c } V-1 \dots V-n)) &== V-n \end{aligned}$$

For our running example, we get the specific equations

$$\begin{aligned} (\text{point-x } (\text{make-point } V \ U \ W)) &== V \\ (\text{point-y } (\text{make-point } V \ U \ W)) &== U \\ (\text{point-z } (\text{make-point } V \ U \ W)) &== W \end{aligned}$$

When DrRacket encounters `(point-y (make-point 3 4 5))` in a calculation, it equates it with `4` and `(point-x (make-point (make-point 1 2 3) 4 5))` to `(make-point 1 2 3)`.

The predicate `c?` can be applied to any value. It returns `#true` if the value is of kind `c` and `#false` otherwise. We can translate both parts into two equations:

$$\begin{aligned} (\text{c? } (\text{make-c } V-1 \dots V-n)) &== \text{\#true} \\ (\text{c? } V) &== \text{\#false} \end{aligned}$$

if `V` is a value not constructed with `make-c`. Again, the equations are best understood in terms of our example:

$$\begin{aligned} (\text{point? } (\text{make-point } U \ V \ W)) &== \text{\#true} \\ (\text{point? } X) &== \text{\#false} \end{aligned}$$

if `X` is a value but not a point structure. Thus, `(point? (make-point 3 4 5))` is `#true` and `(point? 3)` is `#false`.

Exercise 127. Pick the legal from the illegal sentences in the following list:

1. `(define-struct oops [])`
2. `(define-struct child [parents dob date])`
3. `(define-struct (child person) [dob date])`

Explain why the sentences are legal or illegal. ▀

Exercise 128. Identify the *values* among the following lists of expressions, assuming the definitions area contains these structure type definitions:

```
..... (define-struct point [x y z])
..... (define-struct none [])
```

1. `(make-point 1 2 3)`
2. `(make-point (make-point 1 2 3) 4 5)`
3. `(make-point (+ 1 2) 3 4)`
4. `(make-none)`
5. `(make-point (point-x (make-point 1 2 3)) 4 5)`

Explain why the expressions are values or not. ▀

Exercise 129. Suppose the program contains

```
| (define-struct ball [x y speed-x speed-y])
```

Predict the results of evaluating the following expression:

1. (number? (make-ball 1 2 3 4))
2. (ball-speed-y (make-ball (+ 1 2) (+ 3 3) 2 3))
3. (ball-y (make-ball (+ 1 2) (+ 3 3) 2 3))
4. (ball-x (make-posn 1 2))
5. (ball-speed-y 5)

Check your predictions in DrRacket's interactions area. Use the stepper to see how DrRacket produces this result. ■

BSL Tests

Figure 32 collects all the elements of BSL that this intermezzo explains plus several testing forms: `check-expect`, `check-within`, `check-member-of`, `check-range`, `check-error`, `check-random`, and `check-satisfied`. The actual grammar of BSL is even a bit larger; see the Help Desk for details.

```
program = def-or-expr ...

def-or-expr = definition
            | expr
            | test-case

definition = (define (name variable variable ...) expr)
            | (define name expr)
            | (define-struct name [name ...])

expr = (name expr expr ...)
      | (cond [expr expr] ... [expr expr])
      | (cond [expr expr] ... [else expr])
      | (and expr expr expr ...)
      | (or expr expr expr ...)
      | name
      | number
      | string
      | image

test-case = (check-expect expr expr)
           | (check-within expr expr expr)
           | (check-member-of expr expr ...)
           | (check-range expr expr expr)
           | (check-error expr)
           | (check-random expr expr)
           | (check-satisfied expr name)
```

Figure 32: BSL, Full Grammar

The general meaning of testing expressions is easy to explain. When you click the button{run} button, DrRacket collects all testing expressions and moves them from the middle of the program to the end, retaining the order in which they appear. It then evaluates the definitions, expressions, and tests and gets the interactions area ready.

Each test evaluates its pieces and then compares them with the expected outcome via some predicate. Beyond that, tests communicate with DrRacket to collect some success and failure statistics and information on how to display test failures.

For details, read the documentation on these test forms. Here are some illustrative examples:

```
; check-expect compares the outcome and the expected value with equal?
(check-expect 3 3)

; check-member-of compares the outcome and the expected values with equal?
; if one of them yields #true, the test succeeds
(check-member-of "green" "red" "yellow" "green")

; check-within compares the outcome and the expected value with a predicate
; like equal? but allows for a tolerance of epsilon for each inexact number
(check-within (make-posn #i1.0 #i1.1) (make-posn #i0.9 #i1.2) 0.2)

; check-range is like check-within
; but allows for a specification of an interval
(check-range 0.9 #i0.6 #i1.0)

; check-error checks whether an expression signals (any) error
(check-error (/ 1 0))

; check-random evaluates the sequences of calls to random in the
; two expressions such that they yield the same number
(check-random (make-posn (random 3) (random 9))
              (make-posn (random 3) (random 9)))

; check-satisfied determines whether a predicate produces #true
; when applied to the outcome, that is, whether outcome has a certain property
(check-satisfied 4 even?)
```

All of the above tests succeed. The remainig parts of the book re-introduce these test forms as needed.

Exercise 130. Copy and paste the following tests into DrRacket's definitions area:

```
(check-expect 3 4)
(check-member-of "green" "red" "yellow" "grey")
(check-within (make-posn #i1.0 #i1.1) (make-posn #i0.9 #i1.2) 0.01)
(check-range #i0.9 #i0.6 #i0.8)
(check-error (/ 1 1))
(check-random (make-posn (random 3) (random 9))
              (make-posn (random 9) (random 3)))
(check-satisfied 4 odd?)
```

Validate that all of them fail and explain why. ■

BSL Error Messages

The figures in this section sample the error messages that a BSL program may signal. Each figure first mentions whether it assumes anything about the definitions area. The remaining entries consist of three parts:

- the top-left cell shows code fragments that signal an error message;
- the bottom-left one is the error message;
- the cell on the right explains the error and occasionally suggests corrections.

When an error shows up and you need help, find the appropriate figure, search the bottom-left cells for a match, and then study the complete entry.

To illustrate how these tables work, consider the following example, which is the worst possible error message you may ever see:

```
(define (absolute n)
  (cond
    [< 0 (- n)]
    [else n]))
<: expected a function call, but
there is no open parenthesis
before this function
```

A `cond` expression consists of the keyword followed by an arbitrarily long sequence of `cond` clauses. In turn, every clause consists of two parts: a condition and an answer, both of which are expressions. In this definition of `absolute`, the first clause starts with `<`, which is supposed to be a condition but isn't even an expression according to our definition. This confuses the BSL implementation so much that it doesn't even ``see'' the open parenthesis to the left of `<`. Here the fix is to use `(< n 0)` as the condition.

The red highlighting of `<` in the function definition points to the error. Below the definition, you can see the error message that DrRacket presents in the interactions window if you click *RUN*. Study the explanation of the error on the right to understand how to address this somewhat self-contradictory message. And now rest assured that no other error message is even remotely as opaque as this one.

In DrRacket's definitions area:

```
; Number Number -> Number
; find the average of x and y
(define (average x y)
  (/ (+ x y)
      2))
```

```
(f 1)
f: this function is not defined
```

The application names `f` as the function, and `f` is not defined in the definitions area. Define the function, or make sure that the variable name is spelled correctly.

```
(1 3 "three" #true)
function call: expected a
function after the open
parenthesis, but found a number
```

An open parenthesis is always followed by the name of a function, and **1** is not a name. The function name is either supplied by BSL, say **+**, or it is defined in the definitions area, say **average**.

```
(average 7)
average: expects 2 arguments,
but found only 1
```

This function call applies **average** to one argument, **7**, even though its definition calls for two numbers.

```
(average 1 2 3)
average: expects 2 arguments,
but found 3
```

Here **average** is applied to **three** numbers instead of two.

```
(make-posn 1)
make-posn: expects 2 arguments,
but found only 1
```

Functions defined by BSL must also be applied to the correct number of arguments. For example, **make-posn** must be applied to two arguments.

Figure 33: Error messages about function applications in BSL

In DrRacket's definitions area:

```
; Number Number -> Number
; find the average of x and y
(define (average x y)
  (/ (+ x y)
      2))
```

```
(posn-x #true)
posn-x: expects a posn, given
#true
```

A function must be applied to the arguments it expects. For example, **posn-x** expects an instance of **posn**.

```
(average "one" "two")
+: expects a number as 1st
argument, given "one"
```

A function defined to consume two **Numbers** must be applied to two **Numbers**; here **average** is applied to **Strings**. The error message is triggered only when **average** applies **+** to these **Strings**. Like all primitive operations, **+** is a checked function.

Figure 34: Error messages about wrong data in BSL

In DrRacket's definitions area:

```
; N in [0,1,...10]
(define 0-to-9 (random 10))
```

```
(cond
```

Every **cond** clause must consist of

```
[(>= 0-to-9 5)])  
cond: expected a clause with a  
question and an answer, but  
found a clause with only one part
```

exactly two parts: a condition and an answer. Here `(>= 0-to-9 5)` is apparently intended as the condition; the answer is missing.

```
(cond  
[(>= 0-to-9 5)  
"head"  
"tail"]])  
cond: expected a clause with a  
question and an answer, but  
found a clause with 3 parts
```

In this case, the `cond` clause consists of three parts, which also violates the grammar. While `(>= 0-to-9 5)` is clearly intended to be the condition, the clause comes with two answers: `"head"` and `"tail"`. Pick one or create a single value from the two strings.

```
(cond)  
cond: expected a clause after  
cond, but nothing's there
```

A conditional must come with at least one `cond` clause and usually it comes with at least two.

Figure 35: Error messages about conditionals in BSL

```
(define f(x) x)  
define: expected only one  
expression after the variable  
name f, but found 1 extra part
```

A definition consist of three parts: the `define` keyword, a sequence of variable names enclosed in parentheses, and an expression. This definition consists of four parts; this definition is an attempt to use the standard notation from algebra courses for the header `f (x)` instead of `(f x)`.

```
(define (f x x) x)  
define: found a variable that is  
used more than once: x
```

The sequence of parameters in a function definition must not contain duplicate variables.

```
(define (g) x)  
define: expected at least one  
variable after the function  
name, but found none
```

In BSL a function header must contain at least two names. The first one is the name of the function; the remaining variable names are the parameters, and they are missing here.

```
(define (f x x) x)  
define: found a variable that is  
used more than once: x
```

The sequence of parameters in a function definition must not contain duplicate variable names.

```
(define (f (x)) x)  
define: expected a variable, but  
found a part
```

Here the function header contains `(x)`, which is **not** a variable name.

```
(define (average x y) x y)
define: expected only one
expression for the function body,
but found 1 extra part
```

This function definition comes with two expressions following the header: x and y.

Figure 36: Error messages about function definitions in BSL

```
(define-struct [x])
(define-struct [x y z])
define-struct: expected the
structure name after define-
struct, but found a part
```

A structure type definition consists of three parts: the `define-struct` keyword, the structure's name, and a sequence of names enclosed in parentheses. Here the structure's name is missing.

```
(define-struct x [y y])
define-struct: found a field
name that is used more than
once: y
```

The sequence of field names in a structure type definition must not contain duplicate names.

```
(define-struct x y)
(define-struct x y z)
define-struct: expected at least
one field name (in parentheses)
after the structure name, but
found something else
```

These structure type definitions lack the sequence of field names enclosed in parentheses, which must follow the name of the structure.

```
(define-struct x [(y) z])
define-struct: expected a field
name, but found a part
```

Here the sequence of field names contains `(z)`, which is **not** a name.

Figure 37: Error messages about structure type definitions in BSL

v.6.3.0.2

II Arbitrarily Large Data

Every data definition in [Fixed-Size Data](#) describes data of a fixed size. To us, boolean values, numbers, strings, and images are atomic; computer scientists say they have a size of one unit. With a structure, you compose a fixed number of pieces of data. Even if you use the language of data definitions to create deeply nested structures, you always know the exact number of atomic pieces of data in any specific instance. Many programming problems, however, deal with an undetermined number of pieces of information that must be processed as one piece of data. For example, one program may have to compute the average of a bunch of numbers and another may have to keep track of an arbitrary number of objects in an interactive game. Regardless, it is impossible with your knowledge to formulate a data definition that can represent this kind of information as data.

This part revises the language of data definitions so that it becomes possible to describe data of (finite but) arbitrary size. For a concrete illustration, the first half of this part deals with lists, a form of data that appears in most modern programming languages. In parallel to the extended language of data definitions, this part also revises the design recipe to cope with such data definitions. The latter chapters demonstrate how these data definitions and the revised design recipe work in a variety of contexts.

9 Lists

This chapter introduces self-referential data definitions, that is, definitions that refer to themselves. It is highly unlikely that you have encountered such definitions before. Your English teachers certainly stay away from these, and many mathematics courses are somewhat vague when it comes to such definitions. Programmers cannot afford to be vague; programming languages call for precise definitions.

Lists are one ubiquitous instance of self-referential data definitions. With lists, our programming examples become interesting, which is why this chapter introduces them and shows some ways of manipulating them. It thus motivates the revision of the design recipe in the next chapter, which explains how to systematically create functions that deal with self-referential data definitions.

9.1 Creating Lists

All of us make lists all the time. Before we go grocery shopping, we write down a list of items we wish to purchase. Some people write down a to-do list every morning. During December, many children prepare Christmas wish lists. To plan a party, we make a list of invitees. Arranging information in the form of lists is a ubiquitous part of our life.

Given that information comes in the shape of lists, we must clearly learn how to represent such lists as BSL data. Indeed, because lists are so important BSL comes with built-in support for creating and manipulating lists, similar to the support for Cartesian points (`posn`). In contrast to points, the data definition for lists is always left to you. But first things first. We start with the creation of lists.

When we form a list, we always start out with the empty list. In BSL, we represent the

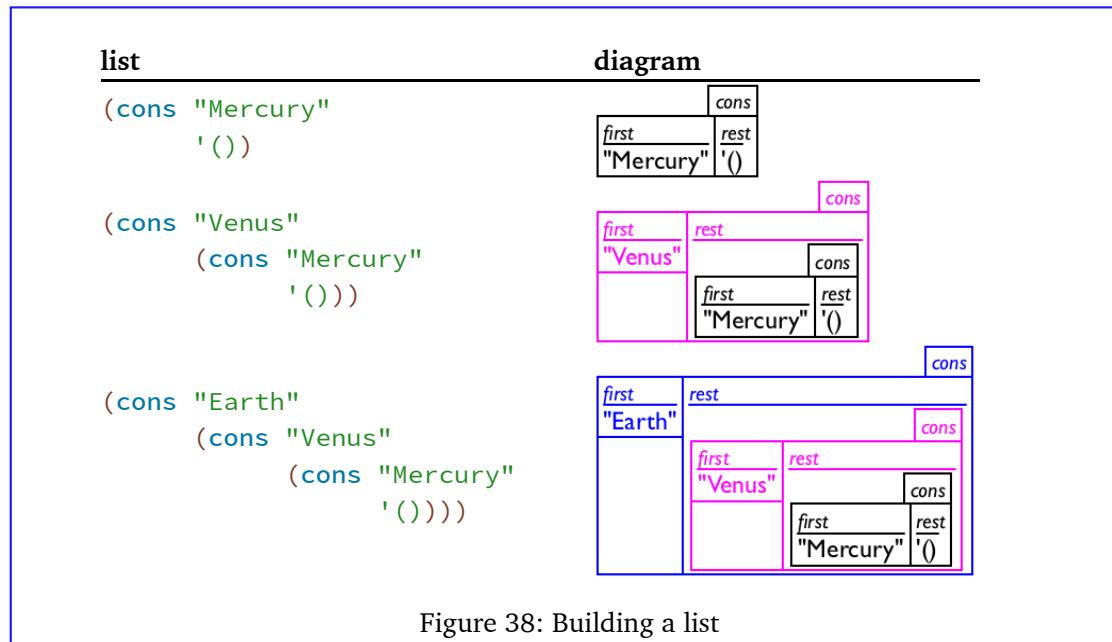
empty list with

```
| '()
```

which is pronounced “empty,” short for “empty list.” Like `#true` or `5`, `'()` is just a constant. When we add something to a list, we construct another list; in BSL, the `cons` operation serves this purpose. For example,

```
| (cons "Mercury" '())
```

constructs a list from the `'()` list and the string `"Mercury"`. Figure 38 presents this list in the same pictorial manner we used for structures. The box for `cons` has two fields: `first` and `rest`. In this specific example the `first` field contains `"Mercury"` and the `rest` field contains `'()`.



Once we have a list with one item in it, we can construct lists with two items by using `cons` again. Here is one:

```
| (cons "Venus" (cons "Mercury" '()))
```

And here is another:

```
| (cons "Earth" (cons "Mercury" '()))
```

The middle row of figure 38 shows how you can imagine lists that contain two items. It is also a box of two fields, but this time the `rest` field contains a box. Indeed, it contains the box from the top row of the same figure.

Finally, we construct a list with three items:

```
| (cons "Earth" (cons "Venus" (cons "Mercury" '()))))
```

The last row of figure 38 illustrates the list with three items. Its `rest` field contains a box that contains a box again. So, as we create lists we put boxes into boxes into boxes, etc. While this may appear strange at first glance, it is just like a set of Chinese gift boxes or a set of nested drinking cups, which we sometimes get for birthdays. The only difference is that BSL programs can nest lists much deeper than any artist could nest physical boxes.

```
(cons "Earth"
  (cons "Venus"
    (cons "Mercury"
      '())))
```

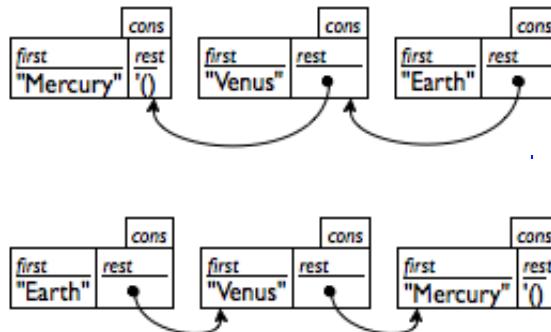


Figure 39: Drawing a list

Because even good artists would have problems with drawing deeply nested structures, computer scientists resort to box-and-arrow diagrams instead. Figure 39 illustrates how to re-arrange the last row of figure 38. Each `cons` structure becomes a separate box. If the rest field is too complex to be drawn inside of the box, we draw a bullet instead and a line with an arrow to the box that it contains. Depending on how the boxes are arranged you get two kinds of diagrams. The first, displayed in the top row of figure 39, lists the boxes in the order in which they are created. The second, displayed in the bottom row, lists the boxes in the order in which they are `consed` together. Hence the second diagram immediately tells you what `first` would produce when applied to the list, no matter how long the list is. For this reason, people tend to prefer the second arrangement.

Exercise 131. Create BSL lists that represent

1. a list of celestial bodies, say, at least all the planets in our solar system;
2. a list of items for a meal, for example, steak, French fries, beans, bread, water, brie cheese, and ice cream; and
3. a list of colors.

Sketch box representations of these lists, similar to those in figure 38 and figure 39. Which of the sketches do you like better? ■

You can also make lists of numbers. As before, `'()` is the list without any items. Here is a list with the 10 digits:

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

To build this list requires 10 list constructions and one `'()`. For a list of three arbitrary numbers, for example,

```
(cons pi
  (cons e
    (cons -22.3 '()))))
```

we need three `conses`.

In general a list does not have to contain values of one kind, but may contain arbitrary values:

```
(cons "Robbie Round"
  (cons 3
    (cons #true
      '())))
```

The first item of this list is a string, the second one is a number, and the last one a Boolean. You may consider this list as the representation of a personnel record with three pieces of data: the name of the employee, the number of years spent at the company, and whether the employee has health insurance through the company. Or, you could think of it as representing a virtual player in some game. Without a data definition, you just can't know what data is all about.

Then again, if this list is supposed to represent a record with a fixed number of pieces, use a structure type instead.

Here is a first data definition that involves `cons`:

```
; A 3LON is (cons Number (cons Number (cons Number '())))
; interpretation (cons 1 (cons -1 (cons -2 '()))) represents a point
; in a three-dimensional space
```

Of course, this data definition uses `cons` like others use constructors for structure instances, and in a sense, `cons` is just a special constructor. What this data definition fails to demonstrate is how to form lists of arbitrary length: lists that contain nothing, one item, two items, ten items, or perhaps 1,438,901 items.

So let's try again:

```
; A List-of-names is one of:
; - '()
; - (cons String List-of-names)
; interpretation a List-of-names represents a list of invitees by last name
```

Stop. Take a deep breath, and read it again. This data definition is one of the most unusual definitions you have ever encountered—where else have you seen a definition that refers to itself?—and it isn't even clear whether it makes sense. After all, if you had told your English teacher that “a table is a table” defines the word “table,” the most likely response would have been “Nonsense!” because a self-referential definition doesn't explain what a word means.

In computer science and in programming, though, self-referential definitions play a central role, and with some care, such definitions actually do make sense. Here “making sense” means that we can use the data definition for what it is intended for, namely, to generate examples of data that belong to the class that is being defined or to check whether some given piece of data belongs to the defined class of data. From this perspective, the definition of `List-of-names` makes complete sense. At a minimum, we can generate the `'()` list as one example, using the first clause in the itemization. Given `'()` as an element of `List-of-names`, it is also easy to make a second example:

```
(cons "Findler" '())
```

Here we are using a `String` and the only list from `List-of-names` to generate a piece of data

according to the second clause in the itemization. With the same rule, we can generate many more lists of this kind:

```
(cons "Flatt" '())
(cons "Felleisen" '())
(cons "Krishnamurthi" '())
```

And while these lists all contain one name (represented as a `String`), it is actually possible to use the second line of the data definition to create lists with more names in them:

```
(cons "Felleisen" (cons "Findler" '()))
```

This piece of data belongs to `List-of-names` because "`Felleisen`" is a `String` and `(cons "Findler" '())` is a `List-of-names` according to our argument above.

Exercise 132. Create an element of `List-of-names` that contains five `Strings`. Sketch a box representation of the list similar to those found in [figure 38](#).

Explain why

```
(cons "1" (cons "2" '()))
```

is an element of `List-of-names` and why `(cons 2 '())` isn't. ■

Now that we know why our first self-referential data definition makes sense, let us take another look at the definition itself. Looking back we see that all the original examples in this section fall into one of two categories: start with an '`()`' list or use `cons` to add something to an existing list. The trick then is to say what kind of “existing lists” you wish to allow, and a self-referential definition ensures that you are not restricted to lists of some fixed length.

Exercise 133. Provide a data definition for representing lists of `Boolean` values. The class contains all arbitrarily long lists of `Booleans`. ■

9.2 What Is '`()`', What Is `cons`

Let us step back for a moment and take a close look at '`()`' and `cons`. As mentioned, '`()`' is just a constant. When compared to constants such as `5` or "`this is a string`", it looks more like a function name or a variable; but when compared with `#true` and `#false`, it should be easy to see that it really is just BSL's representation for empty lists.

As for our evaluation rules, '`()`' is a new kind of atomic value, distinct from any other kind: numbers, Booleans, strings, and so on. It also isn't a compound value, like `Posns`. Indeed, '`()`' is so unique, it belongs into a class of values all by itself. As such, this class of values comes with a predicate that recognizes only '`()`' and nothing else:

```
; Any -> Boolean
; is the given value '()
(define (empty? x) ...)
```

Like all class predicates, `empty?` can be applied to any value from the universe of BSL values. It produces `#true`, however, only if it is applied to '`()`'. Thus,

```
> (empty? '())
#true
> (empty? 5)
```

```
#false
> (empty? "hello world")
#false
> (empty? (cons 1 '()))
#false
> (empty? (make-posn 0 0))
#false
```

Next we turn to `cons`. Everything we have seen so far suggests that `cons` is a constructor just like those introduced by structure type definitions. More precisely, `cons` appears to be the constructor for a two-field structure: the first one for any kind of value and the second one for an list-like value. The following definitions translate this idea into BSL:

```
(define-struct pair [left right])
; A ConsPair is (make-pair Any Any).

; Any Any -> ConsPair
(define (our-cons a-value a-list)
  (make-pair a-value a-list))
```

The only catch is that `our-cons` accepts all possible BSL values for its second argument and `cons` doesn't, as the following experiment validates:

```
> (cons 1 2)
cons: second argument must be a list, but received 1 and 2
```

Put differently, `cons` is really a checked function, the kind discussed in [Itemizations and Structures](#), which suggests the following refinement:

```
; A ConsOrEmpty is one of:
; - '()
; - (cons Any ConsOrEmpty)
; interpretation ConsOrEmpty is the class of all BSL lists

; Any ConsOrEmpty -> ConsOrEmpty
(define (our-cons a-value a-list)
  (cond
    [(empty? a-list) (make-pair a-value a-list)]
    [(pair? a-list) (make-pair a-value a-list)]
    [else (error "cons: list as second argument expected")]))
```

If `cons` is a checked constructor function, you may be wondering how to extract the pieces from the resulting structure. After all, [Adding Structure](#) says that programming with structures requires selectors. Since a `cons` structure has two fields, there are two selectors: `first` and `rest`. They are also easily defined in terms of our `pair` structure:

```
; ConsOrEmpty -> Any
; extracts the left part of the given pair
(define (our-first a-list)
  (if (empty? a-list)
      (error 'our-first "..."))
      (pair-left a-list)))
```

Stop! Define `our-rest`.

If your program can access the structure type definition for `pair`, it is easy to create pairs that don't contain `'()` or another pair in the `right` field. Whether such bad instances are

created intentionally or accidentally, they tend to break functions and programs in strange ways. BSL therefore hides the actual structure type definition for `cons` to avoid these problems. [Local Function Definitions](#) demonstrates one way how your programs can hide such definitions, too, but for now, you don't need this power.

<code>'()</code>	a special value, mostly to represent the empty list
<code>empty?</code>	a predicate to recognize <code>'()</code> and nothing else
<code>cons</code>	a checked constructor to create two-field instances
<code>first</code>	the selector to extract the last item added
<code>rest</code>	the selector to extract the extended list
<code>cons?</code>	a predicate to recognizes instances of <code>cons</code>

Figure 40: List primitives

Figure 40 summarizes this section. The key insight is that `'()` is a unique value and that `cons` is a checked constructor that produces list values. Furthermore, `first`, `rest`, and `cons?` are merely distinct names for the usual predicate and selectors. What this chapter teaches then is **not** a new way of creating data but **a new way of formulating data definitions**.

9.3 Programming with Lists

Here we use the word “friend” in the sense of social networks, not the real world.

Say you’re keeping track of your friends with some list, and say the list has grown so large that you need a program to determine whether some specific name is on the list. To make this idea concrete, let us state it as an exercise:

Sample Problem: You are working on the contact list for some new cell phone. The phone’s owner updates—adds and deletes names—and consults this list—looks for specific names—on various occasions. For now, you are assigned the task of designing a function that consumes this list of contacts and determines whether it contains the name “Flatt.”

We will solve this problem here, and once we have a solution to this sample problem, we will generalize it to a function that finds any name on a list.

The data definition for [List-of-names](#) from the preceding is appropriate for representing the list of names that the function is to search. Next we turn to the header material:

```
; List-of-names -> Boolean
; determines whether "Flatt" occurs on a-list-of-names
(define (contains-flatt? a-list-of-names)
  #false)
```

Following the general design recipe, we next make up some examples that illustrate the purpose of `contains-flatt?`. First, we determine the output for the simplest input: `'()`. Since this list does not contain any strings, it certainly does not contain “Flatt”, and the expected answer is `#false`:

```
(check-expect (contains-flatt? '()) #false)
```

Then we consider lists with a single item. Here are two examples:

```
(check-expect (contains-flatt? (cons "Findler" '())) #false)
(check-expect (contains-flatt? (cons "Flatt" '())) #true)
```

In the first case, the answer is `#false` because the single item on the list is not `"Flatt"`; in the second case, the item is `"Flatt"`, and the answer is `#true`. Finally, here are two more general examples, with lists of several items:

```
(check-expect
  (contains-flatt? (cons "Mur" (cons "Fish" (cons "Find" '()))))
  #false)
(check-expect
  (contains-flatt? (cons "A" (cons "Flatt" (cons "C" '()))))
  #true)
```

Again, the answer in the first case must be `#false` because the list does not contain `"Flatt"`, and in the second case it must be `#true` because `"Flatt"` is one of the items on the list provided to the function.

Take a breath. Run the program. The header is a “dummy” definition for the function; you have some examples; they have been turned into tests; and best of all, some of them actually succeed. They succeed for the wrong reason but succeed they do. If things make sense now, read on.

The fourth step is to design a function template that matches the data definition. Since the data definition for lists of strings has two clauses, the function’s body must be a `cond` expression with two clauses. The two conditions—`(empty? a-list-of-names)` and `(cons? a-list-of-names)`—determine which of the two kinds of lists the function received:

```
(define (contains-flatt? a-list-of-names)
  (cond
    [(empty? a-list-of-names) ...]
    [(cons? a-list-of-names) ...]))
```

Instead of `(cons? a-list-of-names)`, we could use `else` in the second clause.

We can add one more hint to the template by studying each clause of the `cond` expression in turn. Specifically, recall that the design recipe suggests annotating each clause with selector expressions if the corresponding class of inputs consists of compounds. In our case, we know that `'()` does not have compounds, so there are no components. Otherwise the list is constructed from a string and another list of strings, and we remind ourselves of this fact by adding `(first a-list-of-names)` and `(rest a-list-of-names)` to the template:

```
(define (contains-flatt? a-list-of-names)
  (cond
    [(empty? a-list-of-names) ...]
    [(cons? a-list-of-names)
     (... (first a-list-of-names) ... (rest a-list-of-names) ...))])
```

Now it is time to switch to the programming task proper, the fifth step of our design recipe. It starts from template and deals with each `cond`-clause separately. If `(empty? a-list-of-names)` is true, the input is the empty list, in which case the function must produce the result `#false`. In the second case, `(cons? a-list-of-names)` is true. The

annotations in the template remind us that there is a first string and the rest of the list. So let us consider an example that falls into this category:

```
(cons "A"
  (cons ...
    ... '()))
```

The function, just like a human being, must clearly compare the first item with "**Flatt**". In this example, the first string is "**A**" and not "**Flatt**", so the comparison yields **#false**. If we had considered some other example instead, say,

```
(cons "Flatt"
  (cons ...
    ... '()))
```

the function would determine that the first item on the input is "**Flatt**", and would therefore respond with **#true**. This implies that the second line in the **cond** expression should contain an expression that compares the first name on the list with "**Flatt**":

```
(define (contains-flatt? a-list-of-names)
  (cond
    [(empty? a-list-of-names) #false]
    [(cons? a-list-of-names)
      (... (string=? (first a-list-of-names) "Flatt") ...
        ... (rest a-list-of-names) ...))])
```

Furthermore, if the comparison of **(first a-list-of-names)** yields **#true**, the function is done and produces **#true**. If the comparison yields **#false**, we are left with another list of strings: **(rest a-list-of-names)**. Clearly, we can't know the final answer in this case, because depending on what "... represents, the function must produce **#true** or **#false**. Put differently, if the first item is not "**Flatt**", we need some way to check whether the rest of the list contains "**Flatt**".

Fortunately, we have just such a function: **contains-flatt?**, which according to its purpose statement determines whether a list contains "**Flatt**". The purpose statement implies that if **l** is a list of strings, **(contains-flatt? l)** tells us whether **l** contains the string "**Flatt**". Similarly, **(contains-flatt? (rest l))** determines whether the rest of **l** contains "**Flatt**". And in the same vein, **(contains-flatt? (rest a-list-of-names))** determines whether or not "**Flatt**" is in **(rest a-list-of-names)**, which is precisely what we need to know now.

In short, the last line of the function should be **(contains-flatt? (rest a-list-of-names))**:

```
; List-of-names -> Boolean
(define (contains-flatt? a-list-of-names)
  (cond
    [(empty? a-list-of-names) #false]
    [(cons? a-list-of-names)
      (... (string=? (first a-list-of-names) "Flatt") ...
        ... (contains-flatt? (rest a-list-of-names)) ...))])
```

The trick is now to combine the values of the two expressions in the appropriate manner. As mentioned, if the first expression yields **#true**, there is no need to search the rest of the list; if it is **#false**, though, the second expression may still yield **#true**, meaning the name "**Flatt**" is on the rest of the list. All of this suggests that the result of **(contains-flatt?**

`a-list-of-names`) is `#true` if either the first expression in the last line **or** the second expression yields `#true`.

```
; List-of-names -> Boolean
; determines whether "Flatt" occurs on a-list-of-names

(check-expect
  (contains-flatt? (cons "Mur" (cons "Fish" (cons "Find" '()))))
  #false)
(check-expect
  (contains-flatt? (cons "A" (cons "Flatt" (cons "C" '()))))
  #true)

(define (contains-flatt? a-list-of-names)
  (cond
    [(empty? a-list-of-names) #false]
    [(cons? a-list-of-names)
     (or (string=? (first a-list-of-names) "Flatt")
         (contains-flatt? (rest a-list-of-names)))]))
```

Figure 41: Searching a list

Here then is the complete definition: [figure 41](#). Overall it doesn't look too different from the definitions in the first chapter of the book. It consists of a signature, a purpose statement, two examples, and a definition. The only way in which this function definition differs from anything you have seen before is the self-reference, that is, the reference to `contains-flatt?` in the body of the `define`. Then again, the data definition is self-referential, too, so in some sense this second self-reference shouldn't be too surprising.

Exercise 134. Use DrRacket to run `contains-flatt?` in this example:

```
(cons "Fagan"
      (cons "Findler"
            (cons "Fisler"
                  (cons "Flanagan"
                        (cons "Flatt"
                              (cons "Felleisen"
                                    (cons "Friedman" '())))))))
```

What answer do you expect? □

Exercise 135. Here is another way of formulating the second `cond` clause in `contains-flatt?`:

```
... (cond
      [(string=? (first a-list-of-names) "Flatt") #true]
      [else (contains-flatt? (rest a-list-of-names))]) ...
```

Explain why this expression produces the same answers as the `or` expression in the version of [figure 41](#). Which version is better? Explain. □

Exercise 136. Develop the function `contains?`, which determines whether some given string occurs on a list of strings.

Note BSL actually comes with `member?`, a function that consumes two values and checks whether the first occurs in the second, which must be a list:

```
> (member? "Flatt" (cons "a" (cons "b" (cons "Flatt" '()))))
#true
```

Don't use `member?` to define the `contains?` function. ■

9.4 Computing with Lists

Since we are still using BSL, the rules of algebra—see the first intermezzo—tell us how to determine the value of expressions such as

```
(contains-flatt? (cons "Flatt" (cons "C" '())))
```

without DrRacket. Programmers must have an intuitive understanding of how this kind of calculation works, so we step through the one for this simple example:

```
==  
(cond  
  [(empty? (cons "Flatt" (cons "C" '()))) #false]  
  [(cons? (cons "Flatt" (cons "C" '())))  
   (or (string=? (cons "Flatt" (cons "C" '())) "Flatt")  
       (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))]))
```

The first step uses the usual substitution rule—or the beta rule as the first intermezzo calls it—to determine the value of an application. The result is a conditional expression, because, as an algebra teacher would say, the function is defined in a step-wise fashion. So here is how it continues:

```
==  
(cond  
  [#false #false]  
  [(cons? (cons "Flatt" (cons "C" '())))  
   (or (string=? (cons "Flatt" (cons "C" '())) "Flatt")  
       (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))]))  
==  
(cond  
  [(cons? (cons "Flatt" (cons "C" '())))  
   (or (string=? (cons "Flatt" (cons "C" '())) "Flatt")  
       (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))]))  
==  
(cond  
  [#true  
   (or (string=? (first (cons "Flatt" (cons "C" '())))) "Flatt")  
       (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))]))  
==  
(or (string=? (first (cons "Flatt" (cons "C" '())))) "Flatt")  
    (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))
```

To find the correct clause of the `cond` expression, we must determine the value of the conditions, one by one. Since a `consed` list isn't empty, the first condition's result is `#false` and we therefore eliminate the first `cond` clause. Finally the condition in the second clause evaluates to `#true` because `cons?` of a `consed` list holds. From here, it is just three more steps of arithmetic to get the final result:

```
==  
(or (string=? "Flatt" "Flatt")
```

```

  (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))
==

(or #true
  (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))
==

#true

```

First, `(first (cons "Flatt" ...))` is "Flatt" by the laws for `first`. Second, "Flatt" is a string and equal to "Flatt". Third, `(or #true X)` is `#true` regardless of what `X` is.

Exercise 137. Use DrRacket's stepper to check the calculation for

```
(contains-flatt? (cons "Flatt" (cons "C" '())))
```

Also use the stepper to determine the value of

```
(contains-flatt? (cons "A" (cons "Flatt" (cons "C" '()))))
```

What happens when "Flatt" is replaced with "B"? ■

Exercise 138. Validate with DrRacket's stepper

```

(our-first (our-cons "a" '()) == "a"
(our-rest (our-cons "a" '()) == "a"

```

See [What Is '\(\)](#), [What Is cons](#) for the definitions of these functions. ■

10 Designing with Self-Referential Data Definitions

;; A List-of-strings is one of
;; -- '()
;; -- (cons String List-of-strings)

(define (fun-for-los a-list-of-strings)
(cond
 [(empty? a-list-of-strings) ...]
 [else ;(cons? a-list-of-strings)
 (... (first a-list-of-strings)
 ... (rest a-list-of-strings) ...)])))

Figure 42: Arrows for self-references in data definitions and templates

At first glance, self-referential data definitions seem to be far more complex than those for mixed data. But, as the example of `contains-flatt?` shows, the six steps of the design recipe still work. Nevertheless, in this section we generalize the design recipe so that it works even better for self-referential data definitions. The new parts concern the process of discovering when a self-referential data definition is needed, deriving a template, and defining the function body:

1. If a problem statement discusses compound information of arbitrary size, you need a self-referential data definition. At this point, you have seen only one such class, [List-of-names](#). The left side of [figure 42](#) shows how to define *List-of-strings* in the same way. It is also straightforward to imagine lists of [Numbers](#), lists of [Booleans](#), lists of [Images](#), and so on.

Numbers also seem to be arbitrarily large. For inexact numbers, this is an illusion. For precise integers, this is indeed the case. Dealing with integers is therefore a part of this chapter.

For a self-referential data definition to be valid, it must satisfy two conditions. First, it must contain at least two clauses. Second, at least one of the clauses must not refer back to the class of data that is being defined. It is good practice to identify the self-references explicitly with arrows from the references in the data definition back to the term being defined; see [figure 42](#) for an example of such an annotation.

You must check the validity of self-referential data definitions with the creation of examples. Start with the clause that does not refer to the data definition; continue with the other one, using the first example where the clause refers to the definition itself. For the data definition in [figure 42](#), you thus get lists like the following three:

```
; by the first clause
'()
; by the second clause and the preceding example
(define "a" '())
; again by the second clause and the preceding example
(define "b" (cons "a" '()))
```

If it is impossible to generate examples from the data definition, it is invalid. If you can generate examples but you can't see how to generate larger and larger examples, the definition may not live up to its interpretation.

- Nothing changes about the header material: the signature, the purpose statement, and the dummy definition. When you do formulate the purpose statement, focus on **what** the function computes **not how** it goes about it, especially not how it goes through instances of the given data.

Here is an example to make this design recipe concrete:

```
; List-of-strings -> Number
; count how many strings also contains
(define (how-many alos)
  0)
```

The purpose statement clearly states that the function just counts the strings on the given input; there is no need to think ahead about how you might formulate this idea as a BSL function.

- When it comes to functional examples, be sure to work through inputs that use the self-referential clause of the data definition several times. It is the best way to formulate tests that cover the entire function definition later.

For our running example, the purpose statement almost generates functional examples by itself from the data examples:

given	wanted
'()	0
(cons "a" '())	1
(cons "b" (cons "a" '()))	2

The first row is about the empty list, and we know that empty list contains nothing. The second row is a list of one string, so 1 is the desired answer. The last row is about a list

of two strings.

4. At the core, a self-referential data definition looks like a data definition for mixed data. The development of the template can therefore proceed according to the recipe in [Itemizations and Structures](#). Specifically, we formulate a `cond` expression with as many `cond` clauses as there are clauses in the data definition, match each recognizing condition to the corresponding clause in the data definition, and write down appropriate selector expressions in all `cond` lines that process compound values.

Question	Answer
Does the data definition distinguish among different subclasses of data?	Your template needs as many <code>cond</code> clauses as subclasses that the data definition distinguishes.
How do the subclasses differ from each other?	Use the differences to formulate a condition per clause.
Do any of the clauses deal with structured values?	If so, add appropriate selector expressions to the clause.
Does the data definition use self-references?	Formulate “natural recursions” for the template to represent the self-references of the data definition.
If the data definition refers to some other data definition, where is this cross-reference to another data definition?	Specialize the template for the other data definition. Refer to this template. See Designing with Itemizations, Again , steps 4 and 5 of the design recipe.

Figure 43: How to translate a data definition into a template

[Figure 43](#) expresses this idea as a question-and-answer game. In the left column it states questions about the data definition for the argument, and in the right column it explains what the answer means for the construction of the template.

If you ignore the last row and apply the first three questions to any function that consumes a [List-of-strings](#), you arrive at this shape:

```
(define (fun-for-los alos)
  (cond
    [(empty? alos) ...]
    [else
      (... (first alos) ... (rest alos) ...))])
```

Recall, though, that the purpose of a template is to express the data definition as a function layout. That is, a template expresses as code what the data definition for the input expresses as a mix of English and BSL. Hence all important pieces of the data definition must find a counterpart in the template, and this guideline should also hold when a data definition is self-referential—contains an arrow from inside the definition to the term being defined. In particular, when a data definition is self-referential in the *i*th clause and the *k*th field of the structure mentioned there, the template should be self-referential in the *i*th `cond` clause and the selector expression for the *k*th field. For each such selector expression, add an arrow back to the function parameter. At the end, your template must have as many arrows as we have in the data definition.

[Figure 42](#) illustrates this idea with the template for functions that consume [List-of-strings](#) shown side by side with the data definition. Both contain one self-referential

arrow that originates in the second clause—the `rest` field and selector, respectively—and points back to the top of the respective definitions.

Since BSL and most programming languages are text-oriented, you must use an alternative to the arrow, namely, a self-applications of the function to the appropriate selector expression:

```
(define (fun-for-los alos)
  (cond
    [(empty? alos) ...]
    [else
      (... (first alos) ...
            ... (fun-for-los (rest alos)) ...)])))
```

We refer to a self-use of a function as *recursion* and in the first four parts of the book as *natural recursion*.

- For the function body we start with those `cond` lines that do not contain natural recursions. They are called *base cases*. The corresponding answers are typically easy to formulate or are already given by the examples.

Then we deal with the self-referential cases. We start by reminding ourselves what each of the expressions in the template line computes. For the natural recursion we assume that the function already works as specified in our purpose statement. This last step is a leap of faith, but as you will see, it always works.

For the curious among our readers, the design recipe for arbitrarily large data corresponds to so-called “proofs by induction” in mathematics and the “leap of faith” represents the assumption of the induction hypothesis for the inductive step of such a proof. Courses on logic prove the validity of this proof technique with a so-called Induction Theorem.

The rest is then a matter of combining the various values.

Question	Answer
What are the answers for the non-recursive <code>cond</code> clauses?	The examples should tell you which values you need here. If not, formulate appropriate examples and tests.
What do the selector expressions in the recursive clauses compute?	The data definitions tell you what kind of data these expressions extract and the interpretations of the data definitions tell you what this data represents.
What do the natural recursions compute?	Use the purpose statement of the function to determine what the value of the recursion means not how it computes this answer . If the purpose statement doesn't tell you the answer, improve the purpose statement.
How can the function	Find a function in BSL that combines the values. Or, if that doesn't work, make a wish for a helper function.

combine these values to get the desired answer?

For many functions, this last step is straightforward. The purpose, the examples, and the template together tell you which function or expression “combines” the available values into the proper result. We refer to this function or expression as *combinator*, slightly abusing existing terminology.

So, if you are stuck here, ...

... arrange the examples from the third step in a table. Place the given input in the first column and the desired output in the last column. In the intermediate columns enter the values of the selector expressions and the natural recursion(s). Add examples until you see a pattern emerge that suggests a combinator.

If the template refers to some other template, what does the auxiliary function compute?

Consult the other function’s purpose statement and examples to determine what it computes and assume you may use the result even if you haven’t finished the design of this helper function.

Figure 44: How to turn a template into a function definition

Figure 44 formulates questions and answers for this step. Let’s use this game to complete the definition of how-many. Renaming the fun-for-los template to how-many gives us this much:

```
; List-of-strings -> Number
; determines how many strings are on alos
(define (how-many alos)
  (cond
    [(empty? alos) ...]
    [else
      (... (first alos) ... (how-many (rest alos)) ...)]))
```

As the functional examples already suggest, the answer for the base case is `0`. The two expressions in the second clause compute the `first` item and the number of strings in `(rest alos)`. To compute how many strings there are on all of `alos`, we just need to add `1` to the value of the latter expression:

```
(define (how-many alos)
  (cond
    [(empty? alos) 0]
    [else (+ (how-many (rest alos)) 1)]))
```

Dr. Felix Klock suggested this table-based approach to guessing the combinator.

Finding the correct way to combine the values into the desired answer isn’t always as easy. Novice programmers often get stuck with this step. As the question-and-answer game suggests, it is a good idea to arrange the functional examples into a table that also spells out the values of the expressions in the template. Figure 45 shows what this table looks like for our how-many example. The leftmost column lists the sample inputs, while

the rightmost column contains the desired answers for these inputs. The three columns in between show the values of the template expressions: `(first alos)`, `(rest alos)`, and `(how-many (rest alos))`, which is the natural recursion. If you stare at this table long enough, you recognize that the result column is always one more than the values in the natural recursion column. You may thus guess that

```
| (+ (how-many (rest alos)) 1)
```

is the expression that computes the desired result. Since DrRacket is fast at checking these kinds of guesses, plug it in and click *RUN*. If the examples-turned-into-tests pass, think through the expression to convince yourself it works for all lists; otherwise add more example rows to the table until you have a different idea.

The table also points out that not all selector expressions in the template are necessarily relevant for the function definition. Here `(first alos)` is not needed to compute the final answer—which is quite a contrast to `contains-flatt?`, which uses both expressions from the template.

As you work your way through the rest of this book, keep in mind that, in many cases, the combination step can be expressed with BSL's primitives, say, `+`, `and`, or `cons`. In some cases, though, you may have to make a wish, that is, design an auxiliary function. Finally, in yet other cases, you may need nested conditions.

5. Finally, make sure to turn all examples into tests, that these tests pass, and that running them covers all the pieces of the function.

Here are our examples for `how-many` turned into tests:

```
| (check-expect (how-many '()) 0)
  (check-expect (how-many (cons "a" '())) 1)
  (check-expect (how-many (cons "b" (cons "a" '()))) 2)
```

Remember it is best to formulate examples directly as tests and BSL allows this. Doing so also helps if you need to resort to the table-based guessing approach of the preceding step.

alos	<code>(first alos)</code>	<code>(rest alos)</code>	<code>(how-many (rest alos))</code>	<code>(how-many alos)</code>
<code>(cons "a" '())</code>	"a"	'()	0	1
<code>(cons "b" (cons "a" '()))</code>	"b"	<code>(cons "a" '())</code>	1	2
<code>(cons "x" (cons "b" (cons "a" '())))</code>	"x"	<code>(cons "b" (cons "a" '()))</code>	2	3

Figure 45: Tabulating arguments, intermediate values, and results

Figure 46 summarizes the design recipe of this section in a tabular format. The first column names the steps of the design recipe, the second the expected results of each step. In the third column, we describe the activities that get you there. The figure is tailored to the kind of self-referential list definitions we use in this chapter. As always, practice helps you master the process, so we strongly recommend that you tackle the following exercises, which ask you to apply the recipe to several kinds of examples.

You may want to copy [figure 46](#) onto one side of an index card and write down your favorite versions of the questions and answers in [figure 43](#) and [figure 44](#) onto the back of it. Then carry it with you for future reference. Sooner or later, the design steps become second nature and you won't think about them anymore. Until then, refer to your index card whenever you are stuck with the design of a function.

steps	outcome	activity
problem analysis	data definition	identify the information that must be represented; develop a data representation; know how to create data for a specific item of information and how to interpret a piece of data as information; identify self-references in the data definition
header	signature; purpose statement; dummy definition	write down a signature, using the names from the data definitions; formulate a concise purpose statement; create a dummy function that produces a constant value from the specified range
examples	examples and tests	work through several examples, at least one per clause in the (self-referential) data definition; turn them into check-expect tests
template	function template	translate the data definition into a template: one cond clauses per clause; one condition per clause to distinguish the cases; selector expressions per clause if the condition identifies a structure; one natural recursion per self-reference in the data definition
definition	full-fledged definition	find a function that combines the values of the expressions in the cond clauses into the expected answer
test	validated tests	run the tests and validate that they all pass

Figure 46: Designing a function for self-referential data

10.1 Finger Exercises: Lists

Exercise 139. Compare the template for `how-many` and `contains-flatt?`. Ignoring the function name, they are the same. Explain why. ▀

Exercise 140. Here is a data definition for representing amounts of money:

```
; A List-of-amounts is one of:  
; - '()  
; - (cons PositiveNumber List-of-amounts)  
; interpretation a List-of-amounts represents some amounts of money
```

Create some examples to make sure you understand the data definition. Also add an arrow for the self-reference.

Design the `sum` function, which consumes a [List-of-amounts](#) and computes the sum of the amounts. Use DrRacket's stepper to see how `(sum l)` works for a short list `l` in [List-of-amounts](#). ■

Exercise 141. Now take a look at this data definition:

```
; A List-of-numbers is one of:  
; - ()  
; - (cons Number List-of-numbers)
```

Some elements of this class of data are appropriate inputs for `sum` from [exercise 140](#) and some aren't.

Design the function `pos?`, which consumes a [List-of-numbers](#) and determines whether all numbers are positive numbers. In other words, if `(pos? l)` yields `#true`, then `l` is an element of [List-of-amounts](#). Use DrRacket's stepper to understand how `pos?` works for `(cons 5 '())` and `(cons -1 '())`.

Also design `checked-sum`. The function consumes a [List-of-numbers](#). It produces their sum if the input also belongs to [List-of-amounts](#); otherwise it signals an error. **Hint** Recall to use `check-error`.

What does `sum` compute for an element of [List-of-numbers](#)? ■

Exercise 142. Design

- `all-true`, which consumes a list of [Boolean](#) values and determines whether all of them are `#true`. In other words, if there is any `#false` on the list, the function produces `#false`; otherwise it produces `#true`.
- `one-true`, which consumes a list of Boolean values and determines whether at least one item on the list is `#true`.

Follow the design recipe: start with a data definition for lists of Boolean values and don't forget to make up examples. You may also wish to employ the table-based guessing approach; it tends to help understand the answer needed for the base case.

Use DrRacket's stepper to see how these functions process the lists `(cons #true '())`, `(cons #false '())`, and `(cons #true (cons #false '()))`. ■

Exercise 143. If you are asked to design the function `cat`, which consumes a list of strings and appends them all into one long string, you are guaranteed to end up with this partial definition:

```
; List-of-string -> String  
; concatenate all strings in l into one long string  
  
(check-expect (cat '()) "")  
(check-expect (cat (cons "a" (cons "b" '())))) "ab")  
(check-expect (cat (cons "ab" (cons "cd" (cons "ef" '()))))) "abcdef")  
  
(define (cat l)  
  (cond  
    [(empty? l) ""]  
    [else (... (first l) ... (cat (rest l)) ...)]))
```

Fill in the table below:

<code>l</code>	<code>(first l)</code>	<code>(rest l)</code>	<code>(cat (rest l))</code>	<code>(cat l)</code>
<code>(cons "a" (cons "b" '()))</code>	???	???	???	"ab"
<code>(cons "ab" (cons "cd" (cons "ef" '()))))</code>	???	???	???	"abcdef"

Guess a function that can create the desired result from the values computed by the subexpressions.

Use DrRacket's stepper to evaluate `(cat (cons "a" '()))).` ▀

Exercise 144. Design `ill-sized?`. The function consumes a list of images `loi` and a positive number `n`. It produces the first image on `loi` that is not an `n` by `n` square; if it cannot find such an image, it produces `#false`.

Hint Use

```
; ImageOrFalse is one of:  
; - Image  
; - #false
```

for the result part of the signature. ▀

10.2 Non-empty Lists

Now you know enough to use `cons` and to create data definitions for lists. If you solved (some of) the exercises at the end of the preceding section, you can deal with lists of various flavors of numbers, lists of Boolean values, lists of images, and so on. In this section we continue to explore what lists are and how to process them.

Let us start with the simple-looking problem of computing the average of a list of temperatures. To simplify things, we provide the data definitions:

```
; A List-of-temperatures is one of:  
; - '()  
; - (cons CTemperature List-of-temperatures)  
  
; A CTemperature is a Number greater or equal to -273.
```

For our intentions, you should think of temperatures as plain numbers, but the second data definition reminds you that in reality not all numbers are temperatures and you should keep this in mind.

The header material is straightforward:

```
; List-of-temperatures -> Number  
; computes the average temperature  
(define (average alot) 0)
```

Making up examples for this problem is also easy, and so we just formulate one test:

```
(check-expect (average (cons 1 (cons 2 (cons 3 '())))) 2)
```

The expected result is of course the sum of the temperatures divided by the number of temperatures.

A moment's thought should tell you that the template for average should be similar to the ones we have seen so far:

```
(define (average alot)
  (cond
    [(empty? alot) ...]
    [(cons? alot)
      (... (first alot) ... (average (rest alot)) ...)])))
```

The two `cond` clauses follow from the two clauses of the data definition; the questions distinguish empty lists from non-empty lists; and the natural recursion is needed to compute the average of the rest, which is also a list of numbers.

The problem is that it was too difficult to turn this template into a working function definition. The first `cond` clause needs a number that represents the average of an empty collection of temperatures, but there is no such number. Even if we ignore this problem for a moment, the second clause demands a function that combines a temperature and an average for many other temperatures into another average. Although it is isn't impossible to compute this average, it is not what you learned to do and it isn't natural.

When we compute the average of a bunch of temperatures, we divide their sum by the number of temperatures. We said so when we formulated our trivial little example. This sentence, however, suggests that `average` is a function of three tasks: division, summing, and counting. Our guideline from [Fixed-Size Data](#) tells us to write one function per task and if we do so, the design of `average` is obvious:

```
; List-of-temperatures -> Number
; computes the average temperature
(define (average alot)
  (/ (sum alot)
    (how-many alot)))

; List-of-temperatures -> Number
; adds up the temperatures on the given list
(define (sum alot) 0)

; List-of-temperatures -> Number
; counts the temperatures on the given list
(define (how-many alot) 0)
```

The last two function definitions are wishes of course for which we need to design complete definitions. Doing so is fortunately easy because `how-many` from above works for [List-of-strings](#) and [List-of-temperatures](#) (why?) and because the design of `sum` follows the same old routine:

```
; List-of-temperatures -> Number
; adds up the temperatures on the given list
(define (sum alot)
  (cond
    [(empty? alot) 0]
    [else (+ (first alot) (sum (rest alot))))]))
```

Stop! Use the example for `average` to create one for `sum` and ensure that the test runs

properly. Then ensure that the above test for average works out.

When you read this definition of average now, it is obviously correct simply because it directly corresponds to what everyone learns about averaging in school. Still, programs run not just for us but for others. In particular, others should be able to read the signature and use the function and expect an informative answer. But, our definition of average does not work for empty lists of temperatures.

Exercise 145. Determine how average behaves in DrRacket when applied to the empty list of temperatures. Then design checked-average, a function that produces an informative error message when it is applied to '().

From a theoretical perspective, [exercise 145](#) shows that average is a *partial function* because it raises an error for '(). The alternative development explains that, in this case, we can narrow down the domain and create a *total function*.

An alternative solution is to inform future readers through the signature that average doesn't work for empty lists. To do so, we need a data representation for lists of temperatures that excludes '(), that is, a data definition like this:

```
; A NEList-of-temperatures is one of:
; - ???
; - (cons CTemperature NEList-of-temperatures)
```

The question is with what we should replace “???” so that the '() list is excluded but all other lists of temperatures are still constructable. One hint is that while the empty list is the shortest list, any list of one temperature is the next shortest list. In turn, this suggests that the first clause should describe all possible lists of one temperature:

```
; A NEList-of-temperatures is one of:
; - (cons CTemperature '())
; - (cons CTemperature NEList-of-temperatures)
; interpretation non-empty lists of measured temperatures
```

While this definition differs from the preceding list definitions, it shares the critical elements: a self reference and a clause that does **not** use a self-reference. Strict adherence to the design recipe demands that you make up some examples of [NEList-of-temperatures](#) to ensure that the definition makes sense. As always, you should start with the base clause, meaning the example must look like this:

```
(cons ccc '())
```

where ccc must be replaced by a CTemperature, like thus: (cons -273 '()). Furthermore, it is easy to check that all non-empty elements of [List-of-temperatures](#) are also elements of [NEList-of-temperatures](#). For example,

1. (cons 1 (cons 2 (cons 3 '()))) fits the bill if (cons 2 (cons 3 '())) does;
2. and (cons 2 (cons 3 '())) belongs to [NEList-of-temperatures](#) because (cons 3 '()) is an element of [NEList-of-temperatures](#), as confirmed before.

Check for yourself that there is no limit on the size of NEList-of-temperatures.

Let us now return to the problem of designing average so that everyone knows it is for non-empty lists only. With the definition of [NEList-of-temperatures](#) we now have the

means to say what we want in the signature:

```
; NEList-of-temperatures -> Number
; computes the average temperature

(check-expect (average (cons 1 (cons 2 (cons 3 '())))) 2)

(define (average anelot)
  (/ (sum anelot)
      (how-many anelot)))
```

Naturally the rest remains the same: the purpose statement, the example-test, and the function definition. After all, the very idea of computing the average assumes a non-empty collection of numbers, and that was the entire point of our discussion.

Exercise 146. Would `sum` and `how-many` work for `NEList-of-temperatures` even if they were designed for inputs from `List-of-temperatures`? If you think they don't work, provide counter-examples. If you think they would, explain why. ■

Nevertheless the definition also raises the question how to design `sum` and `how-many` because they consume instances of `NEList-of-temperatures` now. Here is the obvious result of the first three steps of the design recipe:

```
; NEList-of-temperatures -> Number
; computes the sum of the given temperatures
(check-expect (sum (cons 1 (cons 2 (cons 3 '())))) 6)
(define (sum anelot) 0)
```

The example is adapted from the example for `average`; the header definition produces a number as requested, but the wrong one for the given test case.

The fourth step is the most interesting part of the design of `sum` for `NEList-of-temperatures`. All preceding examples of design demand a template that distinguishes empty lists from non-empty, i.e., `consed`, lists because the data definitions have an appropriate shape. This is not true for `NEList-of-temperatures`. Here both clauses add `consed` lists. The two clauses differ, however, in the `rest` field of these lists. In particular, the first clause always uses `'()` in the `rest` field and the second one uses a `consed` list instead. Hence the proper questions to distinguish data from the first and the second clause first extract the `rest` field and then use a predicate:

```
; NEList-of-temperatures -> Number
(define (sum anelot)
  (cond
    [(empty? (rest anelot)) ...]
    [(cons? (rest anelot)) ...]))
```

As always, `else` would be an acceptable replacement for `(cons? (rest anelot))` in the second clause, though it would also be less informative.

Next you should inspect both clauses and determine whether one or both of them deal with `anelot` as if it were a structure. This is of course the case, which the unconditional use of `rest` on `anelot` demonstrates. Put differently, you should add appropriate selector expressions to the two clauses:

```
(define (sum anelot)
  (cond
```

```
[(empty? (rest anelot)) (... (first anelot) ...))
[(cons? (rest anelot))
(... (first anelot) ... (rest anelot) ...))])]
```

Before you read on, explain why the first clause does not contain the selector expression **(rest** anelot).

The final question of the template design concerns self-references in the data definition. As you know, NEList-of-temperatures contains one and therefore the template for sum demands one recursive use:

```
(define (sum anelot)
  (cond
    [(empty? (rest anelot)) (... (first anelot) ...))
    [(cons? (rest anelot))
      (... (first anelot) ... (sum (rest anelot)) ...))]))]
```

Specifically, sum must be called on **(rest** anelot) in the second clause because the second clause of the data definition is self-referential at the analogous point.

For the fifth design step, let us understand how much we already have. Since the first **cond** clause looks significantly simpler than the second one with its recursive function call, you should start with that one. In this particular case, the condition says that sum is applied to a list with exactly one temperature, **(first** anelot). Clearly, this one temperature is the sum of all temperatures on the given list:

```
(define (sum anelot)
  (cond
    [(empty? (rest anelot)) (first anelot)]
    [(cons? (rest anelot))
      (... (first anelot) ... (sum (rest anelot)) ...))]))]
```

The second clause says that the list consists of a temperature and at least one more; **(first** anelot) extracts the first position and **(rest** anelot) the remaining ones. Furthermore, the template suggests to use the result of **(sum (rest** anelot)). But sum is the function that you are defining, and you can't possibly know **how** it uses **(rest** anelot). All you do know is what the purpose statement says, namely, that sum adds all the temperatures on the given list, which is **(rest** anelot). If this statement is true, then **(sum (rest** anelot)) adds up all but one of the numbers of anelot. To get the total, the function just has to add the first temperature:

```
(define (sum anelot)
  (cond
    [(empty? (rest anelot)) (first anelot)]
    [(cons? (rest anelot)) (+ (first anelot) (sum (rest anelot)))]))]
```

If you now run the example/test for this function, you will see that the leap of faith is justified. Indeed, for reasons beyond this book, this leap of faith is **always** justified, which is why it is an inherent part of the design recipe.

Exercise 147. Design **sorted>?**. The function consumes a NEList-of-temperatures. It produces **#true** if the temperatures are sorted in descending order, that is, if the second is smaller than the first, the third smaller than the second, and so on. Otherwise it produces **#false**.

Hint Consider using the table-based guessing approach; it tends to help understand the

answer needed for the base case.

Hint This problem is another one where the table-based method for guessing the combinator. Here is a partial table for a number of examples:

<code>l</code>	<code>(first l)</code>	<code>(rest l)</code>	<code>(sorted? (rest l))</code>	<code>(sorted? l)</code>
<code>(cons 1 (cons 2 '()))</code>	1	???	#false	#false
<code>(cons 3 (cons 2 '()))</code>	3	<code>(cons 2 '())</code>	???	#true
<code>(cons 0 (cons 3 (cons 2 '())))</code>	0	<code>(cons 3 (cons 2 '()))</code>	???	???

Fill in the rest of the table. Then try to create an expression that computes the result from the pieces. ■

Exercise 148. Design `how-many` for [NList-of-temperatures](#). Doing so completes `average`, so ensure that `average` passes all of its tests, too. ■

Exercise 149. Develop a data definition for *NList-of-Booleans*, a representation of non-empty lists of Boolean values. Then re-design the functions `all-true` and `one-true` from [exercise 142](#). ■

Exercise 150. Compare the function definitions from this section (`sum`, `how-many`, `all-true`, `one-true`) with the corresponding function definitions from the preceding sections. Is it better to work with data definitions that accommodate empty lists as opposed to definitions for non-empty lists? Why? Why not? ■

10.3 Natural Numbers

The BSL programming language supplies many functions that consume lists and a few that produce them, too. Among those is `make-list`, which consumes a number `n` together with some other value `v` and produces a list that contains `v n` times. Here are some examples:

```
> (make-list 2 "hello")
'("hello" "hello")
> (make-list 3 #true)
'(#true #true #true)
> (make-list 0 17)
'()
```

In short, even though this function consumes atomic data, it produces arbitrarily large pieces of data. Your question should be how this is possible.

Our answer is that `make-list`'s input isn't just a number, it is a special kind of number. In kindergarten you called these numbers “counting numbers,” i.e., these numbers are used to count objects. In computer science, these numbers are dubbed *natural numbers*. Unlike regular numbers, natural numbers come with a data definition:

```
; A N is one of:
; - 0
; - (add1 N)
```

```
; interpretation represents the natural numbers or counting numbers
```

And as you can easily see, this data definition is self-referential just like the data definition for various forms of lists.

Let us take a close look at the data definition of natural numbers. The first clause says that `0` is a natural number; it is of course used to say that there is no object to be counted. The second clause tells you that if n is a natural number, then $n+1$ is one too, because `add1` is a function that adds `1` to whatever number it is given. We could write this second clause as `(+ n 1)` but the use of `add1` is supposed to signal that this addition is special.

What is special about this use of `add1` is that it acts more like a constructor from some structure type definition than a regular numeric function. For that reason, BSL also comes with the function `sub1`, which is the “selector” corresponding to `add1`. Given any natural number m not equal to `0`, you can use `sub1` to find out the number that went into the construction of m . Put differently, `add1` is like `cons` and `sub1` is like `first` and `rest`.

At this point you may wonder what the predicates are that distinguish `0` from those natural numbers that are not `0`. There are two, just as for lists: `zero?`, which determines whether some given number is `0`, and `positive?`, which determines whether some number is larger than `0`.

Now you are in a position to design functions such as `make-list` yourself. The data definition is already available, so let us add the header material:

```
; N String -> List-of-strings
; creates a list of n strings s

(check-expect (copier 2 "hello") (cons "hello" (cons "hello" '())))
(check-expect (copier 0 "hello") '())

(define (copier n s)
  '())
```

Developing the template is the next step. The questions for the template suggest that `copier`'s body is a `cond` expression with two clauses: one for `0` and one for positive numbers. Furthermore, `0` is considered atomic and positive numbers are considered structured values, meaning the template needs a selector expression in the second clause. Last but not least, the data definition for `N` is self-referential in the second clause. Hence the template needs a recursive application to the correspond selector expression in the second clause:

```
(define (copier n s)
  (cond
    [(zero? n) ...]
    [(positive? n) (... (copier (sub1 n) s) ...))])
```

```
; N String -> List-of-strings
; creates a list of n strings s

(check-expect (copier 2 "hello") (cons "hello" (cons "hello" '())))
(check-expect (copier 0 "hello") '())

(define (copier n s)
  (cond
```

```
[(zero? n) '()]
[(positive? n) (cons s (copier (sub1 n) s))])]
```

Figure 47: Creating a list of copies

[Figure 47](#) contains a complete definition of the `copier` function, as obtained from its template. Let us reconstruct this step carefully. As always, we start with the `cond` clause that has no recursive calls. Here the condition tells us that the (important) input is `0` and that means the function must produce a list with `0` items, that is, `none`. Of course, working through the second example has already clarified this case. Next we turn to the other `cond` clause and remind ourselves what the expressions from the template compute:

1. `(sub1 n)` extracts the natural number that went into the construction of `n`, which we know is larger than `0`;
2. `(copier (sub1 n) s)`, according to the purpose statement of `copier`, produces a list of `(sub1 n)` strings `s`.

But the function is given `n` and must therefore produce a list with `n` strings `s`. Given a list with one too few strings, it is easy to see that the function must simply `cons` one `s` onto the result of `(copier (sub1 n) s)`. And that is precisely what the second clause specifies.

At this point, you should run the tests to ensure that this function works at least for the two worked examples. In addition, you may wish to use the function on some additional inputs.

Exercise 151. Does `copier` function properly when you apply it to a natural number and a Boolean or an image? Or do you have to design another function? Read [Similarities Everywhere](#) for an answer.

An alternative definition of `copier` might use `else` for the second condition:

```
(define (copier.v2 n s)
  (cond
    [(zero? n) '()]
    [else (cons s (copier.v2 (sub1 n) s))]))]
```

How do `copier` and `copier.v2` behave when you apply them to `0.1` and "xyz"? Explain. Use DrRacket's stepper to confirm your explanation. ■

Exercise 152. Design the function `add-to-pi`. It consumes a natural number `n` and adds it to `pi` **without** using `+` from BSL. Here is a start:

```
; N -> Number
; computes (+ n pi) without using +
(check-within (add-to-pi 3) (+ 3 pi) 0.001)

(define (add-to-pi n)
  pi)
```

Once you have a complete definition, generalize the function to `add`, which adds a natural number `n` to some arbitrary number `x` without using `+`. Why does the skeleton use `check-within`? ■

Exercise 153. Design the function `multiply`. It consumes a natural number `n` and

multiples it with some arbitrary number x without using $*$.

Use DrRacket's stepper to evaluate `(multiply 3 x)` for any x you like. How does `multiply` relate to what you know from grade school. ▀

Exercise 154. Design two functions: `col` and `row`.

The function `col` consumes a natural number n and an image `img`. It produces a column—a vertical arrangement—of n copies of `img`.

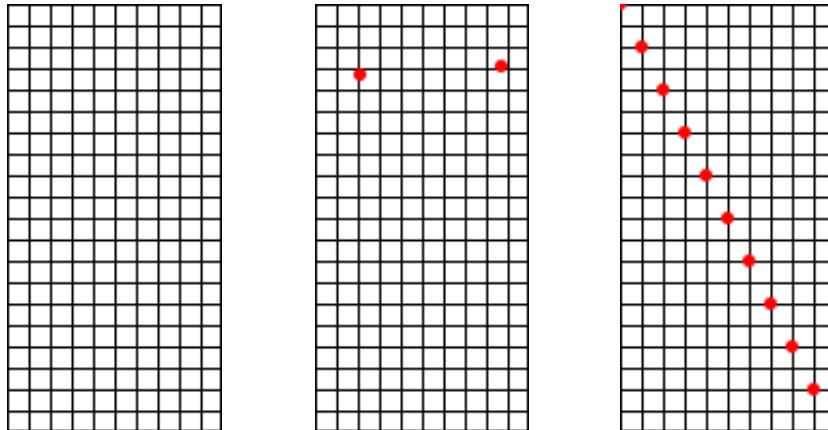
The function `row` consumes a natural number n and an image `img`. It produces a row—a horizontal arrangement—of n copies of `img`. ▀

Exercise 155. The goal of this exercise is to visualize the result of a 1968-style European student riot. Here is the rough idea. A small group of students meets to make paint-filled balloons, enters some lecture hall and randomly throws the balloons at the attendees. Your world program displays how the balloons color the seats in the lecture hall.

Use the two functions from [exercise 154](#) to create a rectangle of 8 by 18 squares, each of which has size 10 by 10. Place it in an `empty-scene` of the same size. This image is your lecture hall.

Design `add-balloon`. The function consumes a list of `Posn` whose coordinates fit into the dimensions of the lecture hall. It produces an image of the lecture hall with red dots added as specified by the `Posns`.

Here are outputs of our solution when given some list of `Posns`:



The leftmost is the clean lecture hall, the second one is after two balloons have hit, and the last one is a highly unlikely distribution of 10 hits. Where is the 10th? ▀

10.4 Russian Dolls

Wikipedia defines a Russian doll as “ a set of dolls of decreasing sizes placed one inside the other.” The paragraph is accompanied by an appropriate picture of dolls:



Of course, here the dolls are taken apart so that the viewer can see them all.

The problem may strike you as somewhat abstract or even absurd; after all it isn't clear why you would want to represent Russian dolls or what you would do with such a representation. Suspend your disbelief and read along; it is a worthwhile exercise.

Now consider the problem of representing such Russian dolls with BSL data. With a little bit of imagination, it is easy to see that an artist can create a nest of Russian dolls that consists of an arbitrary number of dolls. After all, it is always possible to wrap another layer around some given Russian doll. Then again, you also know that deep inside there is a solid doll without anything inside.

For each layer of a Russian doll, we could care about many different things: its size, though it is related to the nesting level; its color; the image that is painted on the surface; and so on. Here we just pick one, namely the color of the doll, which we represent with a string. Given that, we know that each layer of the Russian doll has two properties: its color and the doll that is inside. To represent pieces of information with two properties, we always define a structure type:

```
| (define-struct layer [color doll])
```

And then we add a data definition:

```
| ; An RD (russian doll) is one of:  
| ; - String  
| ; - (make-layer String RD)
```

Naturally, the first clause of this data definition represents the innermost doll or, to be precise, its color. The second clause is for adding a layer around some given Russian doll. We represent this with an instance of `layer`, which obviously contains the color of the doll and one other field: the doll that is nested immediately inside of this doll.

Take a look at this doll:

RD

It consists of three dolls. The red one is the innermost one, the green one sits in the middle, and the yellow is the current outermost wrapper. To represent this doll with an element of `RD`, you start on either end. We proceed from the inside out. The red doll is easy to represent as an `RD`. Since nothing is inside and since it is red, the string "`red`" will do fine. For the second layer, we use

```
| (make-layer "green" "red")
```

which says that a green (hollow) doll contains a the red doll. Finally, to get the outside we just wrap another layer around this last doll:

```
(make-layer "yellow" (make-layer "green" "red"))
```

This process should give you a good idea how to go from any set of colored Russian dolls to a data representation. But keep in mind that a programmer must also be able to do the converse, that is, go from a piece of data to concrete information. In this spirit, draw a schematic Russian doll for the following element of **RD**:

```
(make-layer "pink" (make-layer "black" "white"))
```

You might even try this in BSL.

Now that we have a data definition and understand how to represent actual dolls and how to interpret elements of **RD** as dolls, we are ready to design functions that consume **RDs**. Specifically, let us design the function that counts how many dolls a Russian doll set contains. This sentence is a fine purpose statement and determines the signature, too:

```
; RD -> Number
; how many dolls are part of an-rd
```

As for examples, let us start with `(make-layer "yellow" (make-layer "green" "red"))`. The image above tells us that 3 is the expected answer because there are three dolls: the red one, the green one, and the yellow one. Just working through this one example, also tells us that when the input is a representation of this doll



then the answer is 1.

Step four demands the development of a template. Using the standard questions for this step produces this template:

```
; RD -> Number
; how many dolls are a part of an-rd
(define (depth an-rd)
  (cond
    [(string? an-rd) ...]
    [(layer? an-rd)
     (... (layer-color an-rd) ... (depth (layer-doll an-rd)) ...)]))
```

The number of **cond** clauses is determined by the number of clauses in the definition of **RD**. Each of the clauses specifically spells out what kind of data it is about, and that tells us which predicates to use: **string?** and **layer?**. While strings aren't compound data, instances of **layer** contain two values. If the function needs these values, it uses the selector expressions `(layer-color an-rd)` and `(layer-doll an-rd)`. Finally, the second clause of the data definition contains a self-reference from the **doll** field of the **layer** structure to the definition itself. Hence we need a recursive function call for the second selector expression.

The examples and the template almost dictate the function definition. For the non-recursive **cond** clause, the answer is obviously 1. For the recursive clause, the template expressions compute the following results:

- `(layer-color an-rd)` extracts the string that describes the color of the current layer;
- `(layer-doll an-rd)` extracts the doll contained within the current layer; and

- according to the purpose statement, `(depth (layer-doll an-rd))` determines how many dolls `(layer-doll an-rd)` consists of.

This last number is almost the desired answer but not quite because the difference between `an-rd` and `(layer-doll an-rd)` is one layer meaning one extra doll. Put differently, the function must add 1 to the recursive result to obtain the actual answer:

```
; RD -> Number
; how many dolls are a part of an-rd
(define (depth an-rd)
  (cond
    [(string? an-rd) 1]
    [(layer? an-rd) (+ (depth (layer-doll an-rd)) 1)])))
```

Note how the function definition does not use `(layer-color an-rd)` in the second clause. Once again, we see that the template is an organization schema for everything we know about the data definition but we may not need all of these pieces for the actual definition.

Last but not least, we must translate the examples into tests:

```
(check-expect (depth (make-layer "yellow" (make-layer "green" "red"))))
              3)
(check-expect (depth "red"))
              1)
```

If you run these in DrRacket, you will see that their evaluation touches all pieces of the definition of `depth`.

Exercise 156. Design the function `colors`. It consumes a Russian doll and produces a string of all colors, separate by a comma and a space. Thus our example should produce

```
"yellow, green, red"
```

Exercise 157. Design the function `inner`, which consumes an `RD` and produces the (color of the) innermost doll. Use DrRacket's stepper to evaluate `(inner rd)` for your favorite `rd`.

10.5 Lists and World

With lists and self-referential data definitions in general, you can design and run many more interesting world programs than with finite data. Just imagine you can now create a version of the space invader program from [Itemizations and Structures](#) that allows the player to fire as many shots from the tank as desired. Let us start with a simplistic version of this problem:

Sample Problem: Design a world program that simulates firing shots. Every time the “player” hits the space bar, the program adds a shot to the bottom of the canvas. These shots rise vertically at the rate of one pixel per tick.

If you haven’t designed a world program in a while, re-read section [Designing World Programs](#).

Designing a world program starts with a separation of information into constants and elements of the ever-changing state of the world. For the former we introduce physical and

graphical constants; for the latter we need to develop a data representation for world states. While the sample problem is relatively vague about the specifics, it clearly assumes a rectangular scenery with shots painted along a vertical line. Obviously the locations of the shots change with every clock tick but the size of the scenery and x-coordinate of the line of shots remain the same:

```
; physical constants
(define HEIGHT 80)
(define WIDTH 100)
(define XSHOTS (/ WIDTH 2))

; graphical constants
(define BACKGROUND (empty-scene WIDTH HEIGHT))
(define SHOT (triangle 3 "solid" "red"))
```

Nothing in the problem statement demands these particular choices, but as long as they are easy to change—meaning changing by editing a single definition—we have achieved our goal.

As for those aspects of the “world” that change, the problem statement mentions two. First, hitting the space bar adds a shot. Second, all the shots move straight up by one pixel per clock tick. Given that we cannot predict how many shots the player will “fire,” we use a list to represent them:

```
; A List-of-shots is one of:
; - '()
; - (cons Shot List-of-shots)
; interpretation the collection of shots fired and moving straight up
```

The one remaining question is how to represent each individual shot. We already know that all of them have the same x-coordinate and that this coordinate stays the same throughout. Furthermore, all shots look alike. Hence, their y-coordinates are the only property in which they differ from each other. It therefore suffices to represent each shot as a number:

```
; A Shot is a Number.
; interpretation the number represents the shot's y-coordinate
```

We could restrict the representation of shots to the interval of numbers below HEIGHT because we know that all shots are launched from the bottom of the canvas and that they then move up, meaning their y-coordinate continuously decreases.

Of course, you can also use a data definition like this to represent this world:

```
; A ShotWorld is List-of-numbers.
; interpretation each number represents the y-coordinate of a shot
```

After all, the above two definitions describe all list of numbers; we already have a definition for lists of numbers; and the name `ShotWorld` tells everyone what this class of data is about.

Once you have defined constants and developed a data representation for the states of the world, the key task is to pick which event handlers you wish to employ and to adapt their signatures to the given problem. The running example mentions clock ticks and the space bar, all of which translates into a wish list of three functions:

- the function that turns a world state into an image:

```
; ShotWorld -> Image
; adds each y on w at (MID,y) to the background image
(define (to-image w)
  BACKGROUND)
```

because the problem demands a visual rendering;

- one for dealing with tick events:

```
; ShotWorld -> ShotWorld
; moves each shot up by one pixel
(define (tock w)
  w)
```

- and one function for dealing with key events:

```
; ShotWorld KeyEvent -> ShotWorld
; adds a shot to the world if the space bar was hit
(define (keyh w ke)
  w)
```

Don't forget that in addition to the initial wish list, you also need to define a `main` function that actually sets up the world and installs the handlers. [Figure 48](#) includes this one function that is not designed but defined as a modification of standard schema.

Let us start with the design of `to-image`. We have its signature, purpose statement and header, so we need examples next. Since the data definition has two clauses, there should be at least two examples: `'()` and a `consed` list, say, `(cons 10 '())`. The expected result for `'()` is obviously `BACKGROUND`; if there is a y-coordinate, though, the function must place the image of a shot at MID and the specified coordinate:

```
(check-expect (to-image (cons 10 '())))
              (place-image SHOT XSHOTS 10 BACKGROUND))
```

Before you read on, work through an example that applies `to-image` to a list of two shot representations. It helps to do so to understand **how** the function works.

The fourth step of the design is to translate the data definition into a template:

```
; ShotWorld -> Image
(define (to-image w)
  (cond
    [(empty? w) ...]
    [else ... (first w) ... (to-image (rest w)) ...]))
```

The template for data definitions for lists is so familiar now that it doesn't need much explanation. If you have any doubts, read over the questions in [figure 43](#) and design the template on your own.

From here it is straightforward to define the function. The key is to combine the examples with the template and to answer the questions from [figure 44](#). Following those, you start with the base case of an empty list of shots and, from the examples, you know that the expected answer is `BACKGROUND`. Next you formulate what the template expressions in the second `cond` compute:

- `(first w)` extracts the first coordinate from the list;
- `(rest w)` is the rest of the coordinates; and

- according to the purpose statement of the function, `(to-image (rest w))` adds each shot on `(rest w)` to the background image.

In other words, `(to-image (rest w))` renders the rest of the list as an image and thus does almost all the work. What is missing from this image is the first shot, `(first w)`. If you now apply the purpose statement to these two expressions, you get the desired expression for the second `cond` clause:

```
| (place-image SHOT XSHOTS (first w) (to-image (rest w)))
```

The added icon is the standard image for a shot; the two coordinates are spelled out in the purpose statement; and the last argument to `place-image` is the image constructed from the rest of the list.

```
; physical constants
(define HEIGHT 80)
(define WIDTH 100)
(define XSHOTS (/ WIDTH 2))

; graphical constants
(define BACKGROUND (empty-scene WIDTH HEIGHT))
(define SHOT (triangle 3 "solid" "red"))

; A ShotWorld is List-of-numbers.
; interpretation the collection of shots fired and moving straight up

; ShotWorld -> ShotWorld
(define (main w0)
  (big-bang w0
    [on-tick tock]
    [on-key keyh]
    [to-draw to-image]))

; ShotWorld -> ShotWorld
; moves each shot up by one pixel
(define (tock w)
  (cond
    [(empty? w) '()]
    [else (cons (sub1 (first w)) (tock (rest w))))]))

; ShotWorld KeyEvent -> ShotWorld
; adds a shot to the world if the space bar was hit
(define (keyh w ke)
  (cond
    [(key=? ke " ") (cons HEIGHT w)]
    [else w]))

; ShotWorld -> Image
; adds each y on w at (MID,y) to the background image
(define (to-image w)
  (cond
    [(empty? w) BACKGROUND]
    [else (place-image SHOT XSHOTS (first w) (to-image (rest w))))]))
```

Figure 48: A list-based world program

[Figure 48](#) displays the complete function definition for `to-image` and indeed the rest of the program, too. The design of `tock` is just like the design of `to-image` and you should work through it for yourself. The signature of the `keyh` handler, though, poses one interesting question. It specifies that the handler consumes two inputs with non-trivial data definitions. On one hand, the `ShotWorld` is self-referential data definition. On the other hand, the definition for `KeyEvents` is a large enumeration. For now, we have you “guess” which of the two arguments should drive the development of the template; later we will study such cases in depth.

As far as a world program is concerned, a key handler such as `keyh` is about the key event that it consumes. Hence, we consider it the main argument and use its data definition to derive the template. Specifically, following the data definition for `KeyEvent` from [Enumerations](#) it dictates that the function needs a `cond` expression with numerous clauses like this:

```
(define (keyh w ke)
  (cond
    [(key=? ke "left") ...]
    [(key=? ke "right") ...]
    ...
    [(key=? ke " ") ...]
    ...
    [(key=? ke "a") ...]
    ...
    [(key=? ke "z") ...]))
```

Of course, just like for functions that consume all possible BSL values, a key handler usually does not need to inspect all possible cases. For our running problem, you specifically know that the key handler reacts only to the space bar and all others are ignored. So it is natural to collapse all of the `cond` clauses into an `else` clause except for the clause for `" "`.

Exercise 158. Equip the program in [figure 48](#) with tests and make sure it passes those. Explain what `main` does. Then run the program via `main`. ■

Exercise 159. Experiment whether the arbitrary decisions concerning constants are truly easy to change. For example, determine whether changing a single constant definition achieves the desired outcome:

- change the height of the canvas to 220 pixels;
- change the width of the canvas to 30 pixels;
- change the `x` location of the line of shots to “somewhere to the left of the middle;”
- change the background to a green rectangle; and
- change the rendering of shots to a red elongated rectangle.

Also check whether it is possible to double the size of the shot without changing anything else, change its color to black, or change its form to `"outline"`. ■

Exercise 160. If you run `main`, press the space bar (fire a shot), and wait for a good amount of time, the shot disappears from the canvas. When you shut down the world canvas, however, the result is a world that still contains this invisible shot.

Design an alternative `tock` function, which not just moves shots one pixel per clock tick

but also eliminates those whose coordinates places them above the canvas. **Hint** You may wish to consider the design of an auxiliary function for the recursive `cond` clause. ■

Exercise 161. Turn the exercise of [exercise 155](#) into a world program. Its main function, dubbed `riot`, consumes how many balloons the students want to throw; its visualization shows one balloon dropping after another at a rate of one per second. The function produces the list of `Posns` where the balloons hit.

Hints (1) Here is one possible data representation:

```
(define-struct pair [balloon# lob])
; A Pair is a structure (make-pair N List-of-posns)
; A List-of-posns is one of:
; - '()
; - (cons Posn List-of-posns)
; interpretation (make-pair n lob) means n
; balloons must yet be thrown and the thrown balloons landed at lob
```

(2) A `big-bang` expression is really just an expression. It is legitimate to nest it within an expression.

(3) Recall that `random` creates random numbers. ■

10.6 A Note on Lists and Sets

This book relies on your intuitive understanding of *sets* as collections of BSL values. [The Universe of Data](#) specifically says that a data definition introduces a name for a set of BSL values. There is one question that this book consistently asks about sets, and it is whether some element is in some given set. For example, `4` is in `Number`, while "four" is not. The book also shows how to use a data definition to check whether some value is a member of some named set and how to use some of the data definitions to generate sample elements of sets; but, these two procedures are about data definitions not sets per se.

At the same time, lists are representations of collections of values. Hence you might be wondering what the difference between a list and a set is or whether this is a needless distinction. If so, this section is for you.

Right now the primary difference between sets and lists is that the former is a concept we use to discuss steps in the design of code and the latter is one of many forms of data in BSL, our chosen programming language. The two ideas live at rather different levels in our conversations. However, given that a data definition introduces a data representation of actual information inside of BSL and given that sets are collections of information, you may now ask yourself how sets are represented inside of BSL as data.

Most full-fledged languages directly support data representations of both lists and sets.

While lists have a special status in BSL, sets don't, but at the same time sets somewhat resemble lists. The key difference is the kind of functions a program normally uses with either form of data. BSL provides several basic constants and functions for lists—say, `empty`, `empty?`, `cons`, `cons?`, `first`, `rest`—and some functions that you could define yourself—for example, `member?`, `length`, `remove`, `reverse`, and so on. Here is an example of a function you can define but does not come with BSL

```
; List-of-string String -> N
; determine how often s occurs in los
(define (count los s)
  0)
```

Stop! Finish the design of this function.

Let's proceed in straightforward and possibly naive manner and say sets are basically lists. And, to simplify further, let's focus on lists of numbers in this section. If we now accept that it merely matters whether a number is a part of a set or not, it is almost immediately clear that we can use lists in **two** different ways to represent sets.

<pre>; A Son.L is one of: ; - empty ; - (cons Number Son.L) ; ; Son is used when it applies ; to both Son.L and Son.R</pre>	<pre>; A Son.R is one of: ; - empty ; - (cons Number Son.R) ; ; Constraint If s is a Son.R, ; no number occurs twice in s.</pre>
---	--

Figure 49: Two data representations for sets

Figure 49 displays the two data definitions. Both basically say that a set is represented as a list of numbers. The difference is that the definition on the right comes with the constraint that no number may occur more than once on the list. After all, the key question we ask about a set is whether some number is in the set or not, and whether it is in a set once, twice or three times makes no difference.

Regardless of which definition you choose, you can already define two important notions:

```
; Son
(define es '())

; Number Son -> Son
; is x in s
(define (in? x s)
  (member? x s))
```

The first one is the **empty set**, which in both cases is represented by the empty list. The second one is a membership test.

One way to build larger sets is to use `cons` and the above definitions. Say we wish to build a representation of the set that contains 1, 2, and 3. Here is one such representation:

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

And it works for both data representations. But, is

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

really not a representation of the same set? Or how about

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

The answer has to be affirmative as long as the primary concern is whether a number is in a set or not. Still, while the order of `cons` cannot matter, the constraint in the right-hand data definition rules out the last list as a `Son.R` because it contains 1 twice.

```

; Number Son.L -> Son.L
; remove x from s
(define s1.L
  (cons 1 (cons 1 '())))

(check-expect (set-.L 1 s1.L) es)
(define (set-.L x s)
  (remove-all x s))

; Number Son.R -> Son.R
; remove x from s
(define s1.R
  (cons 1 '()))

(check-expect (set-.R 1 s1.R) es)
(define (set-.R x s)
  (remove x s))

```

Figure 50: Functions for the two data representations of sets

The difference between the two data definitions shows up when we design functions on these set representations. Say we want a function that removes a number from a set. Here is a wish list entry that applies to both representations:

```

; Number Son -> Son
; subtract x from s
(define (set- x s)
  s)

```

The purpose statement uses the word “subtract” because this is what logicians and mathematicians use when they work with sets.

[Figure 50](#) shows the results. The two columns differ in two points:

1. The test on the left uses a list that contains `1` twice, while the one on the right represents the same set with a single `cons`.
2. Because of these differences, the `set-` on the left must use `remove-all`, while the one on the right gets away with `remove`.

Stop! Copy the code into the DrRacket definitions area and make sure the tests pass. Then read on and experiment with the code as you do.

An unappealing aspect of [figure 50](#) is that the tests use `es`, a plain list as the expected result. This problem may seem minor at first glance. Consider the following example, however:

```
(set- 1 set123)
```

where `set123` represents the set containing `1`, `2`, and `3` in one of these two ways:

```
(define set123-version1
  (cons 1 (cons 2 (cons 3 '()))))
```

```
(define set123-version2
  (cons 1 (cons 3 (cons 2 '()))))
```

Regardless of which representation we choose, the result of `(set- 1 set123)` is one of these two lists:

```
(define set23-version1
  (cons 2 (cons 3 '())))
```

```
(define set23-version2
  (cons 3 (cons 2 '())))
```

But, we cannot predict which of those two `set-` produces.

For the simple case of two alternatives, it is possible to use the `check-member-of` testing

facility as follows:

```
(check-member-of (set-.v1 1 set123.v1)
                  set23-version1
                  set23-version2)
```

If the expected set contains three elements, there are six possible variations—not including representations with repetitions, which the left-hand data definition allows.

Fixing this problem calls for the combination of two ideas. First, recall that `set-` is really about ensuring that the given element does not occur in the result. It is an idea that our way of turning the examples into tests does not bring across. Second, with BSL's `check-satisfied` testing facility, it is possible to state just this idea.

Intermezzo: BSL briefly mentions `check-satisfied` but, in a nutshell, the facility determines whether an expression satisfies a certain property. A property is a function from values to `Boolean`. In our specific case, we wish to state that `1` is not a member of some set:

```
; Son -> Boolean
; #true if 1 a member of s; #false otherwise
(define (not-member-1? s)
  (not (in? 1 s)))
```

Using `not-member-1?`, we can formulate the test case as follows:

```
(check-satisfied (set- 1 set123) not-member-1?)
```

and this variant clearly states what the function is supposed to accomplish. Better yet, this formulation simply does not depend on how the input or output set is represented.

In sum, lists and sets are related in that both are about collections of values but they also differ strongly:

property	lists	sets
membership	one among many	critical
ordering	critical	irrelevant
# of occurrences of element	sensible	irrelevant
size	finite but arbitrary	finite or infinite

The last row in this table presents a new idea, though an obvious one, too. Many of the sets mentioned in this book are infinitely large, for example, `Number`, `String`, and also `List-of-strings`. In contrast, a list is **always** finite though it may contain an arbitrarily large number of items.

In sum, this section explains the essential differences between sets and lists and how to represent finite sets with finite lists in two different ways. BSL is not expressive enough to represent infinite sets; [exercise 304](#) introduces a completely different representation of sets, a representation that can cope with infinite sets, too. The question how actual programming languages represent sets is beyond the scope of this book, however.

Exercise 162. Design the functions `set+.L` and `set+.R`, which create a set by adding a number `x` to some given set `s` for the left-hand and right-hand data definition, respectively.

Lists are a versatile form of data that come with almost all languages now. Programmers have used them to build large applications, artificial intelligences, distributed systems, and more. This chapter illustrates some basic ideas from this world, including functions that create lists, data representations that call for structures inside of lists, and representing text files as lists.

11.1 Functions that Produce Lists

Here is a function for determining the wage of an hourly employee:

```
; Number -> Number
; computes the wage for h hours of work
(define (wage h)
  (* 12 h))
```

It consumes the number of hours worked and produces the wage. A company that wishes to use payroll software isn't interested in this function, however. It wants a function that computes the wages for all of its employees.

Call this new function `wage*`. Its task is to process all employee work hours and to determine the wages due to each of them. For simplicity, let us assume that the input is a list of numbers, each representing the number of hours that one employee worked, and that the expected result is a list of the weekly wages earned, also represented with a list of numbers.

Since we already have a data definition for the inputs and outputs, we can immediately move to the second design step:

```
; List-of-numbers -> List-of-numbers
; computes the weekly wages for all given weekly hours
(define (wage* alon)
  '())
```

Next you need some examples of inputs and the corresponding outputs. So you make up some short lists of numbers that represent weekly work hours:

given	expected
'()	'()
(cons 28 '())	(cons 336 '())
(cons 40 (cons 28 '())))	(cons 480 (cons 336 '())))

In order to compute the output, you determine the weekly wage for each number on the given input list. For the first example, there are no numbers on the input list so the output is `'()`. Make sure you understand why the second and third expected output is what you want.

Given that `wage*` consumes the same kind of data as several other functions from [Lists](#) and given that a template depends only on the shape of the data definition, you can reuse these template:

```
(define (wage* alon)
  (cond
    [(empty? alon) ...]
    [else (... (first alon) ... (wage* (rest alon)) ...)]))
```

In case you want to practice the development of templates, use the questions from

figure 43.

It is now time for the most creative design step. Following the design recipe, we consider each `cond`-line of the template in isolation. For the non-recursive case, `(empty? alon)` is true, meaning the input is `'()`. The examples from above specify the desired answer, `'()`, and so we are done.

In the second case, the design questions tell us to state what each expression of the template computes:

- `(first alon)` yields the first number on `alon`, which is the first number of hours worked;
- `(rest alon)` is the rest of the given list; and
- `(wage* (rest alon))` says that the rest is processed by the very function we are defining. As always we use its signature and its purpose statement to figure out the result of this expression. The signature tells us that it is a list of numbers, and the purpose statement explains that this list represents the list of wages for its input, which is the rest of the list of hours.

The key is to rely on these facts when you formulate the expression that computes the result in this case—even though the function is not yet defined.

Since we already have the list of wages for all but the first item of `alon`, the function must perform two computations to produce the expected output for the **entire** `alon`: compute the weekly wage for `(first alon)` and construct the list that represents all weekly wages for `alon` from the first weekly wage and the result of the recursion. For the first part, we reuse `wage`. For the second, we `cons` the two pieces of information together into one list:

```
| (cons (wage (first alon)) (wage* (rest alon)))
```

And with that, the definition is complete: see [figure 51](#).

```
; List-of-numbers -> List-of-numbers
; computes the weekly wages for all given weekly hours
(define (wage* alon)
  (cond
    [(empty? alon) '()]
    [else (cons (wage (first alon)) (wage* (rest alon))))]))

; Number -> Number
; computes the wage for h hours of work
(define (wage h)
  (* 12 h))
```

Figure 51: Computing the wages of all employees

Exercise 163. Translate the examples into tests and make sure they all succeed. Then change the function in [figure 51](#) so that everyone gets \$14 per hour. Now revise the entire program so that changing the wage for everyone is a single change to the **entire** program and not several. ■

Exercise 164. No employee could possibly work more than 100 hours per week. To protect the company against fraud, the function should check that no item of the input list of `wage*` exceeds 100. If one of them does, the function should immediately signal an error. How do we have to change the function in [figure 51](#) if we want to perform this basic

reality check? ■

Exercise 165. Design `convertFC`. The function converts a list of Fahrenheit measurements to a list of Celsius measurements. ■

Exercise 166. Design the function `convert-euro`, which converts a list of US\$ amounts into a list of € amounts. Look up the current exchange rate on the web.

Show the products of the various steps in the design recipe. If you are stuck, show someone how far you got according to the design recipe. The recipe isn't just a design tool for you to use; it is also a diagnosis system so that others can help you help yourself.

Generalize `convert-euro` to the function `convert-euro*`, which consumes an exchange rate and a list of US\$ amounts and converts the latter into a list of € amounts.

Exercise 167. Design the function `subst-robot`, which consumes a list of toy descriptions (one-word strings) and replaces all occurrences of "robot" with "r2d2"; all other descriptions remain the same.

Generalize `subst-robot` to the function `substitute`. The new function consumes two strings, called `new` and `old`, and a list of strings. It produces a new list of strings by substituting all occurrences of `old` with `new`.

11.2 Structures in Lists

Representing a work week as a number is a bad choice because the printing of a paycheck requires more information than hours worked per week. Also, not all employees earn the same amount per hour. Fortunately a list may contain items other than atomic values; indeed, lists may contain whatever values we want, especially structures.

Our running example calls for just such a data representation. Instead of numbers, we use structures that contain information about the employee, including the work hours:

```
(define-struct work [employee rate hours])
; Work is a structure: (make-work String Number Number).
; interpretation (make-work n r h) combines the name (n)
; with the pay rate (r) and the number of hours (h) worked.
```

While this representation is still simplistic, it is just enough of an additional challenge because it forces us to formulate a data definition for lists that contain structures:

```
; Low (list of works) is one of:
; - '()
; - (cons Work Low)
; interpretation an instance of Low represents the hours worked
; of a number of employees
```

Here are three elements of `Low`:

```
'()
(cons (make-work "Robby" 11.95 39)
      '())
(cons (make-work "Matthew" 12.95 45)
      (cons (make-work "Robby" 11.95 39)
```

' ())

Use the data definition to explain why these pieces of data belong to [Low](#).

Stop! Also use the data definition to generate two more examples.

Now that you know that the definition of [Low](#) makes sense, it is time to re-design the function `wage*` so that it consumes elements of [Low](#) not just lists of numbers:

```
; Low -> List-of-numbers
; computes the weekly wages for all given weekly work records
(define (wage*.v2 an-low)
  '())
```

The suffix “`.v2`” at the end of the function name informs every reader of the code that this is a second, revised version of the function. In this case, the revision starts with a new signature and an adapted purpose statement. The header is the same as above.

When you work on real-world projects, you won't use such suffixes; instead you will use a mechanism for managing different versions of the same code.

The third step of the design recipe is to work through an example. Let us start with the second list above. It contains one work record, namely, `(make-work "Robby" 11.95 39)`. Its interpretation is that “[Robby](#)” worked for 39 hours and that he is paid at the rate of \$11.95 per hour. Hence his wage for the week is \$466.05, i.e., `(* 11.95 39)`. The desired result for `wage*.v2` is therefore `(cons 466.05 '())`. Naturally, if the input list contained two work records, we would perform this kind of computation twice, and the result would be a list of two numbers. Before you read on, determine the expected result for the third data example above.

Note on Numbers Keep in mind that BSL—unlike most other programming languages—understands decimal numbers just like you do, namely, as exact fractions. A language such as Java, for example, would produce 466.0499999999995 for the expected wage of the first work record. Since you cannot predict when operations on decimal numbers behave in this strange way, you are better off writing down such examples as

```
(check-expect
  (wage*.v2 (cons (make-work "Robby" 11.95 39) '()))
  (cons (* 11.95 39) '()))
```

just to prepare yourself for other programming languages. Then again, writing down the example in this style also means you have really figured out how to compute the wage.

End

From here we move on to the development of the template. If you use the template questions, you quickly get this much:

```
(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) ...]
    [(cons? an-low)
     (... (first an-low) ... (wage*.v2 (rest an-low)) ...)]))
```

because the data definition consists of two clauses, because it introduces `'()` in the first clause and `consed` structures in the second, and so on. But, you also realize that you know

even more about the input than this template expresses. For example, you know that `(first an-low)` extracts a structure of three fields from the given list. This seems to suggest the addition of three more expressions to the template:

```
(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) ...]
    [(cons? an-low)
      (... (first an-low) ... (work-employee (first an-low)) ...
            (work-rate (first an-low)) ... (work-hours (first an-low))
            (wage*.v2 (rest an-low)) ...)])))
```

This way you would remember that the structure contains potentially interesting data.

We use a different strategy here. Specifically, we suggest **to create and to refer to a separate function template** whenever you are developing a template for a data definition that refers to other data definitions:

```
(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) ...]
    [(cons? an-low)
      (... (for-work (first an-low))
            ... (wage*.v2 (rest an-low)) ...)])))

; Work -> ???
; a template for functions that process work structures
(define (for-work w)
  (... (work-employee w) ... (work-rate w) ... (work-hours w) ...))
```

Splitting the templates leads to a natural partition of work into functions and among functions; none of them grow too large, and all of them relate to a specific data definition.

Finally, you are ready to program. As always you start with the simple-looking case, which is the first `cond` line here. If `wage*.v2` is applied to `'()`, you expect `'()` back and that settles it. Next you move on to the second line and remind yourself of what these expressions compute:

1. `(first an-low)` extracts the first work structure from the list;
2. `(for-work ...)` says that you wish to design a function that processes work structures;
3. `(rest an-low)` extracts the rest of the given list; and
4. according to the purpose statement, `(wage*.v2 (rest an-low))` determines the list of wages for all the work records other than the first one.

If you are stuck here, use the examples to illustrate what these reminders mean.

If you understand it all, you see that it is enough to `cons` the two expressions together:

```
... (cons (for-work (first an-low))
          (wage*.v2 (rest an-low))) ...
```

assuming that `for-work` computes the wage for the first work record. In short, you have finished the function by adding another entry to your wish list of functions.

Since `for-work` is a name that just serves as a stand-in and since it is a bad name for this function, let us call the function `wage.v2` and write down its complete wish list entry:

```
; Work -> Number
; computes the wage for the given work record w
(define (wage.v2 w)
  0)
```

This design of this kind of function is extensively covered in [Fixed-Size Data](#) and thus doesn't need any additional explanation here. [Figure 52](#) shows the final result of developing `wage` and `wage*.v2`.

```
; Low -> List-of-numbers
; computes the weekly wages for all given weekly work records

(check-expect (wage*.v2 (cons (make-work "Robby" 11.95 39) '()))
              (cons (* 11.95 39) '()))

(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) '()]
    [(cons? an-low) (cons (wage.v2 (first an-low))
                          (wage*.v2 (rest an-low)))]))

; Work -> Number
; computes the wage for the given work record w
(define (wage.v2 w)
  (* (work-rate w) (work-hours w)))
```

Figure 52: Computing the wages from work records

Exercise 168. The `wage*.v2` function consumes a list of work records and produces a list of numbers. Of course, functions may also produce lists of structures.

Develop a data representation for pay checks. Assume that a pay check contains two pieces of information: the name of the employee and an amount. Then design the function `wage*.v3`. It consumes a list of work records and computes a list of (representations of) pay checks from it, one per work record.

In reality, a pay check also contains an employee number. Develop a data representation for employee information and change the data definition for work records so that it uses employee information and not just a string for the employee's name. Also change your data representation of pay checks so that it contains an employee's name and number, too. Finally, design `wage*.v4`, a function that maps lists of revised work records to lists of revised pay checks.

Note on Iterative Refinement This exercise demonstrates the *iterative refinement* of a task. Instead of using data representations that include all relevant information, we started from simplistic representation of pay checks and gradually made the representation realistic. For this simple program, refinement is overkill; later we will encounter situations where iterative refinement is not just an option but a necessity.

Exercise 169. Design the function `sum`, which consumes a list of `Posns` and produces the sum of all of its x-coordinates.

Exercise 170. Design the function `translate`. It consumes and produces lists of `Posns`.

For each `(make-posn x y)` in the former, the latter contains `(make-posn x (+ y 1))`.

—We borrow the word “translate” from geometry, where the movement of a point by a constant distance along a straight line is called a *translation*.

Exercise 171. Design the function `legal`. Like `translate` from [exercise 170](#) the function consumes and produces a list of `Posns`. The result contains all those `Posns` whose x-coordinates are between 0 and 100 and whose y-coordinates are between 0 and 200.

Exercise 172. Here is one way to represent a phone number:

```
(define-struct phone [area switch four])
; A Phone is a structure:
; (make-phone Three Three Four)
; A Three is between 100 and 999.
; A Four is between 1000 and 9999.
```

Design the function `replace`. It consumes a list of `Phones` and produces one. It replaces all occurrence of area code `713` with `281`.

11.3 Lists in Lists, Files

[Functions And Programs](#) introduces `read-file`, a function for reading an entire text file as a string. In other words, the creator of `read-file` chose to represent text files as strings, and the function creates the data representation for specific files (specified by a name). Text files aren't plain long texts or strings, however. They are organized into lines and words, rows and cells, and many other ways. In short, representing the content of a file as a plain string might work on rare occasions but is usually a bad choice.

Use `(require 2htdp/batch-io)`.

For concreteness, take a look at this sample files, dubbed "`ttt.txt`":

`ttt.txt`

TTT

Put up in a place
where it's easy to see
the cryptic admonishment
T.T.T.

When you feel how depressingly
slowly you climb,
it's well to remember that
Things Take Time.

Piet Hein

It contains a poem by Piet Hein, and it consists of many lines and words. When you use the program

```
(read-file "ttt.txt")
```

to turn this file into a BSL string, you get this:

```
"TTT\n \nPut up in a place\nwhere ..."
```

where the "`\n`" inside the string indicates line breaks. (The "... aren't really a part of the result as you probably guessed.)

While it is indeed possible to break apart this string with primitive operations on strings, e.g., `explode`, most programming languages—including BSL—support many different representations of files and functions that create such representations from existing files:

- One way to represent this file is as a list of lines, where each line is represented as one string:

```
(cons "TTT"
  (cons ""
    (cons "Put up in a place"
      (cons ...
        '()))))
```

Here the second item of the list is the empty string because the file contains an empty line.

- Another way is to use a list of all possible words, again each word represented to a string:

```
(cons "TTT"
  (cons "Put"
    (cons "up"
      (cons "in"
        (cons ...
          '())))))
```

Note how the empty second line disappears with this representation. After all, there are no words on the empty line.

- And a third representation mixes the first two with a list of list of words:

```
(cons (cons "TTT" '())
  (cons '()
    (cons (cons "Put" (cons "up" (cons "in" (cons ...
      '())))))
    (cons ...
      '())))))
```

The advantage of this representation over the second one is that it preserves the organization of the file, including the emptiness of the second line. Of course, the price is that all of a sudden lists contain lists.

While the idea of list-containing lists may sound frightening at first, you shouldn't worry. The design recipe helps even with such complicated forms of data.

Before we get started, take a look at [figure 53](#). It introduces a number of useful file reading functions that come with BSL. They are not comprehensive and there are many other ways of dealing with text from files, and you will need to know a lot more to deal with all possible text files. For our purposes here—teaching and learning the principles of systematic program design—they suffice, and they empower you to design reasonably interesting programs.

```
; String -> String
; produces the content of file f as a string
(define (read-file f) ...)
```

```

; String -> List-of-string
; produces the content of file f as a list of strings,
; one per line
(define (read-lines f) ...)

; String -> List-of-string
; produces the content of file f as a list of strings,
; one per word
(define (read-words f) ...)

; String -> List-of-list-of-string
; produces the content of file f as a list of list of
; strings, one list per line and one string per word
(define (read-words/line f) ...)

; All functions that read a file consume the name of a file
; as a String argument. They assume the specified file
; exists in the same folder as the program; if not they
; signal an error.

```

Figure 53: Reading files

One problem with [figure 53](#) is that they use the names of two data definitions that do not exist yet, including one involving list-containing lists. As always, we start with a data definition, but this time, we leave this task to you. Hence, before you read on, solve the following exercises. The solutions are needed to make complete sense out of the figure, and without working through the solutions, you cannot really understand the rest of this section.

Exercise 173. You know what the data definition for [List-of-strings](#) looks like. Spell it out. Make sure that you can represent Piet Hein's poem as an instance of the definition where each line is represented as a string and another one where each word is a string. Use [read-lines](#) and [read-words](#) to confirm your representation choices.

Next develop the data definition for [List-of-list-of-strings](#). Again, represent Piet Hein's poem as an instance of the definition where each line is represented as a list of strings, one per word, and the entire poem is a list of such line representations. You may use [read-words/line](#) to confirm your choice.

As you probably know, operating systems come with programs that measure various statistics of files. Some count the number of lines, others count the number of words in a file. A third may determine how many words appear per line. Let us start with the latter to illustrate how the design recipe helps with the design of complex functions.

The first step is to ensure that we have all the necessary data definitions. If you solved the above exercise, you have a data definition for all possible inputs of the desired function, and the preceding section defines [List-of-numbers](#), which describes all possible inputs. To keep things short, we use [LLS](#) to refer to the class of lists of lists of strings, and use it to write down the header material for the desired function:

```

; LLS -> List-of-numbers
; determines the number of words on each line

(define (words-on-line lls)
  '())

```

We name the functions `words-on-line`, because it is appropriate and captures the purpose statement in one phrase.

What is really needed though is a set of examples. Here are some **data examples**:

```
(define line0 (cons "hello" (cons "world" '())))
(define line1 '())

(define lls0 '())
(define lls1 (cons line0 (cons line1 '())))
```

The first two definitions introduce two examples of lines: one contains two words, the other contains none. The last two definitions show how to construct instances of `LLS` from these line examples. Determine what the expected result is when the function is given these two examples.

Once you have data examples, it is easy to formulate functional examples; just imagine applying the function to each of the data example. When apply `words-on-line` to `lls0`, you should get the empty list back, because there are no lines. When you apply `words-on-line` to `lls1`, you should get a list of two numbers back, because there are two lines. The two numbers are `2` and `0`, respectively, given that the two lines in `lls1` contain two and no words each.

Here is how you translate all this into test cases:

```
(check-expect (words-on-line lls0) '())
(check-expect (words-on-line lls1) (cons 2 (cons 0 '()))))
```

You can do this at the end of the second step or for the last step. Doing it now, means you have a complete program, though running it just fails some of the test cases.

The development of the template is the interesting step for this sample problem. By answering the template questions from [figure 43](#), you get the usual list-processing template immediately:

```
(define (words-on-line lls)
  (cond
    [(empty? lls) ...]
    [else
      (... (first lls) ; a list of strings
           ... (words-on-line (rest lls)) ...)]))
```

As in the preceding section, we know that the expression `(first lls)` extracts a [List-of-strings](#), which has a complex organization, too. The temptation is to insert a nested template to express this knowledge, but as you should recall, the better idea is to develop a second auxiliary template and to change the first line in the second condition so that it refers to this auxiliary template.

Since this auxiliary template is for a function that consumes a list, the template looks nearly identical to the previous one:

```
(define (line-processor ln)
  (cond
    [(empty? ln) ...]
    [else
      (... (first ln) ; a string
           ... (line-processor (rest ln)) ...)]))
```

The important differences are that (`first ln`) extracts a string from the list and that we consider strings as atomic values. With this template in hand, we can change the first line of the second case in `words-on-line` to

```
... (line-processor (first lls)) ...
```

which reminds us for the fifth step that the definition for `words-on-line` may demand the design of an auxiliary function.

Now it is time to program. As always, we use the questions from [figure 44](#) to guide this step. The first case, concerning empty lists of lines, is the easy case. Our examples tell us that the answer in this case is '`()`', i.e., the empty list of numbers. The second, concerning `constructed` lists, case contains several expressions and we start with a reminder of what they compute:

- (`first lls`) extracts the first line from the non-empty list of (represented) lines;
- (`line-processor (first lls)`) suggests that we may wish to design an auxiliary function to process this line;
- (`rest lls`) is the rest of the list of line;
- (`(words-on-line (rest lls))`) computes a list of words per line for the rest of the list. How do we know this? We promised just that with the signature and the purpose statement for `words-on-line`.

Assuming we can design an auxiliary function that consumes a line and counts the words on one line—let us call this function `words#`—it is easy to complete the second condition:

```
(cons (words# (first lls)) (words-on-line (rest lls)))
```

This expression `conses` the number of words on the first line of `lls` onto a list of numbers that represents the number of words on the remainder of lines of `lls`.

It remains to design the `words#` function. Its template is dubbed `line-processor` and its purpose is to count the number of words on a line, which is just a list of strings. So here is the wish-list entry:

```
; List-of-strings -> Number
; counts the number of words on los
(define (words# los) 0)
```

At this point, you may recall the example used to illustrate the design recipe for self-referential data in [Designing with Self-Referential Data Definitions](#). The function is called `how-many`, and it too counts the number of strings on a list of strings. Even though the inputs for `how-many` is supposed to represent a list of names, this difference simply doesn't matter; as long as it correctly counts the number of strings on a list of strings, `how-many` solves our problem.

Since it is always good to reuse existing solutions, one way to define `words#` is

```
(define (words# los)
  (how-many los))
```

In reality, however, programming languages come with functions that solve such problems already. BSL calls this function `length`, and it counts the number of values on any list of values, no matter what the values are.

```

; A LLS is one of:
; - '()
; - (cons Los LLS)
; interpretation a list of lines, each line is a list of strings

(define line0 (cons "hello" (cons "world" '())))
(define line1 '())

(define lls0 '())
(define lls1 (cons line0 (cons line1 '())))

; LLS -> List-of-numbers
; determines the number of words on each line

(check-expect (words-on-line lls0) '())
(check-expect (words-on-line lls1) (cons 2 (cons 0 '())))

(define (words-on-line lls)
  (cond
    [(empty? lls) '()]
    [else (cons (length (first lls))
                 (words-on-line (rest lls)))])))

```

Figure 54: Counting the words on a line

Figure 54 summarizes the full design for our sample problem. The figure includes two test cases. Also, instead of using the separate function `words#`, the definition of `words-on-line` simply calls the `length` function that comes with BSL. Experiment with the definition in DrRacket and make sure that the two test cases cover the entire function definition.

You may wish to look over the rest of functions that come with BSL. Some may look obscure now, but they may just be useful in one of the upcoming problems. Then again, using such functions saves only your time. You just may wish to design them from scratch to practice your design skills or to fill time.

With one small step, you can now design your first file utility:

```

; String -> List-of-numbers
; counts the number of words on each line in the given file
(define (file-statistic file-name)
  (words-on-line
   (read-words/line file-name)))

```

The function composes the library function with the just designed `words-on-line` function. The former reads a file as a `List-of-list-of-strings` and hands this value to the latter.

This idea of composing a built-in function with a newly designed function is common. Naturally, people don't design functions randomly and expect to find something in the chosen programming language to complement their design. Instead, program designers plan ahead and design the function **to the output** that available functions deliver. More generally still and as mentioned above, it is common to think about a solution as a composition of two computations and to develop an appropriate data collection with which to communicate the result of one computation to the second one, where each computation is each implemented with a function.

Exercise 174. Design the function `collapse`, which converts a list of lines into a string. The strings should be separated by blank spaces (" ") . The lines should be separated with a newline ("\n").

Challenge: When you are finished with the design of the program, use it like this:

```
(write-file "ttt.dat" (collapse (read-words/line "ttt.txt")))
```

The two files "ttt.dat" and "ttt.txt" should be identical but may not due to extra white space. Remove all extraneous white spaces in your version of the Piet Hein poem.

Exercise 175. Design a program that removes all articles from a text file. The program consumes the name `n` of a file, reads the file, removes the articles, and writes the result out to a file whose name is the result of concatenating "no-articles-" with `n`. For this exercise, an article is one of the following three words: "a", "an", and "the".

Use `read-words/line` so that the transformation retains the organization of the original text into lines and words. When the program is designed, run it on the Piet Hein poem.

Exercise 176. Design a program that encodes text files numerically. Each letter in a word should be encoded as a numeric three-letter string with a value between 0 and 256. Here is our encoding function for single letters:

```
; 1String -> String
; converts the given 1String into a three-letter numeric String

(check-expect (encode-letter "\t") (string-append "00" (code1 "\t")))
(check-expect (encode-letter "a") (string-append "0" (code1 "a")))
(check-expect (encode-letter "z") (code1 "z"))

(define (encode-letter s)
  (cond
    [(< (string->int s) 10) (string-append "00" (code1 s))]
    [(< (string->int s) 100) (string-append "0" (code1 s))]
    [else (code1 s)]))

; 1String -> String
; convert the given 1String into a String

(check-expect (code1 "z") "122")

(define (code1 c)
  (number->string (string->int c)))
```

Before you start, explain these functions.

Hints Use `read-words/line` to preserve the organization of the file into lines and words. Also recall how a string can be converted into a list of `1Strings`.

Exercise 177. Design a BSL program that simulates the Unix command `wc`. The purpose of the command is to count the number of `1Strings`, words, and lines in a given file. That is, the command consumes the name of a file and produces a value that consists of three numbers.

Exercise 178. Mathematics teachers may have introduced you to matrix calculations by now. Numeric programs deal with those, too. In principle, matrix just means rectangle of numbers. Here is one possible data representation for matrices:

```

; A Matrix is one of:
; - (cons Row '())
; - (cons Row Matrix)
; constraint all rows in matrix are of the same length

; An Row is one of:
; - '()
; - (cons Number Row)

```

Note the constraints on matrices. Study the data definition and translate the two-by-two matrix consisting of the numbers 11, 12, 21, 22 into this data representation. Stop, don't read on until you have figured out the data examples.

Here is the solution for the five-second puzzle:

```

(define row1 (cons 11 (cons 12 '())))
(define row2 (cons 21 (cons 22 '())))
(define mat1 (cons row1 (cons row2 '())))

```

If you didn't create it yourself, study it now.

The following function implements the important mathematical operation of transposing the entries in a matrix. To transpose means to mirror the entries along the diagonal, that is, the line from the top-left to the bottom-right. Again, stop! Transpose `mat1` by hand, then read on:

```

; Matrix -> Matrix
; transpose the items on the given matrix along the diagonal

(define wor1 (cons 11 (cons 21 '())))
(define wor2 (cons 12 (cons 22 '())))
(define tam1 (cons wor1 (cons wor2 '())))

(check-expect (transpose mat1) tam1)

(define (transpose lln)
  (cond
    [(empty? (first lln)) '()]
    [else (cons (first* lln) (transpose (rest* lln)))]))

```

Why does `transpose` ask `(empty? (first lln))`?

The definition assumes two auxiliary functions:

- `first*`, which consumes a matrix and produces the first column as a list of numbers;
- `rest*`, which consumes a matrix and removes the first column. The result is a matrix.

Even though you lack definitions for these functions, you should be able to understand how `transpose` works. You should also understand that you **cannot** design this function with the design recipes you have seen so far. Explain why.

Design the two “wish list” functions. Then complete the design of the `transpose` with some test cases.

11.4 A Graphical Editor, Revisited

[A Graphical Editor](#) is about the design of an interactive graphical one-line editor. It suggests two different ways to represent the state of the editor and urges you to explore both: a structure that contains pair of strings or a structure that combines a string with an index to a current position (see [exercise 88](#)).

A third alternative is to use a structure type that combines two lists of [1Strings](#):

```
(define-struct editor [pre post])
; An Editor is (make-editor Lo1S Lo1S)
; An Lo1S is one of:
; - empty
; - (cons 1String Lo1S)
```

Before you wonder why, let us make up two data examples:

```
(define good
  (cons "g" (cons "o" (cons "o" (cons "d" '())))))
(define all
  (cons "a" (cons "l" (cons "l" '()))))
(define lla
  (cons "l" (cons "l" (cons "a" '()))))

; data example 1:
(make-editor all good)

; data example 2:
(make-editor lla good)
```

The two examples demonstrate how important it is to write down an interpretation. While the two fields of an editor clearly represent the letters to the left and right of the cursor, the two examples demonstrate that there are at least two ways to interpret the structure types:

1. `(make-editor pre post)` could mean the letters in `pre` precede the cursor and those in `post` succeed it and that the combined text is

```
| (string-append (implode pre) (implode post))
```

Recall that `implode` turns a list of [1Strings](#) into a [String](#).

2. `(make-editor pre post)` could equally well mean that the letters in `pre` precede the cursor in `reverse` order. If so, we obtain the text in the displayed editor like this:

```
| (string-append (implode (rev pre)) (implode post))
```

The function `rev` must consume a list of [1Strings](#) and produce their reverse.

Even without a complete definition for `rev` you can imagine how it works. Use this understanding to make sure you understand that translating the first data example into information according to the first interpretation and treating the second data example according to the second interpretation yields the same editor display:

Both interpretations are fine choices, but it turns out that using the second one greatly simplifies the design of the program. The rest of this section demonstrates this point, illustrating the use of lists inside of structures at the same time. To appreciate the lesson properly, you should have solved the exercises in [A Graphical Editor](#).

Let us start with the design of `rev`, since we clearly need this function to make sense out of the data definition. Its header material is straightforward:

```
; Lols -> Lols
; produces a reverse version of the given list

(check-expect
  (rev (cons "a" (cons "b" (cons "c" '()))))
  (cons "c" (cons "b" (cons "a" '()))))

(define (rev l)
  l)
```

For good measure, we have added one “obvious” example as a test case. You may want to add some extra examples just to make sure you understand what is needed.

The template for `rev` is the usual list template:

```
(define (rev l)
  (cond
    [(empty? l) ...]
    [else (... (first l) ...
                ... (rev (rest l)) ...))])
```

There are two cases, and the second case comes with several selector expressions and a self-referential one.

Filling in the template is easy for the first clause: the reverse version of the empty list is the empty list. For the second clause, we once again use the coding questions:

- `(first l)` is the first item on the list of `1Strings`;
- `(rest l)` is the rest of the list; and
- `(rev (rest l))` is the reverse of the rest of the list.

Stop! Try to finish the design of `rev` with these hints.

<code>l</code>	<code>(first l)</code>	<code>(rest l)</code>	<code>(rev (rest l))</code>	<code>(rev l)</code>
<code>(cons "a" '())</code>	"a"	'()	'()	<code>(cons "a" '())</code>
<code>(cons "a" (cons "b" (cons "c" '())))</code>	"a"	<code>(cons "b" (cons "c" '()))</code>	<code>(cons "c" (cons "b" '()))</code>	<code>(cons "c" (cons "b" (cons "a" '())))</code>

Figure 55: Tabulating for `rev`

If these hints leave you stuck, remember the table-based method of guessing the function that combines values. Figure 55 shows how `rev` deals with two examples according to the template: `(cons "a" '())` and `(cons "a" (cons "b" (cons "c" '())))`. The second example is particularly illustrative. A look at the next to last column shows that `(rev (rest l))` accomplishes most of the work by producing `(cons "c" (cons "b" '()))`. Since the desired result is `(cons "c" (cons "b" (cons "a" '())))` `rev` must somehow add "a" to the end of the result of the recursion. Indeed, if `(rev (rest l))` is

always the reverse of the rest of the list, it clearly suffices to add (`first l`) to its end. While we don't have a function that adds items to the end of a list, we can wish for it and use it to complete the function definition:

```
(define (rev l)
  (cond
    [(empty? l) '()]
    [else (add-at-end (rev (rest l)) (first l))]))
```

Here is the extended wish list entry for `add-at-end`:

```
; Lo1s 1String -> Lo1s
; creates a new list by adding s to the end of l

(check-expect
  (add-at-end (cons "c" (cons "b" '())) "a")
  (cons "c" (cons "b" (cons "a" '()))))

(define (add-at-end l s)
  l)
```

It is “extended” because it comes with an example formulated as a test case. The example is derived from the example for `rev`, and indeed, it is precisely the example that motivates the wish list entry. Make up an example where `add-at-end` consumes an empty list before you read on.

Since `add-at-end` is also a list-processing function, the template is just a renaming of the one you know so well now:

```
(define (add-at-end l s)
  (cond
    [(empty? l) ...]
    [else (... (first l) ...
               ... (add-at-end (rest l) s) ...)]))
```

To complete it into a full function definition, we proceed according to the recipe questions for step 5. Our first question is to formulate an answer for the “basic” case, i.e., the first case here. If you did work through the suggested exercise, you know that the result of

```
(add-at-end '() s)
```

is always (`cons s '()`). After all, the result must be a list and the list must contain the given `1String`.

The next two questions concern the “complex” or “self-referential” case. We know what the expressions in the second `cond` line compute: the first expression extracts the first `1String` from the given list and the second expression “creates a new list by adding `s` to the end of (`rest l`).” That is, the purpose statement dictates what the function must produce here. From here, it is clear that the function must add (`first l`) back to the result of the recursion:

```
(define (add-at-end l s)
  (cond
    [(empty? l) (cons s '())]
    [else (cons (first l) (add-at-end (rest l) s))]))
```

Run the tests-as-examples to reassure yourself that this function works and that therefore

`rev` works, too. Of course, you shouldn't be surprised to find out that BSL already provides a function that reverses any given list, including lists of `1String`s. And naturally, it is called `reverse`.

Exercise 179. Design the function `create-editor`. The function consumes two strings and produces an `Editor`. The first string is the text to the left of the cursor and the second string is the text to the right of the cursor. The rest of the section relies on this function.

At this point, you should have a complete understanding of our data representation for the graphical one-line editor. Following the design strategy for interactive programs from [Designing World Programs](#), you should define physical constants—the width and height of the editor, for example—and graphical constants—e.g., the cursor. Here are ours:

```
; constants
(define HEIGHT 20) ; the height of the editor
(define WIDTH 200) ; its width
(define FONT-SIZE 16) ; the font size
(define FONT-COLOR "black") ; the font color

; graphical constants
(define MT (empty-scene WIDTH HEIGHT))
(define CURSOR (rectangle 1 HEIGHT "solid" "red"))
```

The important point, however, is to write down the wish list for your event handler(s) and your function that draws the state of the editor. Recall that the *2htdp/universe* library dictates the header material for these functions:

```
; Editor -> Image
; renders an editor as an image of the two texts separated by the cursor
(define (editor-render e)
  MT)

; Editor KeyEvent -> Editor
; deals with a key event, given some editor
(define (editor-kh ed ke)
  ed)
```

In addition, [Designing World Programs](#) demands that you write down a main function for your program:

```
; main : String -> Editor
; launches the editor given some initial string
(define (main s)
  (big-bang (create-editor s "")
    [on-key editor-kh]
    [to-draw editor-render]))
```

Re-read [exercise 179](#) to understand what the initial editor for this program is.

While it doesn't matter what you tackle next, we choose to design `editor-kh` first and `editor-render` second. Since we have the header material, let us explain the functioning of the key event handler with two examples:

```
(check-expect (editor-kh (create-editor "" "") "e")
              (create-editor "e" ""))
(check-expect (editor-kh (create-editor "cd" "fgh")) "e")
```

```
(create-editor "cde" "fgh"))
```

Both of these examples demonstrate what happens when you press the letter “e” on your keyboard. The computer runs the function `editor-kh` on the current state of the editor and “e”. In the first example, the editor is empty, which means that the result is an editor with just the letter “e” in it followed by the cursor. In the second example, the cursor is between the strings “cd” and “fgh”, and therefore the result is an editor with the cursor between “cde” and “fgh”. In short, the function always inserts any normal letter at the cursor position.

Before you read on, you should make up examples that illustrate how `editor-kh` works when you press the backspace (“\b”) key to delete some letter, the “left” and “right” arrow keys to move the cursor, or some other arrow keys. In all cases, consider what should happen when the editor is empty, when the cursor is at the left end or right end of the non-empty string in the editor, and when it is in the middle. Even though you are not working with intervals here, it is still a good idea to develop examples for the “extreme” cases.

Once you have test cases, it is time to develop the template. In the case of `editor-kh` you are working with a function that consumes two complex forms of data: one is a structure containing lists, the other one is a large enumeration of strings. Generally speaking, the use of two complex inputs calls for a special look at the design recipe; but in cases like these, it is also clear that you should deal with one of the inputs first, namely, the keystroke.

Having said that, the template is just a large `cond` expression for checking which `KeyEvent` the function received:

```
(define (editor-kh ed k)
  (cond
    [(key=? k "left") ...]
    [(key=? k "right") ...]
    [(key=? k "\b") ...]
    [(key=? k "\t") ...]
    [(key=? k "\r") ...]
    [(= (string-length k) 1) ...]
    [else ...]))
```

The `cond` expression doesn’t quite match the data definition for `KeyEvent` because some `KeyEvents` need special attention (for example “left”, “right”, “\b”); some need to be ignored (“\t” and “\r”) because they are special; and some should be classified into one large group (ordinary keys).

Exercise 180. Explain why the template for `editor-kh` deals with “\t” and “\r” before it checks for strings of length 1.

For the fifth step—the definition of the function—we tackle each clause in the conditional separately. The first clause demands a result that moves the cursor and leaves the string content of the editor alone. So does the second clause. The third clause, however, demands the deletion of a letter from the editor’s content—if there is a letter. Last but not least, the sixth `cond` clause concerns the addition of letters at the cursor position.

Following the first basic guideline, we make extensive use of a wish list and imagine one function per task:

```
(define (editor-kh ed k)
  (cond
```

```
[ (key=? k "left") (editor-lft ed) ]
[ (key=? k "right") (editor-rgt ed) ]
[ (key=? k "\b") (editor-del ed) ]
[ (key=? k "\t") ed]
[ (key=? k "\r") ed]
[(= (string-length k) 1) (editor-ins ed k)]
[else ed]))
```

As you can tell from the definition of `editor-kh`, three of the four wish list functions have the same signature:

```
; Editor -> Editor
```

The last one takes two arguments instead of one:

```
; Editor 1String -> Editor
```

We leave the proper formulation of wishes for the first three functions to you and focus on the fourth one.

Let us start with a purpose statement and a function header:

```
; insert the 1String k between pre and post
(define (editor-ins ed k)
  ed)
```

The purpose is straight out of the problem statement. For the construction of a function header, we need an instance of `Editor`. Since `pre` and `post` are the pieces of the current one, we just put them back together.

Next we derive some examples for `editor-ins` from the examples for `editor-kh`:

```
(check-expect
  (editor-ins (make-editor '() '()) "e")
  (make-editor (cons "e" '()) '()))

(check-expect
  (editor-ins (make-editor (cons "d" '())
                           (cons "f" (cons "g" '())))
              "e")
  (make-editor (cons "e" (cons "d" '()))
               (cons "f" (cons "g" '()))))
```

You should work through these examples using the interpretation of `Editor`. That is, make sure you understand what the given editor means as information and what the function call is supposed to achieve in terms of information. In this particular case, it is best to draw the visual representation of the editor because it is the information of that we have in mind.

The fourth step demands the development of the template. The first argument is guaranteed to be a structure, and the second one is a string, an atomic piece of data. In other words, the template just pulls out the pieces from the given editor representation:

```
(define (editor-ins ed k)
  (... ed ... k ... (editor-pre ed) ... (editor-post ed) ...))
```

As a reminder that the parameters are available too, we have added them to the template body.

From the template and the examples, it is relatively easy to conclude what `editor-ins` is supposed to create an editor from the given editor's `pre` and `post` fields with `k` added to the front of the former:

```
(define (editor-ins ed k)
  (make-editor (cons k (editor-pre ed)) (editor-post ed)))
```

Even though both `(editor-pre ed)` and `(editor-post ed)` are list of `1String`s, there is clearly no need to design auxiliary functions. To get the desired result, it suffices to use the simplest built-in functions for lists: `cons`, which creates lists.

At this point, you should do two things. First, run the tests for this function. Second, use the interpretation of `Editor` and explain abstractly why this function performs the insertion. And if this isn't enough, you may wish to compare this simple definition with the one from [exercise 85](#) and figure out why the other one needs an auxiliary function while our definition here doesn't.

Exercise 181. Design the functions

```
; Editor -> Editor
; moves the cursor position one 1String left, if possible
(define (editor-lft ed)
  ed)

; Editor -> Editor
; moves the cursor position one 1String right, if possible
(define (editor-rgt ed)
  ed)

; Editor -> Editor
; deletes one 1String to the left of the cursor, if possible
(define (editor-del ed)
  ed)
```

Again, it is critical that you work through a good range of examples.

Designing the rendering function for `Editors` poses some new but small challenges. The first one is to develop a sufficiently large number of test cases. On one hand, it demands coverage of the possible combinations: an empty string to the left of the cursor, an empty one on the right, and both strings empty. On the other hand, it also requires some experimenting with the functions that the image library provides. Specifically, it needs a way to compose the two pieces of strings rendered as text images; and it needs a way of placing the text image into the empty image frame (MT). Here is what we do to create an image for the result of `(create-editor "pre" "post")`:

```
(place-image/align
  (beside (text "pre" FONT-SIZE FONT-COLOR)
          CURSOR
          (text "post" FONT-SIZE FONT-COLOR))
  1 1
  "left" "top"
  MT)
```

If you compare this with the editor image above, you notice some differences, which is fine because the exact layout isn't essential to the purpose of this exercise, and because the revised layout doesn't trivialize the problem. In any case, do experiment in the interactions

area of DrRacket to find your favorite editor display.

You are now ready to develop the template, and you should come up with this much:

```
(define (editor-render e)
  (... (editor-pre e) ... (editor-post e)))
```

The given argument is just a structure type with two fields. Their values, however, are lists of `1Strings`, and you might be tempted to refine the template even more. Don't! Instead, keep in mind that when one data definition refers to another complex data definition, you are better off using the wish list.

If you have worked through a sufficient number of examples, you also know what you want on your wish list: one function that turns a string into a text of the right size and color. Let us call this function `editor-text`. Then the definition of `editor-render` just uses `editor-text` twice and then composes the result with `beside` and `place-image`:

```
; Editor -> Image
(define (editor-render e)
  (place-image/align
    (beside (editor-text (editor-pre e))
            CURSOR
            (editor-text (editor-post e)))
    1 1
    "left" "top"
    MT))
```

Although this definition nests expressions three levels deep, the use of the imaginary `editor-text` function renders it quite readable.

What remains is to design `editor-text`. From the design of `editor-render`, we know that `editor-text` consumes a list of `1Strings` and produces a text image:

```
; Lo1s -> Image
; renders a list of 1Strings as a text image
(define (editor-text s)
  (text "" FONT-SIZE FONT-COLOR))
```

The header here produces an empty string, using the defined font size and font color.

To demonstrate what `editor-text` is supposed to compute, we work through an example derived from the example for `editor-render`. The example input is

```
(create-editor "pre" "post")
```

which is equivalent to

```
(make-editor
  (cons "e" (cons "r" (cons "p" '())))
  (cons "p" (cons "o" (cons "s" (cons "t" '())))))
```

We pick the second list as our sample input for `editor-text`, and we know the expected result from the example for `editor-render`:

```
(check-expect
  (editor-text (cons "p" (cons "o" (cons "s" (cons "t" '())))))
  (text "post" FONT-SIZE FONT-COLOR))
```

You may wish to make up a second example before reading on.

Given that `editor-text` consumes a list of `1Strings`, we can write down the template without much ado:

```
(define (editor-text s)
  (cond
    [(empty? s) ...]
    [else (... (first s)
                ... (editor-text (rest s)) ...))])
```

After all, the template is dictated by the data definition that describes the function input. But you don't need the template if you understand and keep in mind the interpretation for `Editor`. It uses `explode` to turn a string into a list of `1Strings`. Naturally, there is a function `implode` that performs the inverse computation, i.e.,

```
> (implode (cons "p" (cons "o" (cons "s" (cons "t" '())))))
"post"
```

Using this function, the definition of `editor-text` is just a small step from the example to the function body:

```
(define (editor-text s)
  (text (implode s) FONT-SIZE FONT-COLOR))
```

Exercise 182. Design `editor-text`. That is, use the standard design recipe and do not fall back on `implode`.

The true surprise comes when you test the two functions. While our test for `editor-text` succeeds, the test for `editor-render` fails. An inspection of the failure shows that the string to the left of the cursor—`"pre"`—is type-set backwards. We forgot to remember that this part of the editor's state is represented in reverse. Conversely, when we render it, we need to reverse it again. Fortunately, the unit tests for the two functions pinpoint which function is wrong and even tell us what is wrong with the function. The fix is trivial:

```
(define (editor-render ed)
  (place-image/align
    (beside (editor-text (reverse (editor-pre ed)))
            CURSOR
            (editor-text (editor-post ed)))
    1 1
    "left" "top"
    MT))
```

It uses the `reverse` function on the `pre` field of `ed`.

Note Modern applications allow users to position the cursor with the mouse (or other gesture-based devices). While it is in principle possible to add this capability to your editor, we wait with doing so until [A Graphical Editor, with Mouse](#).

12 Design by Composition

This last chapter of part II covers “design by composition.” By now you know that programs are complex products and that their production requires the design of many collaborating functions. This collaboration works well if the designer knows when to

design several functions and how to compose these functions into one program.

You have encountered this need to design interrelated functions several times. Sometimes a problem statement implies several different tasks, and each task is best realized with a function. At other times, a data definition may refer to another one, and in that case, a function processing the former kind of data relies on a function processing the latter.

In this chapter, we present yet other scenarios that call for the design of many functions and their composition. To support this kind of work, the chapter presents some informal guidelines on divvying up functions and composing them. The next chapter then introduces some well-known examples whose solutions rely on the use of these guidelines and the design recipe in general. Since these examples demand complex forms of lists, however, this chapter starts with a section on a convenient way to write down complex lists.

12.1 The `list` Function

At this point, you should have tired of writing so many `conses` just to create a list, especially for lists that contain a bunch of values. Fortunately, we have an additional teaching language for you that provides mechanisms for simplifying this part of a programmer's life. BSL+ does so, too.

The key innovation is the `list` operation, which consumes an arbitrary number of values and creates a list. The simplest way to understand `list` expressions is to think of them as abbreviations. Specifically, every expression of the shape

Yes, you have graduated from BSL. It is time to use the “Language” menu and to select the “Choose Language” entry. From now until further notice, use “Beginning Student Language with List Abbreviations” to work through the book.

```
| (list exp-1 ... exp-n)
```

stands for a series of n `cons` expressions:

```
| (cons exp-1 (cons ... (cons exp-n '()))))
```

Keep in mind that `'()` is not an item of the list here, but the rest of the list. Here is a table with three examples:

short-hand	long-hand
<code>(list "ABC")</code>	<code>(cons "ABC" '())</code>
<code>(list #false #true)</code>	<code>(cons #false (cons #true '()))</code>
<code>(list 1 2 3)</code>	<code>(cons 1 (cons 2 (cons 3 '()))))</code>

They introduce lists with one, two, and three items, respectively.

Of course, we can apply `list` not only to values but also to expressions:

```
| > (list (+ 0 1) (+ 1 1))
 '(1 2)
 | > (list (/ 1 0) (+ 1 1))
 /: division by zero
```

Before the list is constructed, the expressions must be evaluated. If during the evaluation of an expression an error occurs, the list is never formed. In short, `list` behaves just like any other primitive operation that consumes an arbitrary number of arguments; its result

just happens to be a list constructed with `conses`.

The use of `list` greatly simplifies the notation for lists with many items and lists that contains lists or structures. Here is an example:

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

This list contains 10 items and its formation with `cons` would require 10 uses of `cons` and one instance of `'()`. Similarly, the list

```
(list (list "bob" 0 "a")
      (list "carl" 1 "a")
      (list "dana" 2 "b")
      (list "erik" 3 "c")
      (list "frank" 4 "a")
      (list "grant" 5 "b")
      (list "hank" 6 "c")
      (list "ian" 8 "a")
      (list "john" 7 "d")
      (list "karel" 9 "e"))
```

requires 11 uses of `list`, which sharply contrasts with 40 `cons` and 11 additional uses of `'()`.

Exercise 183. Use `list` to construct the equivalent of the following lists:

1. `(cons "a" (cons "b" (cons "c" (cons "d" (cons "e" '())))))`
2. `(cons (cons 1 (cons 2 '())) '())`
3. `(cons "a" (cons (cons 1 '()) (cons #false '()))))`
4. `(cons (cons 1 (cons 2 '())) (cons (cons 2 '()) '()))`
5. `(cons (cons "a" (cons 2 '())) (cons "hello" '()))`

Start by determining how many items each list and each nested list contains. Use `check-expect` to express your answers; this ensures that your abbreviations are really the same as the long-hand.

Exercise 184. Use `cons` and `'()` to construct the equivalent of the following lists:

1. `(list 0 1 2 3 4 5)`
2. `(list (list "adam" 0) (list "eve" 1) (list "louisXIV" 2))`
3. `(list 1 (list 1 2) (list 1 2 3))`

Use `check-expect` to express your answers.

Exercise 185. On some occasions lists are formed with `cons` and `list`. Reformulate the following lists using `cons` and `'()` exclusively:

1. `(cons "a" (list 0 #false))`
2. `(list (cons 1 (cons 13 '()))))`
3. `(cons (list 1 (list 13 '())) '())`

4. `(list '() '() (cons 1 '()))`
5. `(cons "a" (cons (list 1) (list #false '()))))`

Then formulate the lists using only `list`. Use `check-expect` to express your answers.

Exercise 186. Determine the values of the following expressions:

1. `(list (string=? "a" "b") (string=? "c" "c") #false)`
2. `(list (+ 10 20) (* 10 20) (/ 10 20))`
3. `(list "dana" "jane" "mary" "laura")`

Use `check-expect` to express your answers.

Exercise 187. You know about `first` and `rest` from BSL, but BSL+ comes with even more selectors than that. Determine the values of the following expressions:

1. `(first (list 1 2 3))`
2. `(rest (list 1 2 3))`
3. `(second (list 1 2 3))`

Find out from the documentation whether `third`, `fourth`, and `fifth` exist.

12.2 Composing Functions

[How to Design Programs](#) explains that programs are collections of definitions: structure type definitions, data definitions, constant definitions, and function definitions. To guide the division of labor among functions, the section also suggests a rough guideline:

And don't forget tests.

Formulate auxiliary function definitions for every dependency between quantities in the problem statement. In short, design one function per task.

This part of the book introduces another guideline on auxiliary functions:

Formulate auxiliary function definitions when one data definition points to a second data definition. Roughly, design one template per data definition.

In this section, we take a look at one specific place in the design process that may call for additional auxiliary functions: the definition step, which creates a full-fledged definition from a template. Turning a template into a complete function definition means combining the values of the template's sub-expressions into the final answer. As you do so, you might encounter several situations that suggest the need for auxiliary functions:

1. If the composition of values requires knowledge of a particular domain of application—for example, composing two (computer) images, accounting, music, or science—design an auxiliary function.
2. If the composition of values requires a case analysis of the available values—for example, is a number positive, zero, or negative—use a `cond` expression. If the `cond` looks complex, design an auxiliary function whose inputs are the partial results and whose function body is the `cond` expression. Doing so separates out the case analysis

from the recursive process.

3. If the composition of values must process an element from a self-referential data definition—a list, a natural number, or something like those—design an auxiliary function.
4. If everything fails, you may need to design a **more general** function and define the main function as a specific use of the general function. This suggestion sounds counter-intuitive but it is called for in a remarkably large number of cases.

The last two criteria are situations that we haven't discussed in any detail, though examples have come up before. The next two sections illustrate these principles with additional examples.

Before we move forward, though, remember that the key to managing the design of many functions and their compositions is to maintain the often-mentioned

Wish Lists

Maintain a list of function headers that must be designed to complete a program. Writing down complete function headers ensures that you can test those portions of the programs that you have finished, which is useful even though many tests will fail. Of course, when the wish list is empty, all tests should pass and all functions should be covered by tests.

Before you put a function on the wish list, you should check whether something like the function already exists in your language's library or whether something similar is already on the wish list. BSL, BSL+, and indeed all programming languages provide many built-in operations and many library functions. You should explore your chosen language when you have time and when you have a need, so that you know what it provides.

12.3 Recursive Auxiliary Functions

People need to sort things all the time, and so do programs. Investment advisors sort portfolios by the profit each holding generates. Game programs sort lists of players according to scores. And mail programs sort messages according to date or sender or some other criteria.

In general, you can sort a bunch of items if you can compare and order each pair of data items. Although not every kind of data comes with a comparison primitive, we all know one that does: numbers. Hence, we use a simplistic but highly representative sample problem in this section:

Sample Problem: Design a function that sorts a list of (real) numbers.

The exercises of this section clarify how to adapt this design to other forms of data.

Since the problem statement doesn't mention any other task and since sorting doesn't seem to suggest other tasks, we just follow the design recipe. Sorting means rearranging a bunch of numbers. This re-statement implies a natural data definition for the inputs and outputs of the function and thus its signature. Given that we have a definition for [List-of-numbers](#), the first step is easy:

```
; List-of-numbers -> List-of-numbers
; produces a sorted version of alon
(define (sort> alon)
```

```
|   alon)
```

Returning `alon` ensures that the result is appropriate as far as the function signature is concerned, but in general, the given list isn't sorted and this result is wrong.

When it comes to making up examples, it quickly becomes clear that the problem statement is quite imprecise. As before, we use the data definition of [List-of-numbers](#) to organize the development of examples. Since the data definition consists of two clauses, we need two examples. Clearly, when `sort>` is applied to `'()`, the result must be `'()`. The question is what the result for

```
| (cons 12 (cons 20 (cons -5 '()))))
```

should be. The list isn't sorted, but there are two ways to sort it:

- `(cons 20 (cons 12 (cons -5 '()))))`, i.e., a list with the numbers arranged in **descending** order; and
- `(cons -5 (cons 12 (cons 20 '()))))`, i.e., a list with the numbers arranged in **ascending** order.

In a real-world situation, you would now have to ask the person who posed the problem for clarification. Here we go for the descending alternative; designing the ascending alternative doesn't pose any different obstacles.

The decision calls for a revision of the header material:

```
; List-of-numbers -> List-of-numbers
; rearrange alon in descending order

(check-expect (sort> '()) '())

(check-expect (sort> (list 12 20 -5)) (list 20 12 -5))

(check-expect (sort> (list 3 2 1)) (list 3 2 1))

(check-expect (sort> (list 1 2 3)) (list 3 2 1))

(define (sort> alon)
  alon)
```

The header material now includes the examples reformulated as unit tests and using `list`. If the latter makes you uncomfortable, reformulate the test with `cons` to exercise translating back and forth. As for the additional two examples, they demand that `sort` works on lists sorted in ascending and descending order.

Next we must translate the data definition into a function template. We have dealt with lists of numbers before, so this step is easy:

```
(define (sort> alon)
  (cond
    [(empty? alon) ...]
    [else (... (first alon) ... (sort> (rest alon)) ...)]))
```

Using this template, we can finally turn to the interesting part of the program development. We consider each case of the `cond` expression separately, starting with the simple case. If `sort>`'s input is `'()`, the answer is `'()`, as specified by the example. If

`sort>`'s input is a **consed** list, the template suggests two expressions that might help:

- `(first alon)` extracts the first number from the input;
- `(sort> (rest alon))` produces a sorted version of `(rest alon)`, according to the purpose statement of the function.

To clarify these abstract answers, let us use the second example to explain these pieces in detail. When `sort>` consumes `(list 12 20 -5)`,

1. `(first alon)` is `12`,
2. `(rest alon)` is `(list 20 -5)`, and
3. `(sort> (rest alon))` produces `(list 20 -5)`, because this list is already sorted.

To produce the desired answer, `sort>` must insert `12` between the two numbers of the last list. More generally, we must find an expression that inserts `(first alon)` in its proper place into the result of `(sort> (rest alon))`. If we can do so, sorting is an easily solved problem.

Inserting a number into a sorted list clearly isn't a simple task. It demands searching through the sorted list to find the proper place of the item. Searching through any list demands an auxiliary function, because lists are of arbitrary size and, by hint 3 of the preceding section, processing values of arbitrary size, calls for the design of an auxiliary function.

So here is the new wish list entry:

```
; Number List-of-numbers -> List-of-numbers
; inserts n into the sorted list of numbers alon
(define (insert n alon)
  alon)
```

That is, `insert` consumes a number and a list sorted in descending order and produces a sorted list by inserting the former into the right place in the latter.

Using `insert`, it is easy to complete the definition of `sort>`:

```
(define (sort> alon)
  (cond
    [(empty? alon) '()]
    [else (insert (first alon) (sort> (rest alon))))]))
```

The second clause says that, in order to produce the final result, `sort>` extracts the first item of the non-empty list, computes the sorted version of the rest of the list, and uses `insert` to produce the completely sorted list from the two pieces.

Before you read on, test the program. You will see that some test cases pass and some fail. That is progress but clearly, we are not really finished until we have developed `insert`. The next step in its design is the creation of functional examples. Since the first input of `insert` is any number, we use `5` and use the data definition for **List-of-numbers** to make up examples for the second input. First we consider what `insert` should produce when given a number and `'()`. According to `insert`'s purpose statement, the output must be a list, it must contain all numbers from the second input, and it must contain the first argument. This suggests the following:

```
(check-expect (insert 5 '()) (list 5))
```

Instead of 5, we could have used any number.

Second, we use a non-empty list of just one item:

```
(check-expect (insert 5 (list 6)) (list 6 5))
(check-expect (insert 5 (list 4)) (list 5 4))
```

The reasoning of why these are the expected results is just like before. For one, the result must contain all numbers from the second list and the extra number. For two, the result must be sorted.

Finally, let us create an example with a list that contains more than one item. Indeed, we can derive such an example from the examples for `sort>` and especially from our analysis of the second `cond` clause. From there, we know that `sort>` works only if 12 is inserted into `(list 20 -5)` at its proper place:

```
(check-expect (insert 12 (list 20 -5)) (list 20 12 -5))
```

That is, `insert` is given a second list and it is sorted in descending order.

Note what the development of examples teaches us. The `insert` function has to find the first number that is smaller than the given n. When there is no such number, the function eventually reaches the end of the list and it must add n to the end. Now, before we move on to the template, you should work out some additional examples. To do so, you may wish to use the supplementary examples for `sort>`.

In contrast to `sort>`, the function `insert` consumes **two** inputs. Since we know that the first one is a number and atomic, we can focus on the second argument—the list of numbers—for the template development:

```
(define (insert n alon)
  (cond
    [(empty? alon) ...]
    [else (... (first alon) ... (insert n (rest alon)) ...)]))
```

The only difference between this template and the one for `sort>` is that this one needs to take into account the additional argument n.

To fill the gaps in the template of `insert`, we again proceed on a case-by-case basis. The first case concerns the empty list. According to the first example, `(list n)` is the expression needed in the first `cond` clause, because it constructs a sorted list from n and alon.

The second case is more complicated than the first, and so we follow the questions from [figure 44](#):

1. `(first alon)` is the first number on alon, and
2. `(rest alon)` is the rest of alon and, like alon, it is sorted in descending order;
3. `(insert n (rest alon))` produces a sorted list from n and the numbers on `(rest alon)`.

The problem is how to combine these pieces of data to get the final answer.

Let us work through some examples to make all this concrete:

```
(insert 7 (list 6 5 4))
```

Here n is 7 and larger than any of the numbers in the second input. We know so by just looking at the first item of the list. It is 6 but because the list is sorted, all other numbers on the list are even smaller than 6. Hence it suffices if we just `cons` 7 onto (`list` 6 5 4).

In contrast, when the application is something like

```
| (insert 0 (list 6 2 1 -1))
```

n must indeed be inserted into the rest of the list. More concretely, (`first` `alon`) is 6; (`rest` `alon`) is (`list` 2 1 -1); and (`insert` n (`rest` `alon`)) produces (`list` 2 1 0 -1) according to the purpose statement. By adding 6 back onto that last list, we get the desired answer for (`insert` 0 (`list` 6 2 1 -1)).

To get a complete function definition, we must generalize these examples. The case analysis suggests a nested conditional that determines whether n is larger than (or equal to) (`first` `alon`):

- If so, all the items in `alon` are smaller than n because `alon` is already sorted. The answer in that case is (`cons` n `alon`).
- If, however, n is smaller than (`first` `alon`), then the function has not yet found the proper place to insert n into `alon`. The first item of the result must be (`first` `alon`) and that n must be inserted into (`rest` `alon`). The final result in this case is

```
| (cons (first alon) (insert n (rest alon)))
```

because this list contains n and all items of `alon` in sorted order—which is what we need.

The translation of this discussion into BSL+ calls for an `if` expression for such cases. The condition is (`>= n (first alon)`) and the expressions for the two branches have been formulated.

Figure 56 contains the complete definitions of `insert` and `sort>`. Copy it into the definition area of DrRacket, add the test cases back in, and test the program. All tests should pass now and they should cover all expressions.

Terminology This particular program for sorting is known as *insertion sort* in the programming literature. Later we will study alternative ways to sort lists, using an entirely different design strategy.

```
; List-of-numbers -> List-of-numbers
; produces a sorted version of alon
(define (sort> alon)
  (cond
    [(empty? alon) '()]
    [(cons? alon) (insert (first alon) (sort> (rest alon))))]))

; Number List-of-numbers -> List-of-numbers
; inserts n into the sorted list of numbers alon
(define (insert n alon)
  (cond
    [(empty? alon) (cons n '())]
    [else (if (>= n (first alon))
              (cons n alon)
              (cons (first alon) (insert n (rest alon))))])))
```

Figure 56: Sorting lists of numbers

Exercise 188. Take a second look at [Intermezzo: BSL](#), the intermezzo that presents BSL and its ways of formulating tests. One of the latter is [check-satisfied](#), which determines whether an expression satisfies a certain property.

Use `sorted>?` from [exercise 147](#) to re-formulate the tests for `sort>` with [check-satisfied](#).

Now consider this function definition:

```
; List-of-numbers -> List-of-numbers
; produces a sorted version of l
(define (sort>/bad l)
  '(9 8 7 6 5 4 3 2 1 0))
```

Can you formulate a test case that shows `sort>/bad` is **not** a sorting function? Can you use [check-satisfied](#) to formulate this test case?

Notes (1) What may surprise you here is that we define a function to create a test. In the real world, this step is common and, on occasion, you really need to design functions for tests—with their own tests and all. (2) Formulating tests with [check-satisfied](#) is occasionally easier than using [check-expect](#) (or other forms), and it is also a bit more general. When the predicate completely describes the relationship between all possible inputs and outputs of a function, computer scientists speak of a *specification*. [Specifying with lambda](#) explains how to specific `sort>` completely.

Exercise 189. Design a program that sorts lists of email messages by date:

```
(define-struct email [from date message])
; A Email Message is a structure:
;   (make-email String Number String)
; interpretation (make-email f d m) represents text m sent by
; f, d seconds after the beginning of time
```

Also develop a program that sorts lists of email messages by name. To compare two strings alphabetically, use the [string<?](#) primitive.

Exercise 190. Design a program that sorts lists of game players by score:

```
(define-struct gp [name score])
; A GamePlayer is a structure:
;   (make-gp String Number)
; interpretation (make-gp p s) represents player p who scored
; a maximum of s points
```

Exercise 191. Here is the function `search`:

```
; Number List-of-numbers -> Boolean
(define (search n alon)
  (cond
    [(empty? alon) #false]
    [else (or (= (first alon) n) (search n (rest alon)))]))
```

It determines whether some number occurs in a list of numbers. The function may have to traverse the entire list to find out that the number of interest isn't contained in the list.

Develop the function `search-sorted`, which determines whether a number occurs in a sorted list of numbers. The function must take advantage of the fact that the list is sorted.

Exercise 192. Design `prefixes`. The function consumes a list of `1Strings` and produces the list of all prefixes. People say that a list `p` is a *prefix* of `l` if `p` and `l` are the same up through all items in `p`. For example, `(list 1 2 3)` is a prefix of itself and `(list 1 2 3 4)`.

Design the function `suffixes`, which consumes a list of `1Strings` and produces all `suffixes`. A list `s` is a *suffix* of `l` if `p` and `l` are the same from the end, up through all items in `s`. For example, `(list 2 3 4)` is a suffix of itself and `(list 1 2 3 4)`.

12.4 Generalizing Functions

Auxiliary functions are also needed when a problem statement is too narrow. Conversely, it is common for programmers to generalize a given problem just a bit to simplify the solution process. When they discover the need for a generalized solution, they design an auxiliary function that solves the generalized problem and a main function that just calls this auxiliary function with special arguments.

We illustrate this idea with a solution for the following problem:

Sample Problem: Design a program that renders a polygon into an empty 50 by 50 scene.

Mr. Paul C. Fisher inspired the use of this problem.

Just in case you don't recall your basic geometry (domain) knowledge, we add one definition of polygon:

A *polygon* is a planar figure with at least three corners consecutively connected by three straight sides. And so on.

One natural data representation for a polygon is thus a list of `Posns` where each one represents the coordinates of one corner. For example,

```
(list (make-posn 10 10)
      (make-posn 60 60)
      (make-posn 10 60))
```

represents a triangle. You may wonder what `'()` or `(list (make-posn 30 40))` mean as a polygon and the answer is that they do **not** describe polygons, and neither do lists with two `Posns`.

As the domain knowledge statement says, a polygon consists of at least three sides and that means at least three `Posns`. Thus the answer to our question is that representations of polygons should be lists of at least three `Posns`.

Following the development of the data definition for `NEList-of-temperatures` in `Non-empty Lists`, formulating a data representation for polygons is straightforward:

```
; a Polygon is one of:
; - (list Posn Posn Posn)
; - (cons Posn Polygon)
```

The first clause says that a list of three `Posns` is a `Polygon` and the second clause says that `consing` another `Posn` onto some existing `Polygon` creates another one. Since this data definition is the very first to use `list` in one of its clauses, we spell it out with `cons` just to

make sure you see this conversion from an abbreviation to long hand in this context:

```
; a Polygon is one of:  
; - (cons Posn (cons Posn (cons Posn '()))))  
; - (cons Posn Polygon)
```

The point of this discussion is that a naively chosen data representation—lists of *Posns*—may not properly represent the intended information. Revising the data definition during such an exploration is a normal step; indeed, on occasion such revisions become necessary during the rest of the design process. As long as you stick to a systematic approach, though, changes to the data definition can naturally be propagated through the rest of the program design.

The second step calls for the signature, purpose statement, and header of the program. Since the problem statement mentions just one task and no other task is implied, we start with one function:

```
(define MT (empty-scene 50 50))

; Polygon -> Image
; renders the given polygon p into MT
(define (render-poly p)
  MT)
```

The separate definition of *MT* is also called for, considering that the empty scene is mentioned in the problem statement and that it is going to be needed to formulate examples.

For the first example, we use the above-mentioned triangle. A quick look in the *2htdp/image* library suggests *scene+line* is the function needed to render the three lines for a triangle:

```
(check-expect
  (render-poly
    (list (make-posn 20 0) (make-posn 10 10) (make-posn 30 10)))
  (scene+line
    (scene+line
      (scene+line MT 20 0 10 10 "red")
      10 10 30 10 "red")
    30 10 20 0 "red"))
```

The innermost *scene+line* render the line from the first to the second *Posn*; the middle one uses the second and third *Posn*; and the outermost *scene+line* connects the third and the first *Posn*. Of course, we experimented in DrRacket's interaction area to get this expression right, and you should, too.

Given that the first and smallest polygon is a triangle, a rectangle or a square suggests itself as the second example:

```
(check-expect
  (render-poly
    (list (make-posn 10 10) (make-posn 20 10)
          (make-posn 20 20) (make-posn 10 20)))
  (scene+line
    (scene+line
      (scene+line
        (scene+line MT 10 10 20 10 "red"))
```

```

 20 10 20 20 "red")
20 20 10 20 "red")
10 20 10 10 "red"))

```

Both add just one corner to triangles and both are easy to render. You may also wish to draw these shapes on a piece of graph paper before you experiment.

The construction of the template poses a serious challenge. Specifically, the first and the second question of figure 43 ask whether the data definition differentiates distinct subsets and how to distinguish among them. While the data definition clearly sets apart triangles from all other polygons in the first clause, it is not immediately clear how to differentiate the two. Both clauses describe lists of `Posns`. The first describes lists of three `Posns`, the second one describes lists of `Posns` that have at least four items. Thus one alternative is to ask whether the given polygon is three items long:

```
(= (length p) 3)
```

Using the long-hand version of the first clause, i.e.,

```
(cons Posn (cons Posn (cons Posn '()))))
```

suggests a second way to formulate the condition, namely, checking whether the given `Polygon` is empty after using three `rest` functions:

```
(empty? (rest (rest (rest p))))
```

Since all `Polygons` consist of at least three `Posns`, using `rest` three times is legal. Unlike `length`, `rest` is a primitive, easy-to-understand operation with a clear operational meaning. It selects the second field in a `cons` structure and that is all it does.

The rest of the questions in figure 43 have direct answers, and thus we get this template:

In general, it is better to formulate conditions via built-in predicates and selectors than your own (recursive) functions. We will explain this remark in [Intermezzo: The Cost of Computation](#).

```

(define (render-poly p)
  (cond
    [(empty? (rest (rest (rest p))))]
    [... (first p) ... (second p) ... (third p) ...])
    [else ( ... (first p) ... (render-poly (rest p)) ...)])))

```

Because `p` in the first clause describes a triangle, we know that it consists of exactly three `Posns`, which are selected via `first`, `second`, and `third`. Furthermore, `p` in the second clause consists of a `Posn` and a `Polygon`, justifying `(first p)` and `(rest p)`. The former extracts a `Posn` from `p`, the latter a `Polygon`. We therefore add a self-referential function call around the latter; we should also keep in mind that dealing with `(first p)` in this clause and the three `Posns` in the first clause may demand the design of an auxiliary function.

Now we are ready to focus on the function definition, dealing with one clause at a time. The first clause concerns triangles, which in turn suggests a relatively straightforward solution. Specifically, there are three `Posns` and `render-poly` should connect the three in an empty scene of 50 by 50 pixels. Given `Posn` is a separate data definition, we get an obvious wish list entry:

```
; Image Posn Posn -> Image
; draws a red line from Posn p to Posn q into im
(define (render-line im p q)
  im)
```

Using this function, the first `cond` clause in `render-poly` can easily create an answer like this:

```
(render-line
  (render-line
    (render-line MT (first p) (second p))
    (second p) (third p))
  (third p) (first p))
```

This expression obviously renders the given `Polygon` `p` as a triangle by drawing a line from the first to the second, the second to the third, and the third to the first `Posn`.

The second `cond` clause is about `Polygons` that have been extended with one additional `Posn`. In the template, we find two expressions and, following [figure 44](#), we remind ourselves of what these expressions compute:

1. `(first p)` extracts the first `Posn`;
2. `(rest p)` extracts the `Polygon` from `p`; and
3. `(render-polygon (rest p))` renders the `Polygon` extracted by `(rest p)` in `MT`.

The question is how to use these pieces to render the originally given `Polygon` `p`.

One idea that may come to mind is that `(rest p)` consists of at least three `Posns`. It is therefore possible to extract at least one `Posn` from this embedded `Polygon` and to connect `(first p)` with this additional point. Here is what this idea looks like with BSL+ code:

```
(render-line (render-poly (rest p)) (first p) (second p))
```

As mentioned, the highlighted sub-expression renders the embedded `Polygon` in an empty 50 by 50 scene. The use of `render-line` adds one line to this scene, from the first to the second `Posn` of `p`.

Our suggestions imply the following complete function definition:

```
(define (render-poly p)
  (cond
    [(empty? (rest (rest (rest p))))]
    [else
      (render-line
        (render-line
          (render-line MT (first p) (second p))
          (second p) (third p))
        (third p) (first p)))]))
```

Designing `render-line` is the kind of problem that you solved in the first part of the book, so we just provide the final definition so that you can test the program:

```
; Image Posn Posn -> Image
; renders a line from p to q into im
```

```
(define (render-line im p q)
  (scene+line
    im (posn-x p) (posn-y p) (posn-x q) (posn-y q) "red"))
```

Sixth and last, we must test the functions. You should add a test for `render-line` but for now you may just accept our correctness promise. In that case, testing immediately reveals flaws with the definition of `render-poly`; in particular, the test case for the square fails. On one hand, this is fortunate because it is the purpose of tests to find problems before they affect regular consumers. On the other hand, the flaw is unfortunate because we followed the design recipe, we made fairly regular decisions, and yet the function doesn't work.

Before you give up hope in such a situation, you should experiment with the function in DrRacket's interactions area. For our example, the results are illuminating. Here is the failing test case, with the input on the left and the output on the right:

```
p = (render-polygon p) =
(list (make-posn 10 10)
      (make-posn 20 10)
      (make-posn 20 20)
      (make-posn 10 20))
```

To highlight the problems, we have added visible dots for the four given `Posns`. As you can see, the image on the right shows that `render-polygon` connects the three dots of `(rest p)` and then connects `(first p)` to the first point of `(rest p)`, i.e., `(second p)`.

You can easily validate this claim with an interaction that uses `(rest p)` directly as input for `render-poly`:

```
p = (render-polygon p) =
(list (make-posn 20 10)
      (make-posn 20 20)
      (make-posn 10 20))
```

And the result is indeed the triangle of the image above.

In addition, you may wonder what `render-poly` would do if we extended the original square with another point, say, `(make-posn 10 5)`.

```
p = (render-polygon p) =
(list (make-posn 10 5)
      (make-posn 10 10)
      (make-posn 20 10)
      (make-posn 20 20)
      (make-posn 10 20))
```

Instead of a polygon, `render-polygon` always draws the triangle at the end of the given `Polygon` and otherwise connects the `Posns` that precede the triangle.

As a matter of fact, one could also argue that `render-polygon` is really the function that connects the successive dots specified by a list of `Posns` and connects the first and the last `Posn` of the trailing triangle. If we don't draw this last, extra line, `render-polygon` is just

a “connects the dots” function. And as such, it almost solves our original problem; all that is left to do is to add a line from the first `Posn` to the last one in the given `Polygon`.

Put differently, the analysis of our failure suggests two ideas at once. First, we should solve a different, a **more general** looking problem. Second, we should use the solution for this generalized problem to solve the original one.

We start with the statement for the general problem:

Sample Problem: Design a program that connects a bunch of dots.

Although the design of `render-poly` almost solves this problem, we design this function mostly from scratch. First, we need a data definition. Connecting the dots makes no sense unless we have at least one dot or two. To make things simple, we go with the first alternative:

```
; A NELoP is one of:
; - (cons Posn '())
; - (cons Posn NELoP)
```

The organization of this data definition should be familiar from [Non-empty Lists](#). In that section, the definition for `NEList-of-temperatures` describes lists that contain at least one temperature; here `NELoP` introduces the collection of lists of `Posns` with at least one `Posn`.

Second, we formulate a signature, a purpose statement, and a header for a “connect the dots” function:

```
; NELoP -> Image
; connects the dots in p by rendering lines in MT
(define (connect-dots p)
  MT)
```

Third, we adapt the examples for `render-poly` for this new function. As our failure analysis says, the function connects the first `Posn` on `p` to the second one, the second one to the third, the third to the fourth, and so on, all the way to the last one, which isn’t connected to anything. Here is the adaptation of the first example, a list of three `Posns`:

```
(check-expect (connect-dots (list (make-posn 20 0)
                                    (make-posn 10 10)
                                    (make-posn 30 10)))
                  (scene+line
                   (scene+line MT 20 0 10 10 "red")
                   10 10 30 10 "red")))
```

The expected value is an expression that adds two lines to `MT`: one line from the first `Posn` to the second one, and another line from the second to the third `Posn`.

Exercise 193. Adapt the second example for `render-poly` to `connect-dots`.

Fourth, we use the template for functions the process non-empty lists:

```
(define (connect-dots p)
  (cond
    [(empty? (rest p)) (... (first p) ...)]
    [else (... (first p) ... (connect-dots (rest p)) ...)]))
```

The template has two clauses: one for lists of one `Posn` and the second one for lists with more than one. Since there is at least one `Posn` in both cases, the template contains

(`first p`) in both clauses; the second one also contains (`connects-dots (rest p)`) to remind us of the self-reference in the second clause of the data definition.

The fifth and central step is to turn the template into a function definition. Since the first clause is the simplest one, we start with it. As we have already said, it is impossible to connect anything when the given list contains only one `Posn`. Hence, the function just returns MT from the first `cond` clause. For the second `cond` clause, let us remind ourselves of what the template expressions compute:

1. (`first p`) extracts the first `Posn`;
2. (`rest p`) extracts the `NELoP` from `p`; and
3. (`connect-dots (rest p)`) renders the `NELoP` extracted by (`rest p`) in MT.

From our first attempt to design `render-poly`, we know that `connect-dots` needs to add one line to the image that (`connect-dots (rest p)`) produces, namely, from (`first p`) to (`second p`). We know that `p` contains a second `Posn`, because otherwise the evaluation of `cond` would have picked the first clause.

Putting everything together, we get the following definition:

```
(define (connect-dots p)
  (cond
    [(empty? (rest p)) MT]
    [else
      (render-line
        (connect-dots (rest p)) (first p) (second p))))
```

This definition looks simpler than the one for the faulty version of `render-poly`, even though it has to cope with two more lists of `Posns` than `render-poly`.

about the equivalence of

Conversely, we say that `connect-dots` generalizes `render-poly`. Every input for the latter is also an input for the former. Or in terms of data definitions, every `Polygon` is also a `NELoP`. But, there are many `NELoPs` that are **not** `Polygons`. To be precise, all lists of `Posns` that contain two items or one belong to `NELoP` but not to `Polygon`. The key insight for you is, however, that just because a function has to deal with more inputs than another function does **not** mean that the former is more complex than the latter; generalizations often simplify function definitions.

Our arguments in this book are informal. If you ever wish to know that `Polygon` is really the same set as `NELoP` or that `connect-dots` generalizes `render-poly`, you will need to construct a formal argument, called a *proof*. While this book's design process is deeply informed by logic, a course on logic in computation is the ideal complement if you ever get interested in proofs. In general, logic is to computing what analysis is to engineering.

As spelled out above, the definition of `render-polygon` can use `connect-dots` to connect all successive `Posns` of the given `Polygon`; to complete its task, it must then add a line from the first to the last `Posn` of the given `Polygon`. In terms of code, this just means composing two functions: `connect-dots` and `render-line`, but we also need a function to extract the last `Posn` from the `Polygon`. Once we are granted this wish, the definition of `render-poly` is a one-liner:

```
; Polygon -> Image
; adds an image of p to MT
(define (render-polygon p)
  (render-line (connect-dots p) (first p) (last p)))
```

Formulating the wish list entry for `last` is straightforward:

```
; Polygon -> Posn
; extracts the last item from p
```

Then again, it is clear that `last` could be a generally useful function and we might be better off designing it for inputs from [NELoP](#):

```
; NELoP -> Posn
; extracts the last item from p
(define (last p)
  (first p))
```

Stop! Why is it acceptable to use `first` for the stub definition of the `last` function? Then solve the following exercise.

Exercise 194. Argue why it is acceptable to use `last` on [Polygons](#). Also argue why you may reuse the template for `connect-dots` for `last`:

```
(define (last p)
  (cond
    [(empty? (rest p)) (... (first p) ...)]
    [else (... (first p) ... (last (rest p)) ...)]))
```

Finally, develop examples for `last`, turn them into tests, and ensure that the definition of `last` in [figure 57](#) works on your examples.

```
; A Polygon is one of:
; - (list Posn Posn Posn)
; - (cons Posn Polygon)

(define MT (empty-scene 50 50))

; Polygon -> Image
; adds an image of p to MT
(define (render-polygon p)
  (render-line (connect-dots p) (first p) (last p)))

; A NELoP is one of:
; - (cons Posn '())
; - (cons Posn NELoP)

; NELoP -> Image
; connects the Posns in p in an image
(define (connect-dots p)
  (cond
    [(empty? (rest p)) MT]
    [else
      (render-line
        (connect-dots (rest p)) (first p) (second p)))]))

; Image Posn Posn -> Image
```

```

; draws a red line from Posn p to Posn q into im
(define (render-line im p q)
  (scene+line
    im (posn-x p) (posn-y p) (posn-x q) (posn-y q) "red"))

; Polygon -> Posn
; extracts the last item from p
(define (last p)
  (cond
    [(empty? (rest (rest (rest p))))] (third p)]
    [else (last (rest p))]))

```

Figure 57: Drawing a polygon

In summary, the development of `render-poly` naturally points us to consider the general problem of connecting a list of successive dots. We can then solve the original problem by defining a function that composes the general function with other auxiliary functions. The program therefore consists of a relatively straightforward main function—`render-poly`—and complex auxiliary functions that perform most of the work. You will see time and again that this kind of design approach is common and a good method for designing and organizing programs.

Exercise 195. Here are two more ideas for defining `render-poly`:

- `render-poly` could `cons` the last item of `p` onto `p` and then call `connect-dots`.
- `render-poly` could add the first item of `p` to the end of `p` via a version of `add-at-end` that works on `Polygons`.

Use both ideas to define `render-poly`; make sure both definitions pass the test cases.

Exercise 196. Modify `connect-dots` so that it consumes an additional `Posn` structure to which the last `Posn` is connected. Then modify `render-poly` to use this new version of `connect-dots`. **Note** This new version is called an **accumulator** version.

Naturally, functions such as `last` are available in a language like Racket and `polygon`—a function that creates an image of a polygon—is available in one of its libraries. If you are wondering why we just designed a similar function, take a look at the title of the book and the section. The point is not (necessarily) to design useful functions, but to study how programs are designed systematically. Specifically, this section is about the idea of using generalization in the design process; for more on this idea see [Abstraction](#) and [Accumulators](#).

13 Projects: Lists

This chapter presents several extended exercises, all of which aim to solidify your understanding of the elements of design: the design of batch and interactive programs, design by composition, design wish lists, and the design recipe for functions. The first section covers problems involving real-world data: English dictionaries and iTunes libraries. A word-games problem requires two sections: one to illustrate design by composition, the other to tackle the heart of the problem. The remaining sections are about games and finite-state machines.

13.1 Real-world Data: Dictionaries

Beginning with [Generative Recursion](#), this book discusses systematic approaches to discussing the performance of programs and improving it. From here to this part, the focus is on designing programs systematically so that you can then explore performance improvements.

Information in the real world tends to come in large quantities, which is why it makes so much sense to use programs for processing it. For example, a dictionary does not just contain a dozen words, but hundreds of thousands. When you want to process such large pieces of information, you must carefully design the program using small examples and tests. Once you have convinced yourself that the programs work properly, you run them on the real-world data to get real results. If the program is too slow to process this large quantity of data, reflect on each function and how it works. Question whether you can eliminate any redundant computations.

```
; On OS X:  
(define DICTIONARY-LOCATION "/usr/share/dict/words")  
  
; On LINUX:  
;   the file 'words' in /usr/share/dict/ or /var/lib/dict/  
  
; On WINDOWS:  
;   borrow the word file from one of your Linux or Mac friends  
  
; Dictionary (is a List-of-strings)  
(define DICTIONARY-AS-LIST (read-lines DICTIONARY-LOCATION))
```

Figure 58: Reading a dictionary

This section relies on the `2http/batch-io` library.

[Figure 58](#) displays the one line of code needed to read in an entire dictionary of the English language. To get an idea of how large such dictionaries are, adapt the code from the figure for your particular computer and use `length` to determine how many words are in your dictionary. There are 235,886 words in ours today, 8 August 2015.

In the following exercises, letters play an important role. You may wish to add the following to the top of your program in addition to your adaptation of [figure 58](#):

```
; Letter is one of the following 1Strings: "a" ... "z"  
; or, equivalently, a member? of the following list:  
(define LETTERS (explode "abcdefghijklmnopqrstuvwxyz"))
```

Hint Use `list` to formulate examples and tests as you work through the following exercises.

Exercise 197. Design the function `starts-with#`, which consumes a `Letter` and `Dictionary` and then counts how many words in the given `Dictionary` start with the given `Letter`. Once you know that your function works, determine how many words start with "`e`" in your computer's dictionary and how many with "`z`".

Exercise 198. Design `count-by-letter`. The function consumes a list of `Letters` and a `Dictionary`. It produces a list of `Letter-Counts`, that is, pairs of `Letters` and `Ns`; in such a pair, the latter says how many times the former occurs in the given `Dictionary`. Once your function is designed, determine how many words appear for all letters in your computer's dictionary.

Hint The function consumes two lists, a design problem that is covered in [Simultaneous Processing](#) in detail. For now, just assume that `Dictionary` is an atomic piece of data and is along for the ride through `count-by-letter`. It is handed over to other functions that then treat it as a list.

Exercise 199. Design `most-frequent`. The function consumes a `Dictionary` and produces the `Letter-Count` for the letter that is most frequently used as the first one in the words of the given `Dictionary`.

What is the most frequently used letter in your computer's dictionary and how often is it used?

Note on Design Choices This exercise calls for the composition of the solution to the preceding exercise with a function that picks the correct pairing of a letter with a count. There are two days to design this function:

- Design a function that picks the pair with the maximum count.
- Design a function that selects the first pair from a sorted version of the list of pairs.

Consider designing both. Which one do you prefer? Why?

Exercise 200. Design `words-by-first-letter`. The function consumes a `Dictionary` and produces a list of `Dictionarys`, one per `Letter`.

Re-design `most-frequent` from [exercise 199](#) using `words-by-first-letter`; call the new function `most-frequent.v2`. Once you have completed the design, ensure that the two functions compute the same result on your computer's dictionary:

```
(define dictionary (read-lines DICTIONARY-LOCATION))
(check-expect (most-frequent dictionary) (most-frequent.v2 dictionary))
```

Note on Design Choices For `words-by-first-letter` you have a choice for dealing with the situation when the given dictionary does not contain any words for some letter:

- One alternative is to exclude the resulting empty dictionaries from the overall result. Doing so simplifies both the testing of the function and the design of `most-frequent.v2`, but it also requires the design of an auxiliary function.
- The other one is to include `'()` as the result of looking for words of a certain letter, even if there aren't any. This alternative avoids the auxiliary function needed for the first alternative but adds complexity to the design of `most-frequent.v2`. **End**

Note on Intermediate Data and Deforestation This second version computes the desired result via the creation of a large intermediate data structure that serves no real purpose other than that its parts are counted. On occasion, the programming language (implementation) eliminates them automatically by *fusing* the two functions into one, a transformation on programs that is also called *deforestation*. When you know that the language does not deforest programs, consider eliminating such data structures if the program does not process data fast enough.

13.2 Real-world Data: iTunes

Apple's iTunes software is widely used to collect music, videos, tv shows, and so on. You may wish to analyze the information that your iTunes application gathers. It is actually quite easy to extract its database. Select the application's **File** menu, choose **Library** and then **Export**—and voilà you can export a so-called XML representation of the iTunes information. Processing XML is covered in some depth by [The Commerce of XML](#); here we rely on the `2htdp/itunes` library to get hold of the information. Specifically, the library enables you to retrieve the music tracks that your iTunes library contains.

While the details vary, an iTunes library maintains some of the following kind of information for each music track:

- *Track ID*, a unique identifier for the track with respect to your library, example: `442`
- *Name*, the title of the track, example: `Wild Child`
- *Artist*, the producing artists, example: `Enya`
- *Album*, the title of the album to which it belongs, example: `A Day Without Rain`
- *Genre*, the music genre to which the track is assigned, example: `New Age`
- *Kind*, the encoding of the music, example: `MPEG audio file`
- *Size*, the size of the file, example: `4562044`
- *Total Time*, the length of the track in milliseconds, example: `227996`
- *Track Number*, the position of the track within the album, example: `2`
- *Track Count*, the number of tracks on the album, example: `11`
- *Year*, the year of release, example: `2000`
- *Date Added*, when the track was added to your iTunes, example: `2002-7-17 3:55:14`
- *Play Count*, how many times it was played, example: `20`
- *Play Date*, when the track was last played, example: `3388484113` Unix seconds
- *Play Date UTC*, when the track was last played, example: `2011-5-17 17:35:13`

Some tracks may come with less information, some with more.

In addition to the `2http/batch-io` library, this section relies on the `2htdp/itunes` library.

As always, the first task is to choose a BSL data representation for this information. In this section, we use **two** representations for music tracks: a structure-based one and another based on lists. While the former records a fixed number of attributes per track and only if all information is available, the latter comes with whatever information is available represented as data. Each serves particular uses well; for some uses, both representations are useful.

; the `2htdp/itunes` library [documentation, part I](#):

```

; LTracks is one of:
; - '()
; - (cons Track LTracks)

(define-struct track (name artist album time track# added play# played))
; Track is (make-track String String String N N Date N Date)
; interpretation An instance records in order: the track's title, its
; producing artist, to which album it belongs, its playing time in
; milliseconds, its position with the album, the date it was added,
; how often it has been played, and the date when it was last played

; Any Any Any Any Any Any Any -> Track or #false
; creates an instance of Track if the inputs belong to the proper classes
; otherwise it produces #false.
(define (create-track name artist album time track# added play# played)
  ...)

(define-struct date (year month day hour minute second))
; Date is (make-date N N N N N N)
; interpretation An instance records six pieces of information: the
; date's year, month (between 1 and 12 inclusive), day (between 1
; and 31), hour (between 0 and 23), minute (between 0 and 59), and
; second (also between 0 and 59).

; Any Any Any Any Any Any -> Date or #false
; creates an instance of Date if the inputs belong to the proper classes
; otherwise it produces #false.
(define (create-date y mo day h m s)
  ...)

; String -> LTracks
; creates a list of tracks representation for all tracks in file-name,
; which must an XML export from an iTunes library
(define (read-itunes-as-tracks file-name)
  ...)

```

Figure 59: Representing iTunes tracks as structures

Figure 59 introduces the structure-based representation of tracks as implemented by the *2htdp/itunes* library. The `track` structure type comes with eight fields, each representing a particular property of the track. Most fields contain atomic kinds of data, such as `Strings` and `Ns`; others contain `Dates`, which is a structure type with six fields. The *2htdp/itunes* library exports all predicates and selectors for the `track` and `date` structure types, but in lieu of constructors it provides checked constructors.

The last element of the description of the *2htdp/itunes* library is a function that reads an iTunes XML library description and delivers a list of tracks, `LTracks`. Once you have exported the XML library from your iTunes or that of a friend, you can run the following code snippet to retrieve all the records:

```

(define ITUNES-LOCATION "itunes.xml") ; <- adapt to your chosen name

; LTracks
(define itunes-tracks (read-itunes-as-tracks ITUNES-LOCATION))

```

Save the snippet in the same folder as your iTunes XML library. Remember that you don't

really wish to use `itunes-tracks` for examples; it is way too large for that. Indeed, it may be so large that reading the file every time you run your BSL program in DrRacket will take a lot of time. You may therefore wish to comment out this second line while you design functions. Uncomment it only when you wish to compute information about your iTunes collection.

Exercise 201. While the important data definitions are already provided, the first step of the design recipe is still incomplete. Make up examples of [Dates](#), [Tracks](#), and [LTracks](#). These examples come in handy for the following exercises as inputs.

Exercise 202. Design the function `total-time`, which consumes an [LTracks](#) and produces the total amount of play time. Once you have completed the design, compute the total play time of your iTunes collection.

Exercise 203. Design `select-all-album-titles`. The function consumes an [LTracks](#) and produces the list of album titles as a [List-of-strings](#).

Also design the function `create-set`. It consumes a [List-of-strings](#) and constructs one that contains every [String](#) from the given list exactly once. **Hint** If [String](#) `s` is at the front of the given list and occurs in the rest of the list, too, `create-set` does not keep `s`.

Finally design `select-album-titles/unique`, which consumes an [LTracks](#) and produces a list of unique album titles. Use this function to determine all album titles in your iTunes collection and also find out how many distinct albums it contains.

Exercise 204. Design `select-album`. The function consumes the title of an album and an [LTracks](#). It extracts from the latter the list of tracks that belong to the given album.

Exercise 205. Design `select-album-date`. The function consumes the title of an album, a date, and an [LTracks](#). It extracts from the latter the list of tracks that belong to the given album and have been played after the given date. **Hint** You must design a function that consumes two [Dates](#) and determines whether the first occurs before the second.

Exercise 206. Design `select-albums`. The function consumes an [LTracks](#). It produce a list of [LTracks](#), one per album. Each album is uniquely identified by its title and shows up in the result only once. **Hints** You want to use some of the solutions of the preceding exercises. Also, the function that groups consumes two lists: the list of album titles and the list of tracks; it considers the latter as atomic until it is handed over to an auxiliary function. See [exercise 198](#).

Terminology Functions such as `select-album-titles`, `select-album`, and `select-album-date` are representative of so-called *database queries*. See [Project: Database](#) for more concepts from this world. **End**

```
; the 2htdp/itunes library documentation, part I:

; LLists is one of:
; - '()
; - (cons LAssoc LLists)

; LAssoc is one of:
; - '()
; - (cons Association LAssoc)
;
; Association is (cons String (cons BSDN '()))
```

```
; BSDN is one of: Boolean, Number, String, or Date

; String -> LLists
; creates a list of lists representation for all tracks in file-name,
; which must be an XML export from an iTunes library
(define (read-itunes-as-lists file-name)
  ...)
```

Figure 60: Representing iTunes tracks as lists

Figure 60 shows how the *2htdp/itunes* library represents tracks with lists. An **LLists** is a list of track representations, each of which is a list of lists pairing **Strings** with four kinds of values. The **read-itunes-as-lists** function reads an iTunes XML library and produces an element of **LLists**. Hence, if you add the following definitions to your program and save it in the same folder where the iTunes library is stored,

```
(define ITUNES-LOCATION "itunes.xml") ; <- adapt to your chosen name

; LLists
(define list-tracks (read-itunes-as-lists ITUNES-LOCATION))
```

you get access to all tracks as lists.

Exercise 207. Develop examples of **LAssoc** and **LLists**, that is, the list representation of tracks and lists of such tracks.

Exercise 208. Design the function **find-association**. It consumes three arguments: a **String** called **key**; an **LAssoc**; and an element of **Any** called **default**. It produces the first **Association** whose first item is equal to **key** or **default** if there is no such **Association**.

Hint You want to use the BSL+ operation **append**, which consumes two lists and produces the concatenation of the two lists:

```
> (append (list "a" "b" "c") (list "d" "e"))
'("a" "b" "c" "d" "e")
```

Note Read up on **assoc** after you have designed this function.

Exercise 209. Design the function **total-time/list**, which consumes an **LLists** and produces the total amount of play time. **Hint** Solve [exercise 208](#) first.

Once you have completed the design, compute the total play time of your iTunes collection. Compare this result with the time that the **total-time** function from [exercise 202](#) computes. Why is there a difference?

Exercise 210. Design **boolean-attributes**. The function consumes an **LLists** and produces the list of **Strings** that are associated with a boolean attribute. Use **create-set** from [exercise 203](#) to make sure each **String** shows up only once.

Once you have completed the design, determine how many boolean-valued attributes your iTunes library employs for its tracks. Do they make sense?

Note A list-based representation is a bit less organized than a structure-based one. The word *semi-structured* is occasionally used in this context. Such list-representations accommodate properties that show up rarely and thus don't fit the structure type. People often use such representations to explore unknown information and later introduce structures when the format is well-known. Design a function **track-as-struct**, which converts an **LAssoc** to a **Track** when possible. **End**

13.3 Word Games, Composition Illustrated

If you play Scrabble or solve the word puzzles in newspapers, you often confront the following:

Sample Problem: Given a word, find (all possible) words that are made up from some letters. For example “cat” also spells “act.”

Let’s work through an example. Suppose you are given “dear.” There are twenty-four possible arrangements of the four letters:

ader	aedr	aerd	adre	arde	ared
daer	eadr	eard	dare	rade	raed
dear	edar	erad	drae	rdae	read
dera	edra	erda	drea	rdea	reda

In this list, there are three legitimate words: “read,” “dear,” and “dare.”

See [Real-world Data: Dictionaries](#) for dealing with real-world dictionaries.

A systematic enumeration of all possible arrangements is clearly a task for a program as is the search in an English-language dictionary. This section covers the design of the latter, leaving the solution of the second problem to the next section. By separating the two, this first section can focus on the high-level ideas of systematic program design.

Let’s imagine for a moment how we might solve the problem by hand. If you had enough time, you might enumerate all possible arrangements of all letters in a given word and then just pick those variants that also occur in a dictionary. Clearly, a program can proceed in this way too, and this suggests a natural design by composition but, as always, we proceed systematically and start by choosing a data representation for our inputs and outputs.

At least at first glance, it is natural to represent words as [Strings](#) and the result as a list of words or [List-of-strings](#). Based on this choice, we can formulate a contract and purpose statement:

```
; String -> List-of-strings
; find all words that the letters of some given word spell

(define (alternative-words s)
  ...)
```

Next, we need some examples. If the given word is “cat,” we are dealing with three letters: *c*, *a*, and *t*. Some playing around suggests six arrangements of these letters: *cat*, *cta*, *tca*, *tac*, *act*, and *atc*. Two of these are actual words: “cat” and “act.” Because `alternative-words` produces a list of [Strings](#), there are two ways to represent the result: `(list "act" "cat")` and `(list "cat" "act")`. Fortunately, BSL comes with a way to say the function returns one of two possible results:

```
(check-member-of (alternative-words "cat")
                 (list "act" "cat")
                 (list "cat" "act"))
```

Stop! Read up on [check-member-of](#) in the documentation.

Working through this example exposes two problems:

- The first one is about testing. Suppose we had used the word “rat” for which there are three alternatives: “rat,” “tar,” and “art.” In this case, we would have to formulate six lists, each of which might be the result of the function. For a word like “dear” with four possible alternatives, formulating a test would be even harder.
- The second problem concerns the choice of word representation. Although `String` looks natural at first, the examples clarify that some of our functions must view words as sequences of letters, with the possibility of rearranging them at will. It is possible to rearrange the letters within a `String`, but lists of letters are obviously better suited for this purpose.

Let’s deal with these problems one at a time, starting with tests.

To make it all concrete, assume we wish to formulate a test for `alternative-words` when given `"rat"`. From the above, we know that the result must contain `"rat"`, `"tar"`, and `"art"`, but we cannot know in which order these words show up in the result.

See [Intermezzo: BSL](#) for the very basic facts about `check-satisfied`. [Exercise 188](#) explains how to use this construct.

In this situation, `check-satisfied` comes in handy. We can use it with a function that checks whether a list of `Strings` contains our three `Strings`:

```
; List-of-strings -> Boolean
(define (all-words-from-rat? w)
  (and (member? "rat" w)
       (member? "art" w)
       (member? "tar" w)))
```

And with this function it is straightforward to formulate a test for `alternative-words`:

```
(check-satisfied (alternative-words "rat") all-words-from-rat?)
```

Note on Data and Design What this discussion suggests is that `alternative-words` constructs a set, not a list. For a detailed discussion of the differences, see [A Note on Lists and Sets](#). Here it suffices to know that sets represents collections of values **without** regard to the ordering of the values or how often these values occur. When a language comes without support for data representations of sets, programmers tend to resort to a close alternative, such as the `List-of-strings` representation here. As programs grow, this choice may haunt programmers but addressing this kind of problem is subject of the second book. **End**

```
; String -> List-of-strings
; find all words that the letters of some given word spell

(check-member-of (alternative-words "cat"))
  (list "act" "cat")
  (list "cat" "act"))

(define (all-words-from-rat? w)
  (and (member? "rat" w)
       (member? "art" w))
```

```

(member? "tar" w))

(check-satisfied (alternative-words "rat") all-words-from-rat?)

(define (alternative-words s)
  (in-dictionary (words->strings (arrangements (string->word s)))))

; List-of-words -> List-of-strings
; turn all Words in low into Strings
(define (words->strings low)
  empty)

; List-of-strings -> List-of-strings
; pick out all those Strings that occur in the dictionary
(define (in-dictionary los)
  empty)

```

Figure 61: Finding alternative words

For the problem with a word representation, we punt to the next section. Specifically, we say that the next section introduces (1) a data representation for `Words` suitable for re-arranging letters, (2) a data definition for `List-of-words` and (3) a function that maps a `Word` to a `List-of-words`, meaning a list of all possible re-arrangements:

```

; A Word is ...

; A List-of-words is ...

; Word -> List-of-words
; find all re-arrangements of word
(define (arrangements word)
  (list word))

```

Exercise 211. The above leaves us with two additional “wishes:” a function that consumes a `String` and produces its corresponding `Word` and a function for the opposite direction. Here are the signatures and purpose statements:

```

; String -> Word
; convert s to the chosen word representation
(define (string->word s)
  ...)

; Word -> String
; convert w to a string
(define (word->string w)
  ...)

```

Look up the data definition for `Word` in the next section and complete the definitions of `string->word` and `word->string`. Hint You may wish to look in the list of functions that BSL provides.

With those two small problems out of the way, we return to the design of `alternative-words`. We now have: (1) a signature, (2) a purpose statement, (3) examples and test, (4) an insight concerning our choice of data representation, and (5) an idea of how to decompose the problem into two major steps.

So, instead of creating a template, we write down the composition we have in mind:

```
| (in-dictionary (arrangements s))
```

The expression says that, given a word `s`, we use `arrangements` to create a list of all possible re-arrangements of the letters and `in-dictionary` to select those re-arrangements that also occur in a dictionary.

Stop! Formulate signatures and purpose statements for the two functions.

What this expression fails to capture is the fourth point, the decision not to use plain strings to re-arrange the letters. Before we hand `s` to `arrangements`, we need to convert it into a word. Fortunately, [exercise 211](#) asks for just such a function:

```
| (in-dictionary (... (arrangements (string->word s))))
```

Similarly, we need to convert the resulting list of words to a list of strings. While [exercise 211](#) asks for a function that converts a single function, here we need a function that deals with lists of words. It is time to make another wish:

```
| (in-dictionary (words->strings (arrangements (string->word s))))
```

Stop! What is the signature for `words->strings` and what is its purpose?

[Figure 61](#) collects all the pieces into a single function definition and a concrete wish list. The following exercises asks you to design the remaining functions.

Exercise 212. Complete the design of the `words->strings` function specified in [figure 61](#).

Hint Use your solution to [exercise 211](#).

Exercise 213. Complete the design of the `in-dictionary` function specified in [figure 61](#).

Hint See [Real-world Data: Dictionaries](#) for how to read a dictionary.

13.4 Word Games, the Heart of the Problem

The goal of this section is to design the function `arrangements`, a function that consumes a [Word](#) and produces a list of the word's letter-by-letter rearrangements. This extended exercise reinforces the need for deep wish lists, that is, a list of desired functions that seems to grow with every function you finish.

The mathematical term is *permutations*.

As mentioned in the preceding section, [Strings](#) could serve as a representation of words, but a [String](#) is an atomic piece of data in BSL+ and the very fact that `arrangements` needs to rearrange the letters in words suggests we need a compound piece of data here.

Our chosen data representation of a word is therefore a list of [1Strings](#) where each item in the input represents a letter:

```
; A Word is either
; - '()
; - (cons 1String Word)
; interpretation a String as a list of single Strings (letters)
```

For simplicity, we equate letters and [1Strings](#).

Exercise 214. Write down the data definition for *List-of-words*. Systematically make up

examples of **Words** and **List-of-words**. Finally, formulate the functional example from above with **check-expect**. Instead of working with the full example, you may wish to start with a word of just two letters, say "d" and "e".

The template of arrangements is that of a list-processing function:

```
; Word -> List-of-words
; creates a list of all rearrangements of the letters in w
(define (arrangements w)
  (cond
    [(empty? w) ...]
    [else (... (first w) ... (arrangements (rest w)) ...)]))
```

because **Word** is described by a list-shaped data definition.

In preparation of the fifth step, let us look at each **cond**-line in the template:

1. If the input is '(), there is only one possible rearrangement of the input: the '() word.
Hence the result is (**list** '()), the list that contains the empty list as the only item.
2. Otherwise there is a first letter in the word, and (**first** w) is that letter and the recursion produces the list of all possible rearrangements for the rest of the word. For example, if the list is

```
(list "d" "e" "r")
```

then the recursion is (**arrangements** (**list** "e" "r")). It will produce the result

```
(cons (list "e" "r")
      (cons (list "r" "e")
            '()))
```

To obtain all possible rearrangements for the entire list, we must now insert the first item, "d" in our case, into all of these words between all possible letters and at the beginning and end.

Our analysis suggests that we can complete **arrangements** if we can somehow insert one letter into all positions of many different words. The last aspect of this task description implicitly mentions lists and, following the advice of this chapter, calls for an auxiliary function. Let us call this function **insert-everywhere/in-all-words** and let us use it to complete the definition of **arrangements**:

```
(define (arrangements w)
  (cond
    [(empty? w) (list '())]
    [else (insert-everywhere/in-all-words (first w)
                                         (arrangements (rest w))))]))
```

Exercise 215. Design **insert-everywhere/in-all-words**. It consumes a **1String** and a list of words. The result is a list of words like its second argument, but with the first argument inserted at the beginning, between all letters, and at the end of all words of the given list.

Start with a complete wish list entry. Supplement it with tests for empty lists, a list with a one-letter word and another list with a two-letter word, etc. Before you continue, study the following three hints carefully.

Hints (1) Reconsider the example from above. It says that “d” needs to be inserted into the words (`list "e" "r"`) and (`list "r" "e"`). The following application is therefore one natural candidate for an example and unit test:

```
(insert-everywhere/in-all-words "d"
  (cons (list "e" "r")
    (cons (list "r" "e")
      '())))
```

Keep in mind that the second input corresponds to the sequence of (partial) words “er” and “re”.

(2) You want to use the BSL+ operation `append`, which consumes two lists and produces the concatenation of the two lists:

```
> (append (list "a" "b" "c") (list "d" "e"))
'("a" "b" "c" "d" "e")
```

the development of functions like `append` is the subject of section [Simultaneous Processing](#).

(3) This solution of this exercise is a series of functions. Patiently stick to the design recipe and systematically work through your wish list.

Exercise 216. Integrate arrangements with the partial program from [Word Games, Composition Illustrated](#). After making sure that the entire suite of tests passes, run it on some of your favorite examples.

13.5 Feeding Worms

Worm—also known as *Snake*—is one of the oldest computer games. When the game starts, a worm and a piece of food appear. The worm is moving toward a wall. Don’t let it run into the wall; otherwise the game is over. Instead, use the arrow keys to control the worm’s movements.

The goal of the game is to have the worm eat as much food as possible. As the worm eats the food, it becomes longer; more and more segments appear. Once a piece of food is digested, another piece appears. The worm’s growth endangers the worm itself, though. As it grows large enough, it can run into itself and, if it does, the game is over, too.

Figure 62: Playing Worm

Figure 62 displays a sequence of screen shots that illustrates how the game works in practice. On the left, you see the initial setting. The worm consists of a single red segment, its head. It is moving toward the food, which is displayed as a green disk. The screen shot in the center shows a situation when the worm is about to eat some food. In the right-most screen shot the worm has run into the right wall. The game is over; the player scored 11 points.

The following exercises guide you through the design and implementation of a Worm game. Like [Structures in Lists](#), these exercises illustrate how to tackle a non-trivial problem via iterative refinement. That is, you don't design the entire interactive program all at once but in several stages, called *iterations*. Each iteration adds details and refines the program —until it satisfies you or your customer. If you aren't satisfied with the outcome of the exercises, feel free to create variations.

Exercise 217. Design an interactive GUI program that continually moves a one-segment worm and enables a player to control the movement of the worm with the four cardinal arrow keys. Your program should use a red disk to render the one-and-only segment of the worm. For each clock tick, the worm should move a diameter.

Hints (1) Re-read section [Designing World Programs](#) to recall how to design world programs. When you define the `worm-main` function, use the rate at which the clock ticks as its argument. See the documentation for `on-tick` on how to describe the rate. (2) When you develop a data representation for the worm, contemplate the use of two different kinds of representations: a physical representation and a logical one. The **physical** representation keeps track of the actual **physical position** of the worm on the canvas; the **logical** one counts how many (widths of) segments the worm is from the left and the top. For which of the two is it easier to change the physical appearances (size of worm segment, size of game box) of the “game?”

Exercise 218. Modify your program from [exercise 217](#) so that it stops if the worm has run into the walls of the world. When the program stops because of this condition, it should render the final scene with the text "worm hit border" in the lower left of the world scene. **Hint** You can use the `stop-when` clause in [big-bang](#) to render the last world in a special way. Challenge: Show the worm in this last scene as if it were on its way out of the box.

Exercise 219. Develop a data representation for worms with tails. A worm's tail is a possibly empty sequence of “connected” segments. Here “connected” means that the coordinates of a segment differ from those of its predecessor in at most one direction and, if rendered, the two segments touch. To keep things simple, treat all segments—head and tail segments—the same.

Then modify your program from [exercise 217](#) to accommodate a multi-segment worm. Keep things simple: (1) your program may render all worm segments as red disks; (2) one way to think of the worm's movement is to add a segment in the direction in which it is moving and to delete the last segment; and (3) ignore that the worm may run into the wall or into itself.

Exercise 220. Re-design your program from exercise [exercise 219](#) so that it stops if the worm has run into the walls of the world or into itself. Display a message like the one in [exercise 218](#) to explain whether the program stopped because the worm hit the wall or because it ran into itself.

Hints (1) To determine whether a worm is going to run into itself, check whether the position of the head would coincide with one of its old tail segments if it moved. (2) Read up on the BSL+ primitive `member?`.

Exercise 221. Equip your program from [exercise 220](#) with food. At any point in time, the box should contain one piece of food. To keep things simple, a piece of food is of the same size as worm segment. When the worm's head is located at the same position as the food, the worm eats the food, meaning the worm's tail is extended by one segment. As the piece of food is eaten, another one shows up at a different location.

Adding food to the game requires changes to the data representation of world states. In

addition to the worm, the states now also include a representation of the food, especially its current location. A change to the game representation suggests new functions for dealing with events, though these functions can reuse the functions for the worm (from [exercise 220](#)) and their test cases. It also means that the tick handler must not only move the worm; in addition it must manage the eating process and the creation of new food.

Your program should place the food randomly within the box. To do so properly, you need a design technique that you haven't seen before—so-called generative recursion, which is introduced in [Generative Recursion](#)—so we provide function definitions:

For the workings of `random`, read the manual or [exercise 101](#).

```
; Posn -> Posn
; ???

(define (food-create p)
  (food-check-create p (make-posn (random MAX) (random MAX)))))

; Posn Posn -> Posn
; generative recursion
; ???
(define (food-check-create p candidate)
  (if (equal? p candidate) (food-create p) candidate))

; Posn -> Boolean
; use for testing only
(define (not-equal-1-1? p)
  (not (and (= (posn-x p) 1) (= (posn-y p) 1))))
```

Before you use them, however, explain how these functions work—assuming MAX is greater than 1—and then formulate purpose statements.

Hints (1) One way to interpret “eating” is to say that the head moves where the food used to be located and the tail grows by one segment, inserted where the head used to be. Why is this interpretation easy to design as a function? (2) We found it useful to add a second parameter to the `worm-main` function for this last step, namely, a **Boolean** that determines whether `big-bang` displays the current state of the world in a separate window; see the documentation for `state` on how to ask for this information.)

Once you have finished this last exercise, you now have a finished worm game. If you modify your `worm-main` function so that it returns the length of the final worm’s tail, you can use the “Create Executable” menu in DrRacket to turn your program into something that anybody can launch, not just someone that knows about BSL+ and programming.

You may also wish to add extra twists to the game, to make it really your game. We experimented with funny end-of-game messages; having several different pieces of food around; with extra obstacles in the room; and a few other ideas. What can you think of?

13.6 Simple Tetris

Tetris is another game from the early days of software. Since the design of a full-fledged

Tetris game demands a lot of labor with only marginal profit, this section focuses on a simplified version. If you feel ambitious, look up how Tetris really works and design a full-fledged version.

Figure 63: Simple Tetris

In our simplified version, the game starts with individual blocks dropping from the top of the scene. Once it lands on the ground, it comes to a rest and another block starts dropping down from some random place. A player can control the dropping block with the “left” and “right” arrow keys. Once a block lands on the floor of the canvas or on top of some already resting block, it comes to rest and becomes immovable. In a short time, the blocks stack up and, if a stack of blocks reaches the ceiling of the canvas, the game is over. Naturally the objective of this game is to land as many blocks as possible. See [figure 63](#) for an illustration of the idea.

Given this description, we can turn to the design guidelines for interactive programs from [Designing World Programs](#). It calls for separating constant properties from variable ones. The former can be written down as “physical” and graphical constants; the latter suggest the data that makes up all possible states of the world, i.e., the simple Tetris game here. So here are some examples:

- The width and the height of the game are fixed as are the blocks. In terms of BSL+, you want definitions like these:

```
; physical constants
(define WIDTH 10) ; the maximal number of blocks horizontally

; graphical constants
(define SIZE 10) ; blocks are square
(define BLOCK ; they are rendered as red squares with black rims
  (overlay (rectangle (- SIZE 1) (- SIZE 1) "solid" "red")
           (rectangle SIZE SIZE "outline" "black")))

(define SCENE-SIZE (* WIDTH SIZE))
```

Explain the last line before you read on.

- The “landscape” of resting blocks and the moving block differ from game to game and from clock tick to clock tick. Let us make this more precise. The appearances of the blocks remains the same; their positions differ.

We are now left with the central problem of designing a data representation for the dropping blocks and the landscapes of blocks on the ground. When it comes to the dropping block, there are again two

possibilities: one is to choose a “physical” representation, another would be a “logical” one. The **physical**

See [exercise 217](#) for a related design decision.

representation keeps track of the actual physical **position** of the blocks on the canvas; the **logical** one counts how many block widths a block is from the left and the top. When it

comes to the resting blocks, there are even more choices than for individual blocks: a list of physical positions, a list of logical positions, a list of stack heights, etc.

In this section we choose the data representation for you:

```
; A Tetris is (make-tetris Block Landscape)
; A Landscape is one of:
; - '()
; - (cons Block Landscape)
; Block is (make-block N N)

; interpretation given (make-tetris (make-block x y) (list b1 b2 ...))
; (x,y) is the logical position of the dropping block, while
; b1, b2, etc are the positions of the resting blocks
; A logical position (x,y) determines how many SIZES the block is
; from the left-x-and from the top-y.
```

This is what we dubbed the logical representation, because the coordinates do not reflect the physical location of the blocks, just the number of block sizes they are from the origin. Our choice implies that x is always between 0 and WIDTH (exclusive) and that y is between 0 and HEIGHT (exclusive), but we ignore this knowledge.

Exercise 222. When you are presented with a complex data definition—like the one for the state of a Tetris game—you start by creating instances of the various data collections. Here are some suggestive names for examples you can later use for functional examples:

```
(define landscape0 ...)
(define block-dropping ...)
(define tetris0 ...)
(define tetris0-drop ...)
...
(define block landed (make-block 0 (- HEIGHT 1)))
...
(define block-on-block (make-block 0 (- HEIGHT 2)))
```

Design the program `tetris-render`, which turns a given instance of `Tetris` into an `Image`. Use DrRacket's interaction area to develop the expression that renders some of your (extremely) simple data examples. Then formulate the functional examples as unit tests and the function itself.

Exercise 223. Design the interactive program `tetris-main`, which displays blocks dropping in a straight line from the top of the canvas and landing on the floor or on blocks that are already resting. The input to `tetris-main` should determine the rate at which the clock ticks. See the documentation of `on-tick` for how to specify the rate.

To discover whether a block landed, we suggest you drop it and check whether it is on the floor or it overlaps with one of the blocks on the list of resting block. **Hint** Read up on the BSL+ primitive `member?`.

When a block lands, your program should immediately create another block that descends on the column to the right of the current one. If the current block is already in the right-most column, the next block should use the left-most one. Alternatively, define the function `block-generate`, which randomly selects a column different from the current one; see [exercise 221](#) for inspiration.

Exercise 224. Modify the program from [exercise 223](#) so that a player can control the left

or right movement of the dropping block. Each time the player presses the "`left`" arrow key, the dropping block should shift one column to the left unless it is in column 0 or there is already a stack of resting blocks to its left. Similarly, each time the player presses the "`right`" arrow key, the dropping block should move one column to the right if possible.

Exercise 225. Equip the program from [exercise 224](#) with a `stop-when` clause. The game ends when one of the columns contains HEIGHT blocks.

It turns out that the design of the `stop-when` handler is complex. So here are hints: (1) Design a function that consumes a natural number i and creates a column of i blocks. Use the function to define a [Landscape](#) for which the game should stop. (2) Design a function that consumes a [Landscape](#) and a natural number x_0 . The function should produce the list of blocks that have the x -coordinate x_0 . (3) Finally, design a function that determines whether the `length` of any column in some given [Landscape](#) is HEIGHT.

Once you have solved [exercise 225](#) you have a bare-bones Tetris game. You may wish to polish it a bit before you show it to your friends. For example, the final canvas could show a text that says how many blocks the player was able to stack up. Or every canvas could contain such a text. The choice is yours.

13.7 Full Space War

[Structure in the World](#) and [Itemizations and Structures](#) introduces a space invader game with little action; the player can merely move the ground force back and forth. [Lists and World](#) enables the player to fire as many shots as desired. This section poses the final exercises in this series. Specifically it is about making the shots interact with the UFO and more.

As always, a UFO is trying to land on earth. The player's task is to prevent the UFO from landing. To this end, the game comes with a tank that may fire an arbitrary number of shots. When one of these shots comes close enough to the UFO's center of gravity, the game is over and the player won. If the UFO comes close enough to the ground, the player lost.

Exercise 226. Use the lessons learned from the preceding two sections and design the game extension slowly, adding one feature of the game after another. Always use the design recipe and rely on the design guidelines for auxiliary functions. If you like the game, add other features: show a running text; equip the UFO with charges that can harm or eliminate the tank; create an entire fleet of attacking UFOs; and above all, use your imagination.

If you don't like UFOs and tanks shooting at each other, let's use the same ideas to produce a similar, civilized game.

Exercise 227. Design a fire-fighting game.

The game is set in the western states where fires rage through vast forests. It simulates an airborne fire-fighting effort. Specifically, the player acts as the pilot of an airplane that drops loads of water on fires on the ground. The player controls the plane's horizontal movements and the release of water loads.

Your game software starts fires randomly anywhere on the ground. You may wish to limit the number of fires that may burn at any point in time, making them a function of how many fires are currently burning. The purpose of the game is to extinguish all fires with a limited number of water drops.

Use an iterative design approach as illustrated in this chapter to create this game.

13.8 Finite State Machines

Finite state machines (FSMs) and regular expressions are ubiquitous elements of programming problems. As [Finite State Worlds](#) explains, state machines are one possible way to understand and even design world programs. Conversely, [exercise 112](#) shows how to design world programs that implement a FSM and thus check whether a player presses a specific series of key strokes.

As you may also recall, a finite state machine is equivalent to a regular expression. Hence, computer scientists tend to say that a FSM accepts the key strokes that match a particular regular expression, like this one

$$a \ (b|c)^* \ d$$

from [exercise 112](#). Now if you wanted a program that recognizes a different pattern, say,

$$a \ (b|c)^* \ a$$

you would just modify the program appropriately. The two programs would resemble each other, and if you were to repeat this exercise for several different regular expressions, you would end up with a family of similar programs.

A natural thought is to look for a general solution, that is, a world program that consumes a **data description of a FSM** and then recognizes whether a player presses a matching sequence of keys. This section presents the design of just such a world program, though a greatly simplified one. In particular, the FSMs come without initial or final states and they ignore the actual key strokes; instead they transition from one state to another whenever **any** key is pressed. Furthermore, we require that the states are color strings. That way, the FSM-interpreting program can simply display the current state as a color.

Note on Design Choices Here is another attempt to generalize the program:

Sample Problem: Design a program that interprets a given FSM on a specific list of [KeyEvents](#). That is, the program consumes a data representation of a FSM and a string. Its result is `#true` if the string matches the regular expression that corresponds to the FSM; otherwise it is `#false`.

As it turns out, however, you **cannot design** this program with the principles of the first two parts. Indeed, solving this problem has to wait until [Algorithms that Backtrack](#); see [exercise 478. End](#)

```

; A FSM is one of:
;   - '()
;   - (cons Transition FSM)

(define-struct transition [current next])
; A Transition is
;   (make-transition FSM-State FSM-State)

; FSM-State is a String that specifies a color.

; interpretation A FSM represents the transitions that a
; finite state machine can take from one state to another

```

; in reaction to key strokes

Figure 64: Representing and interpreting finite state machines in general

The simplified problem statement dictates a number of points, including the need for a data definition for the representation of FSMs, the nature of its states, and their appearance as an image. Figure 64 collects this information. It starts with a data definition for **FSMs**. As you can see, a **FSM** is just a list of **Transitions**. We must use a list because we want our world program to work with any FSM and that means a finite, but arbitrary large number of states. Each **Transition** combines two states in a structure: the **current state** and the **next state**, that is, the one that the machine transitions to when the player presses a key. The final part of the data definition says that a state is just the name of a color.

Exercise 228. Design `state=?`, an equality predicate for states.

Since this definition is complex, we follow the design recipe and create an example:

```
(define fsm-traffic
  (list (make-transition "red" "green")
        (make-transition "green" "yellow")
        (make-transition "yellow" "red")))
```

You probably guessed that this transition table describes a traffic light. Its first transition tells us that the traffic light jumps from "red" to "green", the second one represents the transition from "green" to "yellow", and the last one is for "yellow" to "red".

Exercise 229. The BW Machine is a FSM that flips from black to white and back to black for every key event. Formulate a data representation for the BW Machine.

Clearly, the solution to our problem is a world program:

```
; FSM -> ???
; match the keys pressed by a player with the given FSM
(define (simulate a-fsm)
  (big-bang ...
    [to-draw ...]
    [on-key ...]))
```

It is supposed to consume a **FSM** but we have no clue what the program is to produce. We call the program **simulate** because it acts like the given **FSM** in response to a players key strokes.

Let's follow the design recipe for world programs anyway and see how far it takes us. The design recipe tells us to differentiate between those things in the “real world” that change and those that remain the same. While the **simulate** function consumes an instance of **FSM**, we also know that this **FSM** does not change. What changes is the current state of the machine.

This analysis suggests the following data definition

```
; A SimulationState.v1 is a FSM-State.
```

According to the design recipe, this data definition implies a wish list with two entries and completes the main function:

The `empty-image` constant represents an “invisible” image. It is a good default value for writing down the headers of rendering

functions.

```
; FSM -> SimulationState.v1
; match the keys pressed by a player with the given FSM
(define (simulate.v1 fsm0)
  (big-bang initial-state
    [to-draw render-state.v1]
    [on-key find-next-state.v1])))

; SimulationState.v1 -> Image
; renders a world state as an image
(define (render-state.v1 s)
  empty-image)

; SimulationState.v1 -> SimulationState.v1
; finds the next state from a key stroke ke and current state cs
(define (find-next-state.v1 cs ke)
  cs)
```

The sketch raises two questions. First, there is the issue of how the very first `SimulationState.v1` is determined. Currently, the chosen state, `initial-state`, is marked in red to warn you about the issue. Second, the second entry on the wish list must cause some consternation:

How can `find-next-state` possibly find the next state when all it is given is the current state and the representation of a key stroke?

This question rings especially true because, according to the simplified problem statement, the exact nature of the key stroke is completely irrelevant; the FSM transitions to the next state regardless of which key is pressed.

What this second issue exposes is a **fundamental limitation of BSL+**. To appreciate this limitation, we start with a work-around. Basically, the analysis demands that the `find-next-state` function receives not only the current state but also the `FSM` so that it can search the list of transitions and pick the next state. In other words, the state of the world must include both the current state of the `FSM` and the `FSM`:

Alonzo Church and Alan Turing, the first two computer scientists, proved in the 1930s that all programming languages can compute certain functions on numbers. Hence, they argued that all programming languages were equal. The first author of this book [disagrees](#). He distinguishes languages according to **how** they allow programmers to express solutions.

```
(define-struct fs [fsm current])
; A SimulationState.v2 is a structure:
;   (make-fs FSM FSM-State)
```

According to the world design recipe, this change also means that the key event handler must return this combination:

```
; SimulationState.v2 -> Image
; renders a world state as an image
(define (render-state.v2 s)
  empty-image)

; SimulationState.v2 -> SimulationState.v2
```

```
; finds the next state from a key stroke ke and current state cs
(define (find-next-state.v2 cs ke)
  cs)
```

Finally, the main function must now consume two arguments: the **FSM** and its first state. After all, the various **FSMs** that simulate consumes come with all kinds of states; we cannot assume that all of them have the same initial state. Here is the revised function header:

```
; FSM FSM-State -> SimulationState.v2
; match the keys pressed by a player with the given FSM
(define (simulate.v2 a-fsm s0)
  (big-bang (make-fs a-fsm s0)
    [to-draw state-as-colored-square]
    [on-key find-next-state]))
```

Let's return to the example of the traffic-light **FSM**. For this machine, it would be best to apply `simulate` to the representation of the machine and "red":

```
(simulate.v2 fsm-traffic "red")
```

Stop! Why do you think "red" is good for traffic lights? What would be a good starting state for the BW Machine of [exercise 229](#)? Does it matter in this case?

Note on Expressive Power Given the work-around, we can now explain the limitation of BSL. Even though the given **FSM** does not change during the course of the simulation, its description must become a part of the world's state. Ideally, the program should express that the description of the **FSM** remains constant but instead the program must treat the **FSM** as part of the ever-changing state. The reader of a program cannot deduce this fact from the first piece of `big-bang` alone.

The next part of the book resolves this conundrum with the introduction of a new programming language and a specific linguistic construct: ISL and local definitions. For details, see ... [Add Expressive Power](#). End

At this point, we can turn to the wish list and work through its entries, one at a time. The first one, the design of `state-as-colored-square` is so straightforward that we simply provide the complete definition:

```
; SimulationState.v2 -> Image
; renders current world state as a colored square

(check-expect (state-as-colored-square (make-fs fsm-traffic "red"))
  (square 100 "solid" "red"))

(define (state-as-colored-square a-fs)
  (square 100 "solid" (fs-current a-fs)))
```

If you have any doubts, see [Adding Structure](#).

In contrast, the design of the key event handler deserves some discussion. Recall the header material:

```
; SimulationState.v2 KeyEvent -> SimulationState.v2
; finds the next state from a key stroke ke and current state cs
(define (find-next-state a-fs current)
  a-fs)
```

According to the design recipe, the handler must consume a state of the world and a [KeyEvent](#), and it must produce the next state of the world. This articulation of the signature in plain words also guides the design of examples. Here are the first two:

```
(check-expect (find-next-state (make-fs fsm-traffic "red") "n")
                  (make-fs fsm-traffic "green"))
(check-expect (find-next-state (make-fs fsm-traffic "red") "a")
                  (make-fs fsm-traffic "green"))
```

It says that when the current state combines the `fsm-traffic` machine and its "red" state, the result combines the same [FSM](#) with the state "green", regardless of whether the player hit "n" or "a" on the keyboard. Here are two more examples:

```
(check-expect (find-next-state (make-fs fsm-traffic "green") "q")
                  (make-fs fsm-traffic "yellow"))
(check-expect (find-next-state (make-fs fsm-traffic "yellow") "n")
                  (make-fs fsm-traffic "red"))
```

Interpret these example before you move on.

Since the function consumes a structure, we write down a template for structures:

```
(define (find-next-state a-fs ke)
  (... (fs-fsm a-fs) ... (fs-current a-fs) ...))
```

Furthermore, because the desired result is an [SimulationState.v2](#), we can refine the template with the addition of an appropriate constructor:

```
(define (find-next-state a-fs ke)
  (make-fsm ... (fs-fsm a-fs) ... (fs-current a-fs) ...))
```

The examples suggest that the extracted [FSM](#) becomes the first component of the new [SimulationState.v2](#) and that the function really just needs to compute the next state from the current one and the list of [Transitions](#) that make up the given [FSM](#). Because the latter is arbitrarily long, we make up a wish—a `find` function that traverses the list to look for a [Transition](#) whose `current` state is `(fs-current a-fs)`—and finish the definition:

```
(define (find-next-state a-fs ke)
  (make-fsm (fs-fsm a-fs)
            (find (fs-fsm a-fs) (fs-current a-fs))))
```

Here is the precise formulation of the new wish:

```
; FSM FSM-State -> FSM-State
; finds the state matching current in the transition table

(check-expect (find fsm-traffic "red") "green")
(check-expect (find fsm-traffic "green") "yellow")
(check-expect (find fsm-traffic "yellow") "red")
(check-error (find fsm-traffic "black") "not found: black")

(define (find transitions current)
  current)
```

The examples are derived from the examples for `find-next-state`. Stop! Develop a couple of additional examples, then tackle the exercises.

Exercise 230. Complete the design of `find`.

Once all the auxiliary functions are tested, use `simulate` to play with `fsm-traffic` and the representation of the BW Machine from [exercise 229](#).

Our simulation program is intentionally quite restrictive. In particular, you cannot use it to represent finite state machines that transition from one state to another depending on which key a player presses. Given the systematic design, though, you can easily extend the program with additional capabilities.

Exercise 231. Here is a revised data definition for `Transition`:

```
(define-struct ktransition [current key next])
; A Transition.v2 is a structure:
;   (make-ktransition FSM-State KeyEvent FSM-State)
```

Represent the FSM from [exercise 112](#) using lists of `Transition.v2`s; ignore error and final states.

Modify the design of `simulate` so that it deals with key strokes in the appropriate manner now. Follow the design recipe, starting the adaptation of the data examples.

Use the revised program to simulate a run of the FSM from [exercise 112](#) on the following sequence of key strokes: "a", "b", "b", "c", and "d".

Finite state machines do come with initial and final states. When a program that “runs” a finite state machine reaches a final state, it should stop. The final exercise revises the data representation of `FSMs` one more time to introduce these ideas.

Exercise 232. Consider the following data representation for finite state machines:

```
(define-struct fsm [initial transitions final])
(define-struct transition [current key next])
; An FSM.v2 is a structure:
;   (make-fsm FSM-State LOT FSM-State)
; A LOT is one of:
; - '()
; - (cons Transition.v3 LOT)
; A Transition.v3 is a structure:
;   (make-transition FSM-State FSM-State KeyEvent)
```

Represent the FSM from [exercise 112](#) in this context.

Design the function `fsm-simulate`, which accepts an `FSM.v2` and runs it on a player’s key strokes. If the sequence of key strokes force the `FSM.v2` to reach a final state, `fsm-simulate` stops. **Hint** The function uses the `initial` field of the given `fsm` structure to keep track of the current state.

Note on Iterative Refinement These last two exercises introduce the notion of “design by iterative refinement.” The basic idea is that the first program implements only a fraction of the desired behavior, the next one a bit more, and so on. Eventually you end up with a program that exhibits all of the desired behavior or at least enough of it to satisfy a customer. For more details, see [Incremental Refinement](#).

This second part of the book is about the design of programs that deal with arbitrarily large data. As you can easily imagine, software is particularly useful when it is used on information that comes without pre-specified size limits, meaning “arbitrarily large data” is a critical step on your way to becoming a real programmer. In this spirit, we suggest that you take away three lessons:

1. This part **refines the design recipe** so that it can deal with self-references and cross-references in data definitions. The occurrence of the former calls for the design of recursive functions, and the occurrence of the latter calls for auxiliary functions.
2. Complex problems call for a **decomposition** into separate problems. When you decompose a problem, you need two pieces: functions that solve the separate problems and data definitions that compose these separate solutions into a single one. To make sure composition does not fail after you have spent time on the separate programs, you need to formulate your functions wishes together with the required data definitions.

A decomposition-composition design is especially useful when the problem statement implicitly or explicitly mentions auxiliary tasks; when the coding step for a function calls for a traversal of an(other) arbitrarily large piece of data; and—perhaps surprisingly—when a general problem is somewhat easier to solve than the specific one described in the problem statement.

3. **Pragmatics matter.** If you wish to design **big-bang** programs, you need to understand its various clauses and what they accomplish. Or, if your task is to design programs that solve mathematical problems, you better make sure you know which mathematical operations the chosen language and its libraries offer.

While this part mostly focuses on lists as a good example of arbitrarily large data—because they are practically useful in languages such as Haskell, Lisp, ML, Racket, and Scheme—the ideas apply to all kinds of such data: files, file folders, databases, etc.

[Intertwined Data](#) continues the exploration of “large” structured data and demonstrates how the design recipe scales to the most complex kind of data. In the meantime, the next part takes care of an important worry you should have at this point, namely, that a programmer’s work is all about creating the same kind of programs over and over and over again.

v.6.3.0.2

Intermezzo: Quote, Unquote

Lists play a central role in our book and in Racket, which spawned our teaching languages. For the design of programs, it is critical to understand how lists are constructed from first principles; it informs the organizations of the functions in our programs. Routine work with lists calls for a compact notation, however, like the [list](#) function introduced in [The list Function](#).

Be sure to use BSL+ or higher for working through this intermezzo.

Since the late 1950s Lisp-style languages have come with an even more powerful pair of list-creation tools: quotation and anti-quotation. Many programming languages support them now, and the PHP web page design language injected the idea into the commercial world.

This intermezzo gives you a taste of this quotation mechanism. It also introduces *symbols*, a new form of data that is intimately tied to quotation. While this introduction is informal and uses simplistic examples, latter parts of the book illustrate how powerful the idea is with near-realistic sample problems. Come back to this intermezzo if any of these examples cause any trouble.

Quote

Quotation is a short-hand mechanism for writing down a large list easily. Roughly speaking, a list constructed with the [list](#) function can be constructed even more concisely by quoting lists. Conversely, a quoted list abbreviates a construction with [list](#).

Technically, [quote](#) is a keyword for a compound sentence—in the spirit of [Intermezzo: BSL](#)—and it is used like this ([quote](#) [\(1 2 3\)](#)). DrRacket translates this expression to ([list](#) [1 2 3](#)). At this point, you may wonder why we call [quote](#) an abbreviation, because the [quoted](#) expressions looks more complicated than its translation. The key is that `'` is a short-hand for [quote](#). Here are some short examples then:

```
> '(1 2 3)
'(1 2 3)
> '("a" "b" "c")
'("a" "b" "c")
> '#(true "hello world" 42)
'#(true "hello world" 42)
```

As you can see, the use of `'` creates the promised lists, and it does so for plain numbers, strings, and lists that contains a mixed bunch of data. In case you are wondering what ([list](#) [1 2 3](#)) means, re-read [The list Function](#), which shows that this list is short for:

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

So far [quote](#) looks like a small improvement over [list](#) but there is a lot more:

```
> '(("a" 1))
```

```

    ("b" 2)
    ("d" 4))
'(("a" 1) ("b" 2) ("d" 4))

```

With **quote** we cannot only construct lists but nested lists. The amazing point is that it does so with a single keystroke.

To understand how **quote** works, imagine it as a function that traverses the shape it is given. When **'** encounters a plain piece of data—a number, a string, a Boolean, or an image—it disappears. When it sits in front of an open parenthesis, **(**, it inserts **list** to the right of the parenthesis and puts **'** on all the items between **(** and the closing **)**. For example,

'(1 2 3) is short for **(list '1 '2 '3)**

As you already know, **'** disappears from numbers so the rest is easy. Here is an example that creates nested lists:

'(("a" 1) 3) is short for **(list '("a" 1) '3)**

To continue this example, we expand the abbreviation in the first position and drop the **'** from **3**:

(list '("a" 1) '3) is short for **(list (list '"a" '1) 3)**

We leave it to you to wrap up this example, and for practice, we recommend the following examples.

Exercise 233. Eliminate **quote** from the following expressions so that they use **list** instead:

- **'(1 "a" 2 #false 3 "c")**
- **'()**
- and this table-like shape:

```

'(("alan" 1000)
  ("barb" 2000)
  ("carl" 1500)
  ("dawn" 2300))

```

Now eliminate **list** in favor of **cons** where needed. ■

Quasiquote And Unquote

The preceding section should convince you of the advantages of **'** and **quote**. You may even wonder why the book introduces **quote** only now and not right from the start. It seems to greatly facilitate the formulation of test cases that involve lists as well as for keeping track of large collections of data. But all good things come with surprises, and that includes **quote**.

When it comes to program design, it is misleading for beginners to think of lists as **quoted** or even **list**-constructed values. The construction of lists with **cons** is far more illuminating for the step-wise creation of programs than short-hands such as **quote**, which hide the underlying construction. So don't forget to think of **cons** whenever you are stuck

during the design of a list-processing function.

Let us move on then to the actual surprises hidden behind `quote`. Suppose your definitions area contains one constant definition:

```
| (define x 42)
```

Imagine running this program and experimenting with

```
| '(40 41 x 43 44)
```

in the interactions area. What result do you expect? Stop! Try to apply the above rules of `'` for a moment.

Here is the experiment

```
| > '(40 41 x 43 44)
| '(40 41 x 43 44)
```

At this point it is important to remember that DrRacket displays values. Everything on the list is a value, including `'x`. It is a value you have never seen before, namely, a *Symbol*. For our purposes, a symbol looks like a variable names except that it starts with `'` and that a **symbol is a value**. Variables only stand for values; they are not values in and of themselves. Symbols play a role similar to those of strings; they are a great way to represent symbolic information as data. [Intertwined Data](#) illustrates how; for now, we just accept symbols as yet another form of data.

To drive home the idea of symbols, consider a second example:

```
| '(1 (+ 1 1) 3)
```

You might expect that this expression constructs `(list 1 2 3)`. If you use the rules for expanding `'`, however, you discover that

`'(1 (+ 1 1) 3)` is short for `(list '1 '(+ 1 1) '3)`

And the `'` on the second item in this list does not disappear. Instead, it abbreviates the construction of another list so that the entire example comes out as

```
| (list 1 (list '+ 1 1) 3)
```

What this means is that `'+` is a symbol just like `'x`. Just as the latter is unrelated to the variable `x`, the former has no immediate relationship to the function `+` that comes with BSL+. Then again, you should be able to imagine that `'+` could serve as an elegant data representation of the function `+` just as `'(+ 1 1)` could serve as a data representation of the expression `(+ 1 1)`. [Intertwined Data](#) picks up this idea.

In some cases, you do not want to create a nested list. You actually want a true expression in a `quoted` list and you want to evaluate the expression during the construction of the list. For such cases, you want to use `quasiquote`. Like `quote`, `quasiquote` is a keyword for a compound sentence: `(quasiquote (1 2 3))`. And like `quote`, `quasiquote` comes with a shorthand, namely the ``` character, which is the “other” single-quote key on your keyboard.

At first glance, ``` acts just like `'` in that it constructs lists:

```
| > `'(1 2 3)
| '(1 2 3)
| > `("a" "b" "c")
```

```
'("a" "b" "c")
> `(#true "hello world" 42)
'(#true "hello world" 42)
```

The best part about ``` is that you can also *unquote*, that is, you can demand an escape back into the programming language proper inside of a **quasiquoted** list. Let us illustrate the idea with the two examples from above:

```
> `(40 41 ,x 43 44)
'(40 41 42 43 44)
> `(1 ,(+ 1 1) 3)
'(1 2 3)
```

As above, the first interaction assumes an definitions area that contains `(define x 42)`. The best way to understand this syntax is to see it with actual keywords instead of ``` and `,`, shorthands:

```
(quasiquote (40 41 (unquote x) 43 44))
(quasiquote (1 (unquote (+ 1 1)) 3))
```

The rules for expanding a **quasiquoted** and an **unquoted** shape are those of **quote** supplemented with one rule. When ``` appears in front of a parenthesis, it is distributed over all parts between it and the matching closing parenthesis. When it appears next to a basic piece of data, it disappears. When it is in front of some variable name, you get a symbol. And the new rule is that when ``` is immediately followed by **unquote**, both characters disappear:

``(1 ,(+ 1 1) 3)` is short for `(list `1 `,(+ 1 1) `3)`

and

`(list `1 `,(+ 1 1) `3)` is short for `(list 1 (+ 1 1) 3)`

And this is how you get `(list 1 2 3)` as seen above.

From here it is a short step to the production of web pages. Yes, you read correctly—web pages. In principle, web pages are coded in the HTML and CSS programming languages. But nobody writes down HTML programs directly; instead people design programs that produce web pages. Not surprisingly, you can write such functions in BSL+, too, and here is a simplistic example:

```
; String String -> ... deeply nested list ...
; produces a (representation of) a web page with given author and title
(define (my-first-web-page author title)
  `(#html
    (#head
      (#title ,title)
      (#meta ((#http-equiv "content-type")
              (#content "text-html"))))
    (#body
      (#h1 ,title)
      (#p "I, " ,author ", made this page.")))
```

As you can immediately see, this function consumes two strings and produces a deeply nested list—a data representation of a web page.

A second look also shows that the `title` parameter shows up twice in the function body:

once nested in a nested list labeled with '`head`' and once nested in the nested list labeled with '`body`'. The other parameter shows up only once. We consider the nested list a page template, and the parameters are holes in the template, to be filled by useful values. As you can imagine, this template-driven style of creating web pages is most useful when you wish to create many similar pages for a site.

To understand how the function works, we experiment in DrRacket's interaction area. Given your knowledge of `quasiquote` and `unquote`, you should be able to predict what the result of

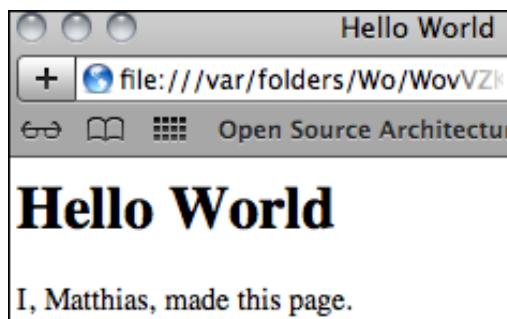
```
(my-first-web-page "Matthias" "Hello World")
```

is. Then again, DrRacket is so fast that it is better to show you the result (left column):

Nested List Representation	Web Page Code (HTML)
<pre>'(html (head (title "Hello World")) (meta ((http-equiv "content-type") (content "text-html")))) (body (h1 "Hello World")) (p "I, " "Matthias" ", made this page.))</pre>	<pre><html> <head> <title> Hello World </title> <meta http-equiv="content- type" content=="text-html" /> </head> <body> <h1> Hello World </h1> <p> I, Matthias, made this page. </p> </body> </html></pre>

The right column of the table contains the equivalent code in HTML. If you were to open this web page in a browser you would see something like this;

You can use the function `show-in-browser` from the `web-io.rkt` library to display the result of functions such as `my-first-web-page` in a web browser.



Note that "`Hello World`" shows up twice again: once in the title bar of the web browser—which is due to the `<title>` specification—and once in the text of the web page.

If this were 1993, you could now sell the above function as a Dot Com company that

generates people's first web page with a simple function call. Alas, in this day and age, it is only an exercise.

Exercise 234. Eliminate `quasiquote` and `unquote` from the following expressions so that they are written with `list` instead:

- `(1 "a" 2 #false 3 "c")

- this table-like shape:

```
| `(("alan" ,( * 2 500))
  ("barb" 2000)
  (,(string-append "carl" " , the great") 1500)
  ("dawn" 2300))
```

- and this second web page:

```
| `(html
  (head
    (title ,title))
  (body
    (h1 ,title)
    (p "A second web page")))
```

where `(define title "ratings")`.

Also write down the nested lists that the expressions produce. ■

Unquote Splice

When `quasiquote` meets `unquote` during the expansion of short-hands, the two annihilate each other:

```
| ` (tr
  , (make-row
    '(1 2 3 4 5)))           is short for      (list 'tr
                                                       (make-row
                                                       (list 1 2 3 4 5)))
```

Thus, whatever `make-row` produces becomes the second item of the list. In particular, if `make-row` produces a list, this list becomes the second item of a list. Say, `make-row` translates the given list of numbers into a list of strings, then the result is

```
| (list 'tr (list "1" "2" "3" "4" "5"))
```

In some cases, however, we may want to splice such a nested list into the outer one, so that for our running example we would get

```
| (list 'tr "1" "2" "3" "4" "5")
```

One way to solve this small problem is to fall back on `cons`. That is, to mix `cons` with `quote`, `quasiquote`, and `unquote`. After all, all of these characters are just short-hands for `consed` lists. Here is what is needed to get the desired result in our example:

```
| (cons 'tr (make-row '(1 2 3 4 5)))
```

Convince yourself that the result is `(list 'tr "1" "2" "3" "4" "5")`.

Since this situation occurs quite often in practice, BSL+ supports one more short-hand mechanism for list creation: `@`, also known as `unquote-splicing` in keyword form. With

this form, it is straightforward to splice a nested list into a surrounding list. For example,

```
| ` (tr ,@(make-row '(1 2 3 4 5)))
```

translates into

```
| (cons 'tr (make-row '(1 2 3 4 5)))
```

which is precisely what we need for our example.

Now consider the problem of creating an HTML table in our nested list representation.

Here is a table of two rows, each row consists of four numbers as strings:

```
'(table ((border "1"))
  (tr (td "1") (td "2") (td "3") (td "4"))
  (tr (td "2.8") (td "-1.1") (td "3.4") (td "1.3")))
```

The first nested lists tells HTML to draw a thin border around each cell in the table; the other two nested lists represent a row each.

In practice, you want to create such tables with arbitrarily wide rows and arbitrarily many rows. For now, we just deal with the first problem, which requires a function that translates lists of numbers into HTML rows:

```
; List-of-numbers -> ... nested list ...
; creates a row for an HTML table from a list of numbers
(define (make-row l)
  (cond
    [(empty? l) '()]
    [else (cons (make-cell (first l)) (make-row (rest l))))]

; Number -> ... nested list ...
; creates a cell for an HTML table from a number
(define (make-cell n)
  `(td ,(number->string n)))
```

Instead of adding examples, we explore the behavior of these functions in DrRacket's interaction area:

```
> (make-cell 2)
'(td "2")
> (make-row '(1 2))
'((td "1") (td "2"))
```

These interactions show the creation of lists that represent a cell and a row, respectively.

To turn such row-lists into actual rows of an HTML table representation, we need to splice them into a list that starts with 'tr':

```
; List-of-numbers List-of-numbers -> ... nested list ...
; creates an HTML table from two lists of numbers
(define (make-table row1 row2)
  `(table ((border "1"))
    (tr ,@(make-row row1))
    (tr ,@(make-row row2))))
```

This function consumes two lists of numbers and creates an HTML table representation. With make-row, it translates the lists into lists of cell representations. With ,@ these lists

are spliced into the table template:

```
> (make-table '(1 2 3 4 5) '(3.5 2.8 -1.1 3.4 1.3))
'(table
  ((border "1"))
  (tr (td "1") (td "2") (td "3") (td "4") (td "5"))
  (tr (td "3.5") (td "2.8") (td "-1.1") (td "3.4") (td "1.3")))
```

This application of `make-table` shows another reason why people prefer to write programs to create web pages from large amounts of data rather than maintain them by hand.

Exercise 235. Eliminate `quasiquote`, `unquote`, and `unquote-splicing` from the following expressions so that they are written with `list` instead:

- ` (0 ,@' (1 2 3) 4)
- this table-like shape:

```
`((("alan" ,(* 2 500))
  ("barb" 2000)
  (,@'(list "carl" " , the great") 1500)
  ("dawn" 2300))
```

- and this third web page:

```
`(html
  (body
    (table ((border "1"))
      (tr ((width "200")) ,@(make-row '( 1 2)))
      (tr ((width "200")) ,@(make-row '(99 65))))))
```

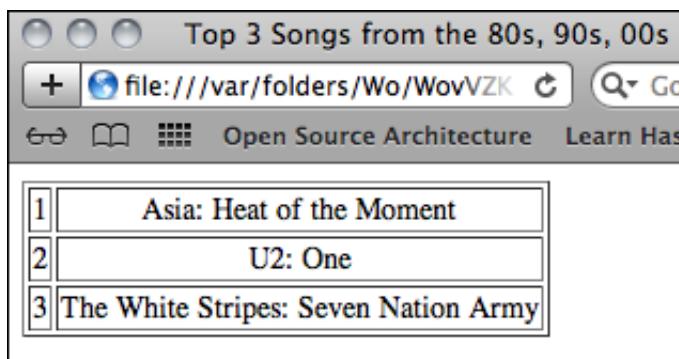
where `make-row` is the function from above.

Also write down the nested lists that the expressions produce. ■

Exercise 236. Create the function `make-ranking`, which consumes a list of ranked song titles and produces a list representation of an HTML table. Consider this example:

```
(define one-list
  '("Asia: Heat of the Moment"
    "U2: One"
    "The White Stripes: Seven Nation Army"))
```

If you apply `make-ranking` to `one-list` and display the result in a browser, you would see something like that:



Hint Although you could design a function that determines the rankings from a list of

strings, we wish you to focus on the creation of tables instead. Thus we supply the following functions:

```
(define (ranking los)
  (reverse (add-ranks (reverse los))))  
  
(define (add-ranks los)
  (cond
    [(empty? los) '()]
    [else (cons (list (length los) (first los))
                (add-ranks (rest los)))])))
```

Before you use these functions, equip them with signatures and purpose statements. Then explore their workings with interactions in DrRacket. [Accumulators](#) expands the design recipe with a way to create simpler functions for computing rankings than `ranking` and `add-ranks`. ■

v.6.3.0.2

III Abstraction

Many of our data definitions and function definitions look alike. For example, the definition for a list of [Strings](#) differs from that of a list of [Numbers](#) in only two places: the names of the classes of data and the words “String” and “Number.” Similarly, a function that looks for a specific string in a list of [Strings](#) is nearly indistinguishable from one that looks for a specific number in a list of [Numbers](#).

Experience shows that these kinds of similarities are problematic. The similarities come about because programmers—physically or mentally—copy code. When programmers are confronted with a problem that is roughly like another one, they copy the solution and modify the new copy to solve the new problem. You will find this behavior both in “real” programming contexts as well as in the world of spreadsheets and mathematical modeling. Copying code, however, means that programmers copy mistakes and the same fix may have to be applied to many copies. It also means that when the underlying data definition is revised or extended, all copies of code must be found and modified in a corresponding way. This process is both expensive and error-prone, imposing unnecessary costs on programming teams.

Good programmers try to eliminate similarities as much as the programming language allows. “Eliminate” implies that programmers write down their first drafts of programs, spot similarities (and other problems), and get rid of them. For the last step, they either *abstract* or use existing (*abstract*) functions. It often takes several iterations of this process to get the program into satisfactory shape.

A program is like an essay. The first version is a draft, and drafts demand editing.

The first half of this part shows how to abstract over similarities in functions and data definitions. Programmers also refer to the result of this process as an *abstraction*, conflating the name of the process and its result. The second half is about the use of existing abstractions and new language elements to facilitate this process. While the examples in this part are taken from the realm of lists, the ideas are universally applicable.

16 Similarities Everywhere

If you have solved only a fraction of the exercises in [Arbitrarily Large Data](#), you know that the solutions look alike. As a matter of fact, the similarities may tempt you to copy the solution of one problem to create the solution for another one. But thou shall not steal code, not even your own. Instead, you must *abstract* over similar pieces of code and this chapter teaches you how to abstract.

Our means of avoiding similarities are specific to “Intermediate Student Language” or ISL for short. Almost all other programming languages provide similar means; in object-oriented languages you may find additional abstraction mechanisms. Regardless, these mechanisms share the basic

In DrRacket, choose “Intermediate Student Language” from the “How to Design Programs” submenu in the “Language” menu.

characteristics spelled out in this chapter, and thus the design ideas explained here apply in other contexts, too.

16.1 Similarities in Functions

The design recipe determines a function's basic organization because the template is created from the data definition without regard to the purpose of the function. Not surprisingly then, functions that consume the same kind of data look alike.

```
; Los -> Boolean
; does l contain "dog"
(define (contains-dog? l)
  (cond
    [(empty? l) #false]
    [else
      (or
        (string=? (first l) "dog")
        (contains-dog?
          (rest l))))]))
```



```
; Los -> Boolean
; does l contain "cat"
(define (contains-cat? l)
  (cond
    [(empty? l) #false]
    [else
      (or
        (string=? (first l) "cat")
        (contains-cat?
          (rest l))))]))
```

Figure 65: Two similar functions

Consider the two functions in [figure 65](#), which consume lists of strings and look for specific strings. The function on the left looks for "dog", the one on the right for "cat". The two functions are nearly indistinguishable. Each consumes lists of strings; each function body consists of a `cond` expression with two clauses. Each produces `#false` if the input is '(); each uses an `or` expression to determine whether the first item is the desired item and, if not, uses recursion to look in the rest of the list. The only difference is the string that is used in the comparison of the nested `cond` expressions: `contains-dog?` uses "dog" and `contains-cat?` uses "cat". To highlight the differences, the two strings are shaded.

Good programmers are too lazy to define several closely related functions. Instead they define a single function that can look for both a "dog" and a "cat" in a list of strings. This general function consumes an additional piece of data—the string to look for—and is otherwise just like the two original functions:

```
; String Los -> Boolean
; determines whether l contains the string s
(define (contains? s l)
  (cond
    [(empty? l) #false]
    [else (or (string=? (first l) s)
              (contains? s (rest l)))]))
```

If you really needed a function such as `contains-dog?` now, you could define it as a one-line function, and the same is true for the `contains-cat?` function. [Figure 66](#) does just that, and you should briefly compare it with [figure 65](#) to make sure you understand how we get from there to here. Best of all, though, with `contains?` it is now trivial to look for any string in a list of strings and there is no need to ever define a specialized function such as `contains-dog?` again.

```
; Los -> Boolean
; does l contain "dog"
```



```
; Los -> Boolean
; does l contain "cat"
```

```
(define (contains-dog? l)
  (contains? "dog" l))
```

```
(define (contains-cat? l)
  (contains? "cat" l))
```

Figure 66: Two similar functions, revisited

What you have just seen is called *functional abstraction*. Abstracting different versions of functions is one way to eliminate similarities from programs, and as you can see with this one simple example, doing so simplifies programs.

Exercise 237. Use `contains?` to define functions that search for `"atom"`, `"basic"`, and `"zoo"`, respectively. ■

Computing borrows the term “abstract” from mathematics. A mathematician refers to “6” as an abstract number because it only represents all different ways of naming six things. In contrast, “6 inches” or “6 eggs” are concrete instances of “6” because they express a measurement and a count.

Exercise 238. Create test suites for the following two functions:

<pre>; Lon -> Lon ; add 1 to each number on l (define (add1* l) (cond [(empty? l) '()] [else (cons (add1 (first l)) (add1* (rest l))))]))</pre>	<pre>; Lon -> Lon ; adds 5 to each number on l (define (plus5 l) (cond [(empty? l) '()] [else (cons (+ (first l) 5) (plus5 (rest l))))]))</pre>
--	--

■

Then abstract over them. Define the above two functions in terms of the abstraction as one-liners and use the existing test suites to confirm that the revised definitions work properly. Finally, design a function that subtracts 2 from each number on a given list.}

16.2 Different Similarities

Abstraction looks easy in the case of `contains-dog?` and `contains-cat?`. It takes only a comparison of two function definitions, a replacement of a literal string with a function parameter, and a quick check that it is easy to define the old functions with the abstract function. This kind of abstraction is so natural that it showed up in the preceding two parts of the book without much ado.

This section illustrates how the very same principle yields a powerful form of abstraction. Take a look at [figure 67](#). Both functions consume a list of numbers and a threshold. The left one produces a list of all those numbers that are below the threshold, while the one on the right produces all those that are above the threshold.

<pre>; Lon Number -> Lon ; select those numbers on l ; that are below t (define (small l t) (cond [(empty? l) '()] [else (cond [(< (first l) t) (cons (first l) (small (rest l) t))]</pre>	<pre>; Lon Number -> Lon ; select those numbers on l ; that are above t (define (large l t) (cond [(empty? l) '()] [else (cond [(> (first l) t) (cons (first l) (large (rest l) t))]</pre>
--	--

```
(cons (first l)
      (small (rest l) t)))
[else
  (small (rest l) t))]))
```

```
(cons (first l)
      (large (rest l) t)))
[else
  (large (rest l) t))]))
```

Figure 67: Two more similar functions

The two functions differ in only one place: the comparison operator that determines whether a number from the given list should be a part of the result or not. The function on the left uses `<`, the right one `>`. Other than that, the two functions look identical, not counting the function name.

Let us follow the first example and abstract over the two functions with an additional parameter. This time the additional parameter represents a comparison operator rather than a string:

```
(define (extract R l t)
  (cond
    [(empty? l) '()]
    [else (cond
              [(R (first l) t)
               (cons (first l) (extract R (rest l) t))]
              [else
               (extract R (rest l) t))]))]
```

To apply this new function, we must supply three arguments: a function `R` that compares two numbers; a list `l` of numbers, and a threshold `t`. The function then extracts all those items `i` from `l` for which `(R i t)` evaluates to `#true`.

Stop! At this point you should ask whether this definition makes any sense. Without further ado, we have created a function that consumes a function—something that you probably have not seen before. It turns out, however, that your simple little teaching language ISL supports these kinds of functions, and that defining such functions is one of the most powerful tools of good programmers—even in languages in which function-consuming functions do not seem to be available.

If you have taken a calculus course, you encountered the differential operator and the indefinite integral, both of which are functions that consume and produce a function. But we do not assume that you have taken such a course and show you these wonderful tricks anyway.

Testing shows that `(extract < l t)` computes the same result as `(small l t)`:

```
(check-expect (extract < '() 5) (small '() 5))
(check-expect (extract < '(3) 5) (small '(3) 5))
(check-expect (extract < '(1 6 4) 5) (small '(1 6 4) 5))
```

Similarly, `(extract > l t)` produces the same output as `(large l t)`, which means that you can define the two original functions like this:

```
; Lon Number -> Lon
(define (small-1 l t)
  (extract < l t))
; Lon Number -> Lon
(define (large-1 l t)
  (extract > l t))
```

The important insight is **not** that `small-1` and `large-1` are one-line definitions. Once you have an abstract function such as `extract`, you can put it to good uses elsewhere:

1. (`extract = l t`): This expression extracts all those numbers in `l` that are equal to `t`.
2. (`extract <= l t`): This one produces the list of numbers in `l` that are less than or equal to `t`.
3. (`extract >= l t`): This last expression computes the list of numbers that are greater than or equal to the threshold.

As a matter of fact, the first argument for `extract` need not be one of ISL's predefined operations. Instead, you can use any function that consumes two arguments and produces a `Boolean`. Consider this example:

```
; Number Number -> Boolean
; is the area of a square with side x larger than c
(define (squared>? x c)
  (> (* x x) c))
```

That is, `squared>?` checks whether the claim $x^2 > c$ holds, and it is usable with `extract`:

```
(extract squared>? (list 3 4 5) 10)
```

This application extracts those numbers in `(list 3 4 5)` whose square is larger than `10`.

Exercise 239. Evaluate `(squared>? 3 10)`, `(squared>? 4 10)`, and `(squared>? 5 10)` in DrRacket. ■

So far you have seen that abstracted function definitions can be more useful than the functions you start from. For example, `contains?` is more useful than `contains-dog?` and `contains-cat?`, and `extract` is more useful than `small` and `large`. Another important aspect of abstraction is that you now have a single point of control over all these functions. If it turns out that the abstract function contains a mistake, fixing its definition suffices to fix all other definitions. Similarly, if you figure out how to accelerate the computations of the abstract function or how to reduce its energy consumption, then all functions defined in terms of this function are improved without further ado. The following exercises indicate how these single-point-of-control improvements work.

These effects of abstraction are crucial for large, industrial programming projects. For that reason, programming language and software engineering research has focused on how to create single points of control in large projects. Of course, the same idea applies to all kinds of computer designs (word documents, spread sheets) and organizations in general.

<pre>; Nelon -> Number ; determines the smallest ; number on l (define (inf l) (cond [(empty? (rest l)) (first l)] [else (cond [(< (first l) (inf (rest l))) (first l)] [else (inf (rest l))])]))</pre>	<pre>; Nelon -> Number ; determines the largest ; number on l (define (sup l) (cond [(empty? (rest l)) (first l)] [else (cond [(> (first l) (sup (rest l))) (first l)] [else (sup (rest l))])]))</pre>
---	--

Figure 68: Finding the `inf` and `sup` in a list of numbers

Exercise 240. Abstract the two functions in [figure 68](#) into a single function: Both consume non-empty lists of numbers (*Nelon*) and produce a single number. The left one produces the smallest number in the list, the right one the largest.

Define `inf-1` and `sup-1` in terms of the abstract function. Test them with these two lists:

```
(list 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
(list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25)
```

Why are these functions slow on some of the long lists?

Modify the original functions with the use of `max`, which picks the larger of two numbers, and `min`, which picks the smaller one. Then abstract again, define `inf-2` and `sup-2`, and test them with the same inputs again. Why are these versions so much faster?

For a complete answer to the two questions on performance, see [Local Function Definitions](#).

16.3 Similarities in Data Definitions

Now take a close look at the following two data definitions:

```
; List-of-numbers           ; List-of-String
; A Lon is one of:         ; A Los is one of:
; - '()                   ; - '()
; - (cons Number Lon)    ; - (cons String Los)
```

The one on the left introduces lists of numbers; the one on the right describes lists of strings. And the two data definitions are similar. Like similar functions, the two data definitions use two different names, but this is irrelevant because any name would do. The only real difference concerns the first position inside of `cons` in the second clause, which specifies what kind of items the list contains.

In order to abstract over this one difference, we proceed as if a data definition were a function. We introduce a parameter, which makes the data definition look like a function, and where there used to be different references, we use this parameter:

```
; A [List-of ITEM] is one of:
; - '()
; - (cons ITEM [List-of ITEM])
```

We call such abstract data definitions *parametric data definitions* because of the parameter. Roughly speaking, a parametric data definition abstracts from a reference to a particular collection of data in the same manner as a function abstracts from a particular value.

The question is, of course, what these parameters range over. For a function, they stand for an unknown value; when the function is applied, the value becomes known. For a parametric data definition, a parameter stands for an entire class of values. The process of supplying the name of a data collection to a parametric data definition is called *instantiation*; here are some sample instantiations of the `List-of` abstraction:

- `[List-of Number]` says that `ITEM` represents all numbers so the notation is just another name for `List-of-numbers`;

- [List-of String] defines the same class of data as List-of-String; and
- if we had identified a class of inventory records, like this:

```
| (define-struct ir [name price])
; An IR is
; (make-ir String Number)
```

then [List-of IR] would be a name for the class of lists of inventory records.

By convention, we use names with all capital letters for parameters of data definitions, while the arguments are (usually) the names of existing data collections.

Our way to validate that these shorthands really mean what we say they mean is to substitute the actual name of a data definition, e.g., Number, for the parameter ITEM of the data definition and to use a plain name for the data definition:

```
| ; A List-of-numbers-again is one of:
| ; - '()
| ; - (cons Number List-of-numbers-again)
```

Since the data definition is self-referential, we copied the entire data definition. The resulting definition looks exactly like the conventional one for lists of numbers and truly identifies the same class of data.

Let us take a brief look at a second example, starting with a structure type definition:

```
| (define-struct point [hori veri])
```

Here are two different data definitions that rely on this structure type definition:

```
| ; A Pair-boolean-string is a ; A Pair-number-image is a
| ; (make-point Boolean String) ; (make-point Number Image)
```

In this case, the data definitions differ in two places—both marked by highlighting. The differences in the hori fields correspond to each other, and so do the differences in the veri fields. It is thus necessary to introduce two parameters to create an abstract data definition:

```
| ; A [CP H V] is a structure:
| ; (make-point H V)
```

Here H is the parameter for data collections for the hori field, and V stands for data collections that can show up in the veri field.

To instantiate a data definition with two parameters, you need two names of data collections. Using Number and Image for the parameters of CP, you get [CP Number Image], which describes the collections of points that combine a number with an image. In contrast [CP Boolean String] combines Boolean values with strings in a point structure.

Exercise 241. A list of two items is another frequently used form of data in ISL programming. Here is data definition with two parameters:

```
| ; A [List X Y] is a structure:
| ; (cons X (cons Y '()))
```

Instantiate this definition to retrieve

- pairs of Numbers,

- pairs of **Numbers** and **1Strings**, and
- pairs of **Strings** and **Booleans**.

Also make one concrete example for each of these three data definitions. ▀

Once you have parametric data definitions, you can even mix and match them to great effect. Consider this one:

```
| ; [List-of [CP Boolean Image]]
```

The outermost notation is **[List-of ...]**, which means that you are dealing with a list.

Question is what kind of data the list contains, and to answer that question, you need to study the inside of the **List-of** expression:

```
| ; [CP Boolean Image]
```

The inner part combines **Boolean** and **Image** in a point, naming yet another class of data that pairs up two different classes in structure instances. In short,

```
| ; [List-of [CP Boolean Image]]
```

is a list of **CPs** that combine **Booleans** and **Images**. Similarly,

```
| ; [CP Number [List-of Image]]
```

is an instantiation of **CP** that combines one **Number** with a list of **Images**. If this went too fast, tease apart this data expression, like above, and explain each pieces as you go.

Exercise 242. Here are two strange but similar data definitions:

```
; A Nested-string is one of:  
; - String  
; - (make-layer Nested-string)      ; A Nested-number is one of:  
; - Number  
; - (make-layer Nested-number)
```

Both data definitions exploit this structure type definition:

```
| (define-struct layer [stuff])
```

Both define nested forms of data: one is about numbers and the other about strings. Make examples for both. Abstract over the two. Then instantiate the abstract definition to get back the originals. ▀

Exercise 243. Compare and contrast the data definitions for **NEList-of-temperatures** and **NEList-of-Booleans**. Then formulate an abstract data definition **NEList-of**. ▀

Exercise 244. Take a look at this data definition:

```
| ; A [Bucket ITEM] is  
|   (make-bucket N [List-of ITEM])  
|   interpretation the n in (make-bucket n l) is the length of l  
|   i.e., (= (length l) n) is always true
```

When you instantiate **Bucket** with **String**, **IR**, and **Posn**, you get three different data collections. Describe each of them with a sentence and with two distinct examples.

Now consider this instantiations:

```
| ; [Bucket [List-of [List-of String]]]
```

Construct three distinct pieces of data that belong to this collection. ▀

Exercise 245. Here is one more parametric data definition:

```
; A [Maybe X] is one of:  
; - #false  
; - X
```

Interpret the following data definitions:

- [Maybe String]
- [Maybe [List-of String]]
- [List-of [Maybe String]]

What does the following function signature mean:

```
; String [List-of String] -> [Maybe [List-of String]]  
; returns the remainder of the list los if it contains s  
; #false otherwise  
(check-expect (occurs "a" (list "b" "a" "d")) (list "d"))  
(check-expect (occurs "a" (list "b" "c" "d")) #f)  
(define (occurs s los)  
  los)
```

Design the function. ▀

16.4 Functions Are Values

The functions of this section stretch our understanding of program evaluation. It is easy to understand how functions consume more than numbers, say strings, images, and Boolean values. Structures and lists are a bit of a stretch, but they are finite “things” in the end. Function-consuming functions, however, are strange. Indeed, these kind of functions violate the BSL grammar of the first intermezzo in two ways. First, the names of primitive operations and functions are used as arguments in applications. Second, parameters are used as if they were functions, that is, the first position of applications.

Spelling out the problem tells you how the ISL grammar differs from BSL’s. First, our expression language should include the names of functions and primitive operations in the definition. Second, the first position in an application should allow things other than function names and primitive operations; at a minimum, it must allow variables and function parameters. In anticipation of other uses of functions, we agree on allowing arbitrary expressions in that position.

The changes to the grammar seem to demand changes to the evaluation rules, but they don’t change at all. All that changes is the set of values. To accommodate functions as arguments of functions, the simplest change is to say that **functions are values**. Thus, we start using the names of functions and primitive operations as values; later we introduce another way to deal with functions as values.

Exercise 246. Assume the definitions area in DrRacket contains `(define (f x) x)`.

Identify the values among the following expressions:

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

```
3. (cons f (cons 10 (cons (f 10) '())))
```

Explain why they are values and why the remaining expressions are not values. ▀

Exercise 247. Argue why the following sentences are now legal definitions:

1. (define (f x) (x 10))
2. (define (f x) (x f))
3. (define (f x y) (x 'a y 'b))

Explain your reasoning. ▀

Exercise 248. Develop function=? . The function determines whether two functions from numbers to numbers produce the same results for 1.2, 3, and -5.775.

Mathematicians say that two functions are equal if they compute the same result when given the same input—for all possible inputs.

Can we hope to define function=? , which determines whether two functions from numbers to numbers are equal f? If so, define the function. If not, explain why and consider the implication that you have encountered the first easily definable idea for which you cannot define a function. ▀

16.5 Computing with Functions

The switch from BSL+ to ISL allows the use of functions as values, that is, as arguments to functions, as results, as items in lists, and so on. In function applications, the first position is now just an expression and DrRacket evaluates this expression like all others. Naturally, it expects a function as a result. Surprisingly, a liberal interpretation of the laws of algebra suffices to evaluate programs in ISL.

Let us see how this works for the extract function from [Different Similarities](#). Since <, '(), and 5 are values, it's obvious that

```
(extract < '() 5) == '()
```

holds. We can use the law of substitution—also known as beta from [Intermezzo: BSL](#)—and continue computing with the body of the function. Like so many times, the parameters, R, l, and t, are replaced by their arguments, <, '(), and 5, respectively. From here, it is almost plain arithmetic, starting with the conditionals:

```
==  
  (cond  
    [(empty? '()) '()]  
    [else (cond  
      [(< (first '()) t) (cons (first '()) (extract < (rest '()) 5))]  
      [else (extract < (rest '()) 5)])])  
==  
  (cond  
    [#true '()]  
    [else (cond  
      [(< (first '()) t) (cons (first '()) (extract < (rest '()) 5))]  
      [else (extract < (rest '()) 5)])])
```

```

  ==
  '()

```

So next we look at a one-item list:

```
(extract < (cons 4 '()) 5)
```

The result should be `(cons 4 '())` because the only item of this list is `4` and `(< 4 5)` is true. Here is the first step of the evaluation:

```

(extract < (cons 4 '()) 5)
==
(cond
  [(empty? (cons 4 '()))] '())
  [else (cond
    [(< (first (cons 4 '())) 5)
     (cons (first (cons 4 '()))) (extract < (rest (cons 4 '()) 5)))]
    [else (extract < (rest (cons 4 '()) 5))])])

```

Again, all occurrences of R are replaced by `<`, l by `(cons 4 '())`, and t by `5`. The rest is straightforward:

```

(cond
  [(empty? (cons 4 '()))] '())
  [else (cond
    [(< (first (cons 4 '())) 5)
     (cons (first (cons 4 '()))) (extract < (rest (cons 4 '()) 5)))]
    [else (extract < (rest (cons 4 '()) 5))])])
==

(cond
  [#false '()]
  [else (cond
    [(< (first (cons 4 '())) 5)
     (cons (first (cons 4 '()))) (extract < (rest (cons 4 '()) 5)))]
    [else (extract < (rest (cons 4 '()) 5))])])
==

(cond
  [(< (first (cons 4 '())) 5)
   (cons (first (cons 4 '()))) (extract < (rest (cons 4 '()) 5)))]
  [else (extract < (rest (cons 4 '()) 5))])
==

(cond
  [(< 4 5)
   (cons (first (cons 4 '()))) (extract < (rest (cons 4 '()) 5)))]
  [else (extract < (rest (cons 4 '()) 5))])

```

This is the key step, with `<` used normally after being substituted into this position. And it continues with arithmetic:

```

==
(cond
  [#true
   (cons (first (cons 4 '()))) (extract < (rest (cons 4 '()) 5)))]
  [else (extract < (rest (cons 4 '()) 5))])
==

(cons 4 (extract < (rest (cons 4 '()) 5)))

```

```

==  

(cons 4 (extract < '() 5))  

==  

(cons 4 '())

```

The last step is the equation discussed above, meaning there is no need to spell out the reasoning again.

Our final example is an application of **extract** to a list of two items:

```

(extract < (cons 6 (cons 4 '()))) 5)  

== (extract < (cons 4 '()) 5)  

== (cons 4 (extract < '() 5))  

== (cons 4 '())

```

Step 1 is new and says that **extract** determines that the first item on the list is not less than the threshold and that it therefore is not added to the result of the natural recursion.

```

(extract squared? (list 3 4 5) 10) == (1)  

==  

(cond  

  [(empty? (list 3 4 5)) '()]  

  [else (cond  

    [(squared?? (first (list 3 4 5)) 10) (2)  

     (cons (first (list 3 4 5))  

      (extract squared?? (rest (list 3 4 5)) 10)))]  

    [else (extract squared?? (rest (list 3 4 5)) 10))])])  

...  

==  

(cond  

  [(squared?? 3 10) (3)  

   (cons (first (list 3 4 5))  

    (extract squared?? (rest (list 3 4 5)) 10)))]  

  [else (extract squared?? (rest (list 3 4 5)) 10))])  

==  

(cond  

  [(> (* 3 3) 10) (4)  

   (cons (first (list 3 4 5))  

    (extract squared?? (rest (list 3 4 5)) 10)))]  

  [else (extract squared?? (rest (list 3 4 5)) 10))])  

==  

(cond  

  [(> 9 10) (5)  

   (cons (first (list 3 4 5))  

    (extract squared?? (rest (list 3 4 5)) 10)))]  

  [else (extract squared?? (rest (list 3 4 5)) 10))])  

==  

(cond  

  [#false (6)  

   (cons (first (list 3 4 5))  

    (extract squared?? (rest (list 3 4 5)) 10)))]  

  [else (extract squared?? (rest (list 3 4 5)) 10))])  

...  

== (7)  

(extract squared?? (rest (list 3 4 5)) 10)

```

Figure 69: Computing with functions

Exercise 249. Use DrRacket's stepper to fill in the gaps between the steps in this last calculation. Pay attention to how the stepper deals with functions as arguments and how it retains 4. ▀

Exercise 250. Check step 1 of the last calculation

```
(extract < (cons 6 (cons 4 '()) 5)
==

(extract < (cons 4 '()) 5)
```

using DrRacket's stepper. ▀

Exercise 251. Evaluate the expression

```
(extract < (cons 8 (cons 6 (cons 4 '()))) 5)
```

with DrRacket's stepper. ▀

Exercise 252. Evaluate `(squared? 3 10)`, `(squared? 4 10)`, and `(squared? 5 10)` in DrRacket's stepper. Figure 69 shows how DrRacket performs the following evaluation:

```
> (extract squared? (list 3 4 5) 10)
'(4 5)
```

Use the stepper to fill in the gap between lines (2) and (3). Also explain the step from line (3) to line (4) and complete the evaluation. ▀

Exercise 253. Functions are values: arguments, results, items in lists. Place the following definitions and expressions into DrRacket's definitions window and use the stepper to find out how running this program works:

```
(define (f x) x)
(cons f '())
(f f)
(cons f (cons 10 (cons (f 10) '()))))
```

Note The stepper displays functions as so-called `lambda` expressions. [Nameless Functions](#) explains this idea in detail. ▀

17 Designing Abstractions

In essence, to abstract is to turn something concrete into a parameter. We have this several times in the preceding section. To abstract similar function definitions, you add parameters that replace concrete values in the definition. To abstract similar data definitions, you create parametric data definitions. When you will encounter other programming languages, you will see that their abstraction mechanisms also require the introduction of parameters, though they may not be function parameters.

17.1 Abstractions from Examples

When you first learned to add, you worked with concrete examples. Your parents probably taught you to use your fingers to add two small numbers. Later on, you studied how to add two arbitrary numbers; you were introduced to your first kind of abstraction. Much later

still, you learned to formulate expressions that convert temperatures from Celsius to Fahrenheit or calculate the distance that a car travels at a certain speed in a given amount of time. In short, you went from very concrete examples to abstract relations.

```

; List-of-numbers -> List-of-numbers ; Inventory -> List-of-strings
; converts a list of Celsius ; extracts the names of toys
; temperatures to Fahrenheit ; from an inventory
(define (cf* l)
  (cond
    [(empty? l) '()]
    [else
      (cons (C2F (first l))
            (cf* (rest l))))])))

; Number -> Number
; converts one Celsius
; temperature to Fahrenheit
(define (C2F c)
  (+ (* 9/5 c) 32))

; Inventory -> List-of-strings
; extracts the names of toys
; from an inventory
(define (names i)
  (cond
    [(empty? i) '()]
    [else
      (cons (IR-name (first i))
            (names (rest i))))]))

(define-struct IR [name price])
; An IR is (make-IR String Number)
; An Inventory is one of:
; - '()
; - (cons IR Inventory)

```

Figure 70: A pair of similar functions

This section introduces a design recipe for creating abstractions from examples. As the preceding section shows, creating abstractions is easy. We leave the difficult part to the next section where we show you how to find and use existing abstractions.

Recall the essence of [Similarities Everywhere](#). We start from two concrete function definitions or two concrete data definitions; we compare them; we mark the differences; and then we abstract. And that is mostly all there is to creating abstractions:

1. Step 1 is to **compare** two items for similarities.

When you find two function definitions that are almost the same except for their names and some *values* at a few *analogous* places, compare them, mark the differences. If the two definitions differ in more than one place, connect the corresponding differences with a line or a comment.

The recipe requires a substantial modification for abstracting over non-values.

[Figure 70](#) shows a pair of similar function definitions:. The two functions apply a function to each item in a list. They differ only as to which function they apply to each item. The two highlights emphasize this essential difference. They also differ in two inessential ways: the names of the function and the names of the parameters.

2. Next we abstract. **To abstract** means to replace the contents of corresponding code highlights with new names and add these names to the parameter list. For our running example, we obtain the following pair of functions after replacing the differences with g:

```

(define (cf* l g)
  (cond
    [(empty? l) '()]
    [else
      (cons (g (first l))
            (cf* (rest l) g))])))

(define (names i g)
  (cond
    [(empty? i) '()]
    [else
      (cons (g (first i))
            (names (rest i) g))]))

```

This first change eliminates the essential difference. Now each function traverses a list and applies some given function to each item.

The inessential differences—the names of the functions and occasionally the names of some parameters—are easy to eliminate. Indeed, if you have explored DrRacket, you know that check syntax allows you to do this systematically and easily:

```
(define (map1 k g)
  (cond
    [(empty? k) '()]
    [else
      (cons (g (first k))
            (map1 (rest k) g))]))
```

```
(define (map1 k g)
  (cond
    [(empty? k) '()]
    [else
      (cons (g (first k))
            (map1 (rest k) g))))])
```

We choose to use `map1` for the name of the function and `k` for the name of the list parameter. No matter which names you choose, the result is two identical function definitions.

Our example is simple. In many cases, you will find that there is more than just one pair of differences. The key is to find pairs of differences. When you mark up the differences on paper and pencil, connect related boxes with a line. Then introduce one additional parameter per line. And don't forget to change all recursive uses of the function so that the additional parameters go along for the ride.

3. Now we must validate that the new function is a correct abstraction of the original pair of functions. To validate means **to test**, which here means to define the two original functions in terms of the abstraction.

Thus suppose that one original function is called `f-original` and consumes one argument and that the abstract function is called `abstract`. If `f-original` differs from the other concrete function in the use of one value, say, `val`, the following function definition

```
(define (f-from-abstract x)
  (abstract x val))
```

introduces the function `f-from-abstract`, which should be equivalent to `f-original`. That is, for every proper value `V`, `(f-from-abstract V)` should produce the same answer as `(f-original V)`. This is particularly true for all values that your tests for `f-original` use. So re-formulate and re-run those tests for `f-from-abstract` and make sure they succeed.

Let us return to our running example:

```
; List-of-numbers -> List-of-numbers ; Inventory -> List-of-strings
(define (cf*-from-map1 l) (define (names-from-map1 i)
  (map1 l C2F)) (map1 i IR-name))
```

A complete example would include some tests, and thus we can assume that both `cf*` and `names` come with some tests:

```
(check-expect
  (cf*
   (list 100 0 -40))
  (list 212 32 -40))
```

```
(check-expect
  (names
   (list
    (make-IR "doll" 21.0)
    (make-IR "bear" 13.0)))
  (list "doll" "bear"))
```

To ensure that the functions defined in terms of `map1` work properly, you can copy the

tests and change the function names appropriately:

```
(check-expect
  (cf*-from-map1
    (list 100 0 -40))
  (list 212 32 -40))

(check-expect
  (names-from-map1
    (list
      (make-IR "doll" 21.0)
      (make-IR "bear" 13.0)))
  (list "doll" "bear")))
```

4. To make a new abstraction useful, it needs a **signature**. As [Using Abstractions](#) explains, reuse of abstract functions start with their signatures. Finding useful signatures is, however, a serious problem. For now we just use the running example to illustrate the problem. [Similarities in Signatures](#) below resolves the issue.

So consider the problem of finding a signature for `map1`. On the one hand, if you view `map1` as an abstraction of `cf*`, you might think the signature is

```
; List-of-numbers [Number -> Number] -> List-of-numbers
```

That is, the original signature extended with one signature for functions:

```
; [Number -> Number]
```

Since the additional parameter for `map1` is a function, the use of a function signature shouldn't surprise you. This function signature is also quite simple; it is a "name" for all the functions from numbers to numbers. Here `C2F` is such a function, and so are `add1`, `sin`, and `imag-part`.

On the other hand, if you view `map1` as an abstraction of `names`, the signature is quite different:

```
; Inventory [IR -> String] -> List-of-strings
```

This time the additional parameter is `IR-name`, which is a selector function that consumes `IRs` and produces `Strings`. But clearly this second signature would be useless in the first case, and vice versa. To accommodate both cases, the signature for `map1` must express that `Number`, `IR`, and `String` are coincidental.

Also concerning signatures, you are probably eager to use `List-of` by now. It is clearly easier to write `[List-of IR]` than spelling out a data definition for `Inventory`. So yes, as of now, we use `List-of` when it is all about lists and you should too.

Once you have abstracted two functions, you should check whether there are other uses for the abstract function. If so, the abstraction is truly useful. Consider `map1` for example. It is easy to see how to use it to add `1` to each number on a list of numbers:

```
; List-of-numbers -> List-of-numbers
(define (add1-to-each l)
  (map1 l add1))
```

Similarly, `map1` can also be used to extract the price of each item in an inventory. When you can imagine many such uses for a new abstraction, add it to a library of useful functions to have around. Of course, it is quite likely that someone else has thought of it and the function is already a part of the language. For a function like `map1`, see [Using Abstractions](#).

Exercise 254. Design `tabulate`, which is the abstraction of the following two functions:

```

; Number -> [List-of Number]
; tabulates sin between n
; and 0 (inclusive) in a list
(define (tab-sin n)
  (cond
    [(= n 0) (list (sin 0))]
    [else
      (cons (sin n)
        (tab-sin (sub1 n))))]))
```

```

; Number -> [List-of Number]
; tabulates sqrt between n
; and 0 (inclusive) in a list
(define (tab-sqrt n)
  (cond
    [(= n 0) (list (sqrt 0))]
    [else
      (cons (sqrt n)
        (tab-sqrt (sub1 n))))]))
```

When tabulate is properly designed, use it to define a tabulation function for `sqr` and `tan`. ■

Exercise 255. Design `fold1`, which is the abstraction of the following two functions:

```

; [List-of Number] -> Number
; computes the sum of
; the numbers on l
(define (sum l)
  (cond
    [(empty? l) 0]
    [else
      (+ (first l)
        (sum (rest l))))]))
```

```

; [List-of Number] -> Number
; computes the product of
; the numbers on l
(define (product l)
  (cond
    [(empty? l) 1]
    [else
      (* (first l)
        (product (rest l))))]))
```

Exercise 256. Design `fold2`, which is the abstraction of the following two functions:

```

; [List-of Number] -> Number
(define (product l)
  (cond
    [(empty? l) 1]
    [else
      (* (first l)
        (product (rest l))))]))
```

```

; [List-of Posn] -> Image
(define (image* l)
  (cond
    [(empty? l) emt]
    [else
      (place-dot (first l)
        (image* (rest l))))]))
```



```

; Posn Image -> Image
(define (place-dot p img)
  (place-image dot
    (posn-x p) (posn-y p)
    img))
```



```

; graphical constants:
(define emt (empty-scene 100 100))
(define dot (circle 3 "solid" "red"))
```

Compare this exercise with [exercise 255](#). Even though both involve the `product` function, this exercise poses an additional challenge because the second function, `image*`, consumes a list of `Posns` and produces an `Image`. Still, the solution is within reach of the material in this section, and it is especially worth comparing the solution with the one to the preceding exercise. The comparison yields interesting insights into abstract signatures. ■

Last but not least, when you are dealing with data definitions, the abstraction process proceed in an analogous manner. The extra parameters to data definitions stands for collections of values, and testing means spelling out a data definition for some concrete examples. All in all, abstracting over data definitions tends to be easier than abstracting over functions, and so we leave it to you to adapt the design recipe appropriately.

17.2 Similarities in Signatures

As it turns out, a function's signature is key to its reuse. Thus, to increase the usefulness of an abstract function, you must learn to formulate signatures that describes abstracts in their most general terms possible. To understand how this works, we start with a second look at signatures and from the simple—though possibly startling—insight that signatures are basically data definitions.

Both signatures and data definitions specify a class of data; the difference is that data definitions also name the class of data while signatures don't. Nevertheless, when you write down

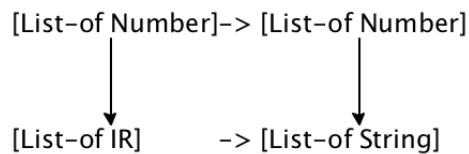
```
; Number Boolean -> String
(define (f n b) "hello world")
```

your first line describes an entire class of data, and your second line states that `f` belongs to that class. To be precise, this signature describes the class of **all functions** that consume a `Number` and a `Boolean` and that produce a `String`.

In general, the arrow notation of signatures is like the `List-of` notation from [Similarities in Data Definitions](#). The latter consumes (the name of) one class of data, say `X`, and describes all lists of `X` items—without assigning it a name. The arrow notation consumes an arbitrary number of classes of data and describes collections of functions.

What this means is that the abstraction design recipe applies to signatures, too. You compare similar signatures; you highlight the differences; and then you replace those with parameters. But the process of abstracting signatures feels more complicated than the one for functions, partly because signature are already abstract pieces of the design recipe and partly because the arrow-based notation is much more complex than anything else we have encountered.

Let us start with the signatures of `c f*` and names:



The diagram is the result of the compare-and-contrast step. Comparing the two signatures shows that they differ in two places: to the left of the arrow, we see `Number` versus `IR` and to its right, it is `Number` versus `String`.

If we replace the two differences with some kind of parameters, say `X` and `Y`, we get the same signature:

```
; [X Y] [List-of X] -> [List-of Y]
```

The new signature starts with a sequence of variables, drawing an analogy to function definitions and the data definitions above. Roughly speaking, these variables are the parameters of the signature, like those of functions and data definitions. To make the latter concrete, the variable sequence is like `ITEM` in the definition of `List-of` or the `X` and `Y` in the definition of `CP` from [Similarities in Data Definitions](#). And just like those, `X` and `Y` range over classes of values.

An instantiation of this parameter list is the rest of the signature with the parameters replaced by the data collections: either their names or other parameters or abbreviations such as `List-of` from above. Thus, if you replace both `X` and `Y` with `Number`, you get back

the signature for `cf*`:

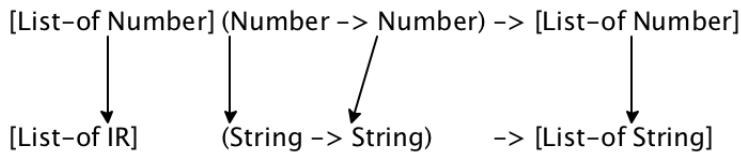
`; [List-of Number] -> [List-of Number]`

If you choose `IR` and `String`, respectively, you get back the signature for `names`:

`; [List-of IR] -> [List-of String]`

And that explains why we may consider this parametrized signature as an abstraction of the original signatures for `cf*` and `names`.

Once we add the extra function parameter to these two functions we get `map1` and the signatures are as follows:



Again, the signatures are in pictorial form and with arrows connecting the corresponding differences. These mark-ups suggest that the differences in the second argument—a function—are related to the differences in the original signatures. Specifically, `Number` and `IR` on the left of the new arrow refer to items on the first argument—a list—and the `Number` and `String` on the right refer to the items on the result—also a list.

Since listing the parameters of a signature is extra work for our purposes, we simply say that from now on all variables in signatures are parameters. Other programming languages, however, insist on explicitly listing the parameters of signatures, but in return you can articulate additional constraints in such signatures and the signatures are checked before you run the program.

Now let's apply the same trick to get a signature for `map1`:

`; [List-of X] [X -> Y] -> [List-of Y]`

Concretely, `map1` consumes a list of items, all of which belong to some (yet to be determined) collection of data called `X`. It also consumes a function that consumes elements of `X` and produces elements of a second unknown collection, called `Y`. The result of `map1` are lists that contain items from `Y`.

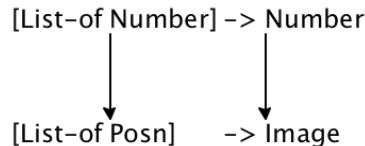
As you may guess from our first example, abstracting over signatures requires practice. So here is a second pair of signatures:

`; [List-of Number] -> Number ; [List-of Posn] -> Image`

They are the signatures for `product` and `image*` in [exercise 256](#). While the two signatures have some common organization, the differences are distinct. Let us first spell out the common organization in detail:

- both signatures describe one-argument functions;
- both argument descriptions employ the `List-of` construction;

The difference is that the first signature refers to `Number` twice, and the second one refers to `Posns` and `Images`. Putting similarities and differences together, the first occurrence of `Number` corresponds to `Posn` and the second one to `Image`:



To make more progress on a signature for the abstraction of the two functions in [exercise 256](#), we need to take the first two steps of the design recipe:

```

(define (pr* l bs jn)
  (cond
    [(empty? l) bs]
    [else
      (jn (first l)
           (pr* (rest l) bs jn))]))
  
```



```

(define (im* l bs jn)
  (cond
    [(empty? l) bs]
    [else
      (jn (first l)
           (im* (rest l) bs jn))]))
  
```

Since the two functions differ in two pairs of values, the revised versions consume two additional values: one is an atomic value, to be used in the base case, and the other one is a function that joins together the result of the natural recursion with the first item on the given list.

The key is to translate this insight into two signatures for the two new functions. When you do so for `pr*`, you get

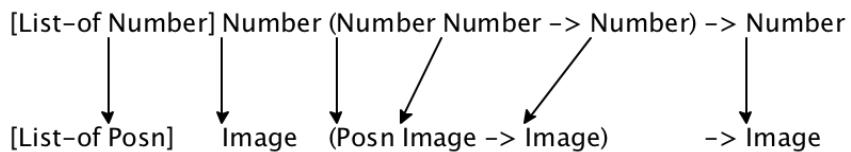
| ; [List-of Number] Number [Number Number -> Number] -> Number

because the result in the base case is a number and the function that combines the first item and the natural recursion is `+` in the original function. Similarly, for `im*` the signature is

| ; [List-of Posn] Image [Posn Image -> Image] -> Image

As you can see from the function definition for `im*`, the base case returns an image and the combination function is `place-dot`, which combines a `Posn` and an `Image` into an `Image`.

Now we take the diagram from above and extend it to the signatures with the additional inputs:



From this diagram, you can easily see that the two revised signatures share even more organization than the original two. Furthermore, the pieces that describe the base cases correspond to each other and so do the pieces of the sub-signature that describe the combination function. All in all there are six pairs of differences but they boil down to just two:

1. some occurrences of `Number` correspond to `Posn`
2. and other occurrences of `Number` correspond to `Image`.

So to abstract we need two variables, one per kind of correspondence.

Here, then, is the signature for `fold2`, the abstraction that [exercise 256](#) requests:

| ; [List-of X] Y [X Y -> Y] -> Y

Stop! Make sure that replacing both parameters of the signature, `X` and `Y`, with `Number`

yields the signature for `pr*` and that replacing the same variables with `Posn` and `Image`, respectively, yields the signature for `im*`.

The two examples illustrate how to find general signatures. In principle the process is just like the one for abstracting functions. Due to the informal nature of signatures, however, it cannot be checked with running examples the way code is checked. Here is step-by-step formulation:

1. Given two similar function definitions, `f` and `g`, compare their signatures for similarities and differences. The goal is to discover the organization of the signature and to mark the places where one signature differs from the other. Connect the differences as pairs just like you do when you analyze function bodies.
2. Abstract `f` and `g` into `f-abs` and `g-abs`. That is, add parameters that eliminate the differences between `f` and `g`. Create signatures for `f-abs` and `g-abs`. Keep in mind what the new parameters originally stood for; this helps you figure out the new pieces of the signatures.
3. Check whether the analysis of step 1 extends to the signatures of `f-abs` and `g-abs`. If so, replace the differences with variables that range over classes of data. Once the two signatures are the same you have a signature for the abstracted function.
4. Test the abstract signature in two ways. First, ensure that suitable substitutions of the variables in the abstract signature yield the signatures of `f-abs` and `g-abs`. Second, check that the generalized signature is in sync with the code. For example, if `p` is a new parameter and its signature is

```
| ; ... [A B -> C] ....
```

then `p` should always be applied to two arguments, the first one from `A` and the second one from `B`. And the result of an application of `p` is going to be a `C` and should be used where elements of `C` are expected.

As with abstracting functions, the key is to compare the concrete signatures of the examples and to determine the similarities and differences. With enough practice and intuition, you will soon be able to abstract signatures without much guidance.

Exercise 257. Each of the following signatures describes a collection of functions:

```
| ; [Number -> Boolean]
| ; [Boolean String -> Boolean]
| ; [Number Number Number -> Number]
| ; [Number -> [List-of Number]]
| ; [[List-of Number] -> Boolean]
```

Describe these collections with at least one example from ISL. ■

Exercise 258. Formulate signatures for the following functions:

- `sort-n`, which consumes a list of numbers and a function that consumes two numbers (from the list) and produces a `Boolean`; `sort-n` produces a sorted list of numbers.
- `sort-s`, which consumes a list of strings and a function that consumes two strings (from the list) and produces a `Boolean`; `sort-s` produces a sorted list of strings.

Then abstract over the two signatures, following the above steps. Also show that the generalized signature can be instantiated to describe the signature of a sort function for

lists of [IRs](#). ▀

Exercise 259. Formulate signatures for the following functions:

- `map-n`, which consumes a list of numbers and a function from numbers to numbers to produce a list of numbers.
- `map-s`, which consumes a list of strings and a function from strings to strings and produces a list of strings.

Then abstract over the two signatures, following the above steps. Also show that the generalized signature can be instantiated to describe the signature of the `map-IR` function above. ▀

17.3 Single Point of Control

In general, programs are like drafts of papers. Editing drafts is important to correct typos, to fix grammatical mistakes, to make the document consistent, and to eliminate repetitions. Nobody wants to read papers that repeat themselves a lot, and nobody wants to read such programs either.

The elimination of similarities in favor of abstractions has many advantages. Creating an abstraction simplifies definitions. It may also uncover problems with existing functions, especially when similarities aren't quite right. But, the single most important advantage is the creation of *single points of control* for some common functionality.

Putting the definition for some functionality in one place makes it easy to maintain a program. When you discover a mistake, you have to go to just one place to fix it. When you discover that the code should deal with another form of data, you can add the code to just one place. When you figure out an improvement, one change improves all uses of the functionality. If you had made copies of the functions or code in general, you would have to find all copies and fix them; otherwise the mistake might live on or the only one of the functions would run faster.

We therefore formulate this guideline:

Creating Abstractions: Form an abstraction instead of copying and modifying any piece of a program.

Our design recipe for abstracting functions is the most basic tool to create abstractions. To use it requires practice. As you practice, you expand your capabilities to read, organize, and maintain programs. The best programmers are those who actively edit their programs to build new abstractions so that they collect things related to a task at a single point. Here we use functional abstraction to study this practice; in other courses on programming, you will encounter other forms of abstraction, most importantly *inheritance* in class-based object-oriented languages.

17.4 Abstractions from Templates

Over the course of the first two chapters, we have designed many functions using the same template. After all, the design recipe says to organize functions around the organization of the (major) input data definition. It is therefore not surprising that many function definitions look similar to each other.

Indeed, you should abstract from the templates directly, you should do so automatically, and some experimental programming languages do so. Even though this topic is still a subject of research, you are now in a position to understand the basic idea. Consider the template for lists:

```
(define (fun-for-l l)
  (cond
    [(empty? l) ...]
    [else (... (first l) ... (fun-for-l (rest l)) ...))])
```

It contains two gaps, one in each clause. When you use this template to define a list-processing function, you usually fill these gaps with a value in the first `cond` clause and with a function `combine` in the second clause. The `combine` function consumes the first item of the list and the result of the natural recursion and creates the result from these two pieces of data.

Now that you know how to create abstractions, you can complete the definition of the abstraction from this informal description:

```
; [List-of X] Y [X Y -> Y] -> Y
(define (reduce l base combine)
  (cond
    [(empty? l) base]
    [else (combine (first l)
                  (reduce (rest l) base combine))]))
```

It consumes two extra arguments: `base`, which is the value for the base case, and `combine`, which is the function that performs the value combination for the second clause.

Using `reduce` you can define many plain list-processing functions as “one liners.” Here are definitions for `sum` and `product`, two functions used several times in the first few sections of this chapter:

$; \text{[List-of Number]} \rightarrow \text{Number}$ <code>(define (sum lon)</code> <code> (reduce lon 0 +))</code>	$; \text{[List-of Number]} \rightarrow \text{Number}$ <code>(define (product lon)</code> <code> (reduce lon 1 *))</code>
---	---

For `sum`, the base case always produces `0`; adding the first item and the result of the natural recursion combines the values of the second clause. Analogous reasoning explains `product`. Other list-processing functions can be defined in a similar manner using `reduce`.

18 Using Abstractions

Once you have abstractions, you should use them when possible. They create single points of control, and they are a work-saving device. More precisely, the use of an abstraction helps **readers** of your code to understand your intentions. If the abstraction is well-known and built into the language or comes with its standard libraries, it signals more clearly what your function does than custom-designed code.

This chapter is all about the reuse of existing ISL abstractions. It starts with a section on existing ISL abstractions, some of which you have seen under false names. The remaining sections are about re-using such abstractions. One key ingredient is a new syntactic construct, `local`, for defining functions and variables (and even structure types) locally within a function. An auxiliary ingredient, introduced in the last section, is the `lambda` construct for creating nameless functions; `lambda` is a convenience but inessential to the

idea of re-using abstract functions.

```

; N [N -> X] -> [List-of X]
; constructs a list by applying f to 0, 1, ..., (sub1 n)
;   (build-list n f) == (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

; [X -> Boolean] [List-of X] -> [List-of X]
; produces a list from all those items on alox for which p holds
(define (filter p alox) ...)

; [List-of X] [X X -> Boolean] -> [List-of X]
; produces a version of alox that is sorted according to cmp
(define (sort alox cmp) ...)

; [X -> Y] [List-of X] -> [List-of Y]
; constructs a list by applying f to each item on alox
;   (map f (list x-1 ... x-n)) == (list (f x-1) ... (f x-n))
(define (map f alox) ...)

; [X -> Boolean] [List-of X] -> Boolean
; determines whether p holds for every item on alox
;   (andmap p (list x-1 ... x-n)) == (and (p x-1) ... (p x-n))
(define (andmap p alox) ...)

; [X -> Boolean] [List-of X] -> Boolean
; determines whether p holds for at least one item on alox
;   (ormap p (list x-1 ... x-n)) == (or (p x-1) ... (p x-n))
(define (ormap p alox) ...)

; [X Y -> Y] Y [List-of X] -> Y
; applies f from right to left to each item in alox and base
;   (foldr f base (list x-1 ... x-n)) == (f x-1 ... (f x-n base))
(define (foldr f base alox) ...)

; [X Y -> Y] Y [List-of X] -> Y
; applies f from left to right to each item in alox and base
;   (foldl f base (list x-1 ... x-n)) == (f x-n ... (f x-1 base))
(define (foldl f base alox) ...)

```

Figure 71: ISL's abstract functions for list-processing

18.1 Existing Abstractions

ISL provides a number of abstract functions for processing natural numbers and lists.

[Figure 71](#) collects the header material for the most important ones. The first one processes natural numbers and builds lists:

```

> (build-list 3 add1)
'(1 2 3)

```

The next three process lists and produce lists:

```

> (filter odd? (list 1 2 3 4 5))
'(1 3 5)

```

```
> (sort (list 3 2 1 4 5) >)
'(5 4 3 2 1)
> (map add1 (list 1 2 2 3 3 3))
'(2 3 3 4 4 4)
```

In contrast, `andmap` and `ormap` process lists and produce a Boolean:

```
> (andmap odd? (list 1 2 3 4 5))
#false
> (ormap odd? (list 1 2 3 4 5))
#true
```

This kind of computation is called a *reduction* because a list of values is reduced to a single value.

Of all the functions in figure 71, `foldr` and `foldl` are the most powerful ones. Both reduce lists to values. The following two computations explain how to use the abstract examples in the headers of `foldr` and `foldl` to explain an application to `+`, `0`, and `(list 1 2 3 4 5)`:

<pre>(foldr + 0 '(1 2 3 4 5)) == (+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0)))))</pre>	<pre>(foldl + 0 '(1 2 3 4 5)) == (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))</pre>
<pre>== (+ 1 (+ 2 (+ 3 5)))</pre>	<pre>== (+ 5 (+ 4 (+ 3 2)))</pre>
<pre>== (+ 1 (+ 2 9))</pre>	<pre>== (+ 5 (+ 4 3))</pre>
<pre>== (+ 1 (+ 2 12))</pre>	<pre>== (+ 5 (+ 4 6))</pre>
<pre>== (+ 1 14)</pre>	<pre>== (+ 5 10)</pre>
<pre>== 15</pre>	<pre>== 15</pre>

As you can see from these calculations, `foldr` processes the list values from right to left and `foldl` from left to right. While for functions such as `+` the direction seems to make no difference, this isn't true in general as you can see soon.

Exercise 260. Explain the following two functions:

In mathematics such functions are called *associative*. And indeed, `+` is associative on integers and rational numbers in ISL. For inexact numbers, however, this is not true. See below.

```
; [X -> Number] [List-of X] -> X
; finds the (first) item in alox that maximizes f, that is:
; if (argmax f (list x-1 ... x-n)) == x-i,
; then (>= (f x-i) (f x-1)), (>= (f x-i) (f x-2)), and so on
(define (argmax f alox) ...)

; [X -> Number] [List-of X] -> X
; finds the (first) item in alox that minimizes f, that is:
; if (argmin f (list x-1 ... x-n)) == x-i,
; then (<= (f x-i) (f x-1)), (<= (f x-i) (f x-2)), and so on
(define (argmin f alox) ...)
```

```
(define-struct address [first-name last-name street])
; Addr is (make-address String String String)
; interpretation associates a street address with a person's name

; [List-of Addr] -> String
; creates a string of first names, sorted in alphabetical order,
```

```

; separated and surrounded by blank spaces
(define (listing l)
  (foldr string-append-with-space
    " "
    (sort (map address-first-name l)
      string<?)))

; String String -> String
; concatenates two strings and prefixes with space
(define (string-append-with-space s t)
  (string-append " " s t))

(define ex0
  (list (make-address "Matthias" "Fellson" "Sunburst")
    (make-address "Robert" "Findler" "South")
    (make-address "Matthew" "Flatt" "Canyon")
    (make-address "Shriram" "Krishna" "Yellow")))

(check-expect (listing ex0) " Matthew Matthias Robert Shriram ")

```

Figure 72: Creating a list of names with abstractions

Figure 72 illustrates the power of composing the functions from figure 71. Its main function is `listing`. The purpose is to create a string from a list of addresses. More precisely, the expected result is a string that represents a sorted list of first names, separated and surrounded by blank spaces.

A moment's reflection suggests a straightforward design plan for this problem:

1. design a function that extracts the first names from the given list of Addr;
2. design a function that sorts these names in alphabetical order;
3. design a function that concatenates the strings from step 2.

Before you read on, you may wish to execute this plan. That is, design all three functions and then compose them in the sense of [Composing Functions](#) to obtain your own version of `listing`.

In the new world of abstractions, it becomes unnecessary to design three separate functions. Take a close look at the innermost expression of `listing` in figure 72:

... (`map` address-first-name l)

By the purpose statement of `map`, it applies `address-first-name` to every single instance of `address` producing a list of first names as strings. Here is the immediately surrounding expression:

... (`sort` .. `string<?`)

The dots represent the result of the `map` expression. Since the latter supplies a list of strings, the `sort` expression produces a sorted list of first names. And that leaves us with the outermost expression:

... (`foldr` string-append-with-space " " ..)

This one reduces the sorted list of first names to a single string, using a function named `string-append-with-space`. With such a suggestive name, you can easily imagine now that this reduction concatenates all the strings in the desired way—even if you do not quite

understand the details of how the combination of `foldr` and `string-append-with-space` works.

Exercise 261. You can design `build-list` and `foldl` with the design recipes that you know, but they are not going to be like the ones that ISL provides. For example, the design of your own `foldl` function requires a use of the list `reverse` function:

```
; [X Y -> Y] Y [List-of X] -> Y
; my-foldl works just like foldl
(check-expect (my-foldl cons '() '(a b c)) (foldl cons '() '(a b c)))
(check-expect (my-foldl / 1 '(6 3 2)) (foldl / 1 '(6 3 2)))
(define (my-foldl f e l)
  (foldr f e (reverse l)))
```

Design `my-build-list`, which works just like `build-list`. **Hint** Recall the add-at-end function from [exercise 195](#). **Note on Design** Accumulators covers the concepts that you need to design these functions properly. ■

18.2 Local Function Definitions

Take a second look at [figure 72](#). The `string-append-with-space` function clearly plays a subordinate role and has no use outside of this narrow context. Almost all programming languages support some way for stating this relationship as a part of the program. The idea is called a *local definition*, sometimes also a *private definition*. In ISL, `local` expressions introduce locally defined functions, variables, and structure types, and this section introduces the mechanics of `local`.

Here is a revision of [figure 72](#) using `local`:

```
; [List-of Addr] -> String
; creates a string of first names, sorted in alphabetical order,
; separated and surrounded by blank spaces
(define (listing.v2 l)
  (local (; String String -> String
          ; concatenates two strings and prefixes with space
          (define (string-append-with-space s t)
            (string-append " " s t)))
    ; - IN -
    (foldr string-append-with-space
      " "
      (sort (map address-first-name l)
        string<?))))
```

The body of the `listing.v2` function is now a `local` expression, which consists of two pieces: a sequence of definitions and a body expression. All local definitions are visible everywhere within the opening parenthesis of `local` and the closing one and only there.

In this example, the sequence of definitions consists of a single function definition, the one for `string-append-with-space`. The body of `local` is the body of the original `listing` function. Its reference to `string-append-with-space` is now resolved locally, that is, there is no need to look in the global sequence of definitions. Conversely, outside of the `local` expression, it is impossible to refer to `string-append-with-space`. Since there is no such reference in the original program, it remains intact and you can confirm this with the test suite.

In general, a `local` expression has this shape:

```
(local ( ... sequence of definitions ...) body-expression)
```

The evaluation of such an expression proceeds like the evaluation of a complete program. First, the definitions are set up, which may involve the evaluation of the right-hand side of a constant definition. Just as with the top-level definitions that you know and love, the definitions in a `local` expression may refer to each other. They may also refer to parameters of the surrounding function. Second, the *body-expression* is evaluated. The result of *body-expression* is the result of the `local` expression. It is often helpful to separate the `local definitions` from the body-expression with a comment; we use – IN – because the word suggests that the definitions are available in a certain expression.

```
; [List-of Number] [Number Number -> Boolean] -> [List-of Number]
(define (sort-cmp alon0 cmp)
  (local ( ; [List-of Number] -> [List-of Number]
            ; produces a variant of alon sorted by cmp
            (define (isort alon)
              (cond
                [(empty? alon) '()]
                [else (insert (first alon) (isort (rest alon))))]

            ; Number [List-of Number] -> [List-of Number]
            ; inserts n into the sorted list of numbers alon
            (define (insert n alon)
              (cond
                [(empty? alon) (cons n '())]
                [else (if (cmp n (first alon))
                          (cons n alon)
                          (cons (first alon)
                                (insert n (rest alon))))])))

  (isort alon0)))
```

Figure 73: Local, interconnected function definitions

Let's take a second look at the above example. Now that we know that `local` may introduce several local definitions, we can further improve the readability of the listing function:

```
(define (listing.v3 l)
  (local ( ; String String -> String
            ; concatenates two strings and prefixes with space
            (define (string-append-with-space s t)
              (string-append " " s t))
            (define first-names (map address-first-name l))
            (define sorted-names (sort first-names string<?)))
  ; – IN –
  (foldr string-append-with-space " " sorted-names)))
```

The most visually appealing difference concerns the body of the `local` expression. It is no longer a three-line expression but a single line and its purpose is clearly expressed, especially because it uses an aptly named variable to refer to the sorted list of names. An inspection of the definition of `sorted-names` shows another such uses of a `local` definition: `first-names`. In general, when you see a multi-line, deeply nested expression you may wish to formulate `local` definitions with properly named variables to bring

across what the expressions compute. Future readers will appreciate it because they can often comprehend the code just by looking at the variable names and the body of the `local` expression.

For a second example, consider [figure 73](#). The organization of this definition tells the reader that `sort-cmp` needs two auxiliary functions: `isort` and `insert`. Note the reference to `cmp` in the latter, which tells the reader that the comparison function remains the same for the entire sorting process.

By making `insert` local, it also becomes impossible to abuse `insert`. Re-read its purpose statement. The adjective “sorted” means that a program should use `insert` only if the second argument is already sorted. As long as `insert` is defined at the top-level, nothing guarantees that `insert` is always used properly. Once its definition is local to the `sort-cmp` function, the proper use is guaranteed—because `isort` applies `insert` to a sorted version of (`rest` alone).

Exercise 262. Use a `local` expression to organize the functions for drawing a polygon in [figure 57](#). If a globally defined functions is widely useful, do not make it local. ■

Exercise 263. Use a `local` expression to organize the functions for rearranging words from [Word Games, the Heart of the Problem](#). ■

```
; Nelon -> Number
; determines the smallest number on l
(define (inf l)
  (cond
    [(empty? (rest l)) (first l)]
    [else
      (local ((define smallest-in-rest (inf (rest l))))
        (cond
          [(< (first l) smallest-in-rest) (first l)]
          [else smallest-in-rest]))]))
```

Figure 74: Using `local` to improve performance

For a final example, let us take a look at the `inf` function from [figure 68](#). Using `local` to improve this function’s performance yields the definition in [figure 74](#). Here the `local` expression shows up in the middle of a `cond` expression. It also doesn’t define a function but a variable whose initial value is the result of a natural recursion. Now recall that the evaluation of a `local` expression evaluates the definitions once and for all before the body is evaluated. What this means here is that `(inf (rest l))` is evaluated once, but the body of the `local` expression refers to the result twice. Put differently, the use of `local` saves the re-evaluation of `(inf (rest l))` at each stage in the computation. To confirm this insight, re-evaluate the above version of `inf` on the inputs suggested in [exercise 240](#).

Exercise 264. Consider the following function definition:

```
; Inventory -> Inventory
; creates an Inventory from an-inv for all
; those items that cost less than $1
(define (extract1 an-inv)
  (cond
    [(empty? an-inv) '()]
    [else (cond
              [(<= (ir-price (first an-inv)) 1.0)
```

```
(cons (first an-inv) (extract1 (rest an-inv)))
[else (extract1 (rest an-inv))]))])
```

Both clauses in the nested `cond` expression extract the first item from `an-inv` and both compute `(extract1 (rest an-inv))`. Use a `local` expression to name the repeated expressions. Does this help increase the speed at which the function computes its result? Significantly? A little bit? Not at all? ■

Exercise 265. The `sort>` function consumes a list of numbers and produces a sorted version:

```
; Lon -> Lon
; constructs a list from the items in l in descending order
(define (sort> l0)
  (local (; Lon -> Lon
          (define (sort l)
            (cond
              [(empty? l) '()]
              [else (insert (first l) (sort (rest l))))])
          ; Number Lon -> Lon
          (define (insert an l)
            (cond
              [(empty? l) (list an)]
              [else
                (cond
                  [(> an (first l)) (cons an l)]
                  [else (cons (first l) (insert an (rest l))))]])))
        (sort l0)))
```

Create a test suite for `sort>`.

Design the function `sort-<`, which sorts lists of numbers in **ascending** order.

Create `sort-a`, which abstracts `sort>` and `sort-<`. It consumes the comparison operation in addition to the list of numbers. Define versions `sort>` and `sort-<` in terms of `sort-a`.

Use `sort-a` to define a function that sorts a list of strings by their lengths, both in descending and ascending order.

Later we will introduce several different ways to sort lists of numbers, all of them faster than `sort-a`. If you then change `sort-a`, all uses of `sort-a` will benefit. ■

18.3 ... Add Expressive Power

The third and last example illustrates how `local` adds expressive power to BSL and BSL+. **Finite State Machines** presents the design of a world program that simulates how a finite state machine recognizes sequences of key strokes. While the data analysis leads in a natural manner to the data definitions in [figure 64](#), an attempt to design the main function of the world program fails. Specifically, even though the given finite state machine remains the same over the course of the simulation, the state of the world must include it so that the program can transition from one state to the next when the player presses a key.

```
; FSM FSM-State -> FSM-State
; match the keys pressed by a player with the given FSM
```

```

(define (simulate fsm s0)
  (local (; State of the World: FSM-State

    ; FSM-State KeyEvent -> FSM-State
    ; finds the next state in the transition table of fsm0
    (define (find-next-state s key-event)
      (find fsm s)))

  ; NOW LAUNCH THE WORLD
  (big-bang s0
    [to-draw state-as-colored-square]
    [on-key find-next-state])))

; FSM-State -> Image
; renders current state as colored square
(define (state-as-colored-square s)
  (square 100 "solid" s))

; FSM FSM-State -> FSM-State
; finds the state matching current in the given transition table (fsm)
(define (find transitions current)
  (cond
    [(empty? transitions) (error "not found")]
    [else
      (local ((define s (first transitions)))
        (if (state=? (transition-current s) current)
            (transition-next s)
            (find (rest transitions) current))))]))

```

Figure 75: Power from local function definitions

Figure 75 shows an ISL solution to the problem. It uses `local` function definitions and can thus equate the state of the world with the current state of the finite state machine. Specifically, `simulate` locally defines the key event handler, which consumes only the current state of the world and the `KeyEvent` that represents the player's key stroke. Because this locally defined function can refer to the given finite state machine `fsm`, it is possible to find the next state in the transition table—even though the transition table—finite state machine representation—is **not** an argument to this function.

As the figure also shows, all other functions are defined in parallel to the main function. This includes the function `find`, which performs the actual search in the transition table. The key improvement over BSL is that a locally defined function can reference **both** parameters to the function **and** globally defined auxiliary functions.

In short, this program organization signals to a future reader exactly the insights that the data analysis stage of the design recipe for world programs finds. First, the given representation of the finite state machine remains unchanged. Second, what changes over the course of the simulation is the current state of the finite machine.

The lesson is that the chosen programming language affects a programmer's ability to express solutions, and a future reader's ability to recognize the design insight of the original creator.

Exercise 266. Design the function `identity`, which creates diagonal—also called *identity*—squares of `0`s and `1`s:

⋮

```
> (diagonal 0)
'()
> (diagonal 1)
'((1))
> (diagonal 4)
'((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1))
> (diagonal 5)
'((1 0 0 0 0) (0 1 0 0 0) (0 0 1 0 0) (0 0 0 1 0) (0 0 0 0 1))
```

Do **not** use existing abstractions, but use the familiar structural design recipe and exploit the additional expressive power of `local`. ■

18.4 Computing with `local`

ISL's `local` expression calls for the first rule of calculation that is truly beyond pre-algebra knowledge. The rule is relatively simple but quite unusual. It's best illustrated with some examples. We start with a second look at this definition:

```
(define (simulate fsm s0)
  (local ((define (find-next-state s key-event)
            (find fsm s)))
    (big-bang s0
      [to-draw state-as-colored-square]
      [on-key find-next-state])))
```

Now suppose we wish to calculate what DrRacket might produce for

```
(simulate AN-FSM A-STATE)
```

where AN-FSM and A-STATE are unknown values. Using the usual substitution rule, we proceed as follows:

```
==

(local ((define (find-next-state s key-event)
            (find AN-FSM A-STATE)))
  (big-bang s0
    [to-draw state-as-colored-square]
    [on-key find-next-state]))
```

This is the body of `simulate` with all occurrences of the parameters, `fsm` and `s`, replaced by the argument values, AN-FSM and A-STATE, respectively.

And at this point we are stuck because the expression is a `local` expression, and we don't know how to calculate with it. So here we go. To deal with a `local` expression in a program evaluation, we proceed in two steps:

1. We rename the locally defined constants and functions to use names that aren't used elsewhere in the program.
2. We lift the definitions in the `local` expression to where we keep the top-level definitions and evaluate the body of the `local` expression next.

Stop! Don't think. Accept the two steps for now. We explain their rationale below.

Let's apply these two steps to our running example. Here is the result of step 1:

```

==

(local ((define (find-next-state-1 s key-event)
                 (find an-fsm a-state)))
       (big-bang s0
                  [to-draw state-as-colored-square]
                  [on-key find-next-state-1]))

```

Our choice is to append “-1” to the end of the function name. If this variant of the name already exists, we use “-2” instead, and so on.

We use \oplus to indicate the step produces two pieces.

So here is the result of step 2:

```

==

(define (find-next-state-1 s key-event)
  (find an-fsm a-state))

⊕

(big-bang s0
           [to-draw state-as-colored-square]
           [on-key find-next-state-1])

```

The result is an ordinary program: some globally defined constants and functions followed by an expression. The normal rules apply, and there is nothing else to say.

At this point, it is time to rationalize the two steps. For the renaming step, we use a variant of the `inf` function from [figure 74](#). Clearly,

```
(inf (list 2 1 3)) == 1
```

The question is whether you can show the calculations that DrRacket performs to determine this result.

The first step is straightforward:

```

(inf (list 2 1 3))
==

(cond
  [(empty? (rest (list 2 1 3))) (first (list 2 1 3))]
  [else
    (local ((define smallest-in-rest (inf (rest (list 2 1 3)))))
      (cond
        [(< (first (list 2 1 3)) smallest-in-rest) (first (list 2 1 3))]
        [else smallest-in-rest]))])

```

We substitute the argument, `(list 2 1 3)`, for the parameter, `l`.

Since the list clearly isn’t empty, we skip the steps for evaluating the conditional and focus on the next expression to be evaluated:

```

...
==

(local ((define smallest-in-rest (inf (rest (list 2 1 3)))))
  (cond
    [(< (first (list 2 1 3)) smallest-in-rest) (first (list 2 1 3))])

```

```
[else smallest-in-rest]))
```

Applying the two steps for the rule of `local` yields two parts: the local definition lifted to the top and the body of the `local` expression. Here is how we write this down:

```
==  
(define smallest-in-rest-1 (inf (rest (list 2 1 3))))  
⊕  
(cond  
  [(< (first (list 2 1 3)) smallest-in-rest-1 3) (first (list 2 1 3))]  
  [else smallest-in-rest-1])
```

Curiously, the next expression we need to evaluate is the right-hand side of a constant definition in a `local` expression. But the point of computing is that you can replace expressions with their equivalents wherever you want:

```
==  
(define smallest-in-rest-1  
  (cond  
    [(empty? (rest (list 1 3))) (first (list 1 3))]  
    [else  
      (local ((define smallest-in-rest (inf (rest (list 1 3)))))  
        (cond  
          [(< (first (list 1 3)) smallest-in-rest) (first (list 1 3))]  
          [else smallest-in-rest]))])  
⊕  
(cond  
  [(< (first (list 2 1 3)) smallest-in-rest-1) (first (list 2 1 3))]  
  [else smallest-in-rest-1])
```

Once again, we skip the conditional steps and focus on the `else` clause, which is also a `local` expression. Indeed it is another variant of the `local` expression in the definition of `inf`, with a different list value substituted for the parameter:

```
(define smallest-in-rest-1  
  (local ((define smallest-in-rest (inf (rest (list 1 3)))))  
    (cond  
      [(< (first (list 1 3)) smallest-in-rest) (first (list 1 3))]  
      [else smallest-in-rest])))  
⊕  
(cond  
  [(< (first (list 2 1 3)) smallest-in-rest-3) (first (list 2 1 3))]  
  [else smallest-in-rest-3])
```

Because it originates from the same `local` expression in `inf`, it uses the same name for the constant, `smallest-in-rest`. **If we didn't rename local definitions before lifting them, we would introduce two conflicting definitions for the same name**, and conflicting definitions are catastrophic for mathematical calculations.

Here is how we continue:

```
==  
(define smallest-in-rest-2  
  (inf (rest (list 1 3))))  
⊕  
(define smallest-in-rest-2
```

```
(cond
  [(< (first (list 1 3)) smallest-in-rest-2) (first (list 1 3))]
  [else smallest-in-rest-2])
⊕
(cond
  [(< (first (list 2 1 3)) smallest-in-rest-2) (first (list 2 1 3))]
  [else smallest-in-rest-2])
```

The key is that we now have **two** definitions generated from **one and the same local** expression in the function definition. As a matter of fact we get one such definition per item in the given list (minus 1).

For the rest of the calculation, see [figure 76](#); it omits the very last step, which would require replacing the name of a constant with its value. The exercises request that you explore this example some more.

```
(inf (list 2 1 3)) (1)
...
==

(define smallest-in-rest-2
  (cond
    [(empty? (rest (list 3))) (first (list 3))]
    [else
      (local ((define smallest-in-rest (inf (rest (list 3)))))
        (cond
          [(< (first (list 3)) smallest-in-rest) (first (list 3))]
          [else smallest-in-rest]))])) (2)
⊕
(define smallest-in-rest-1
  (cond
    [(< (first (list 1 3)) smallest-in-rest-2) (first (list 1 3))]
    [else smallest-in-rest-2]))
⊕
(define smallest-in-rest-1
  (cond
    [(< (first (list 2 1 3)) smallest-in-rest-1) (first (list 2 1 3))]
    [else smallest-in-rest-1]))
==

(define smallest-in-rest-2 3) (3)
⊕
(define smallest-in-rest-1
  (cond
    [(< (first (list 1 3)) smallest-in-rest-2) (first (list 1 3))]
    [else smallest-in-rest-2]))
⊕
(define smallest-in-rest-1
  (cond
    [(< (first (list 2 1 3)) smallest-in-rest-1) (first (list 2 1 3))]
    [else smallest-in-rest-1]))
==

(define smallest-in-rest-2 3) (4)
⊕
(define smallest-in-rest-1 1)
⊕
(define smallest-in-rest-1
  (cond
    [(< (first (list 2 1 3)) smallest-in-rest-1) (first (list 2 1 3))]
    [else smallest-in-rest-1]))
==

(define smallest-in-rest-2 3) (5)
⊕
(define smallest-in-rest-1 1)
```

smallest-in-rest-1

Figure 76: Computing with `local`

Exercise 267. Use DrRacket's stepper to fill in the gaps in the calculation for `inf`, say, between step 2 and 3 in figure 76. ■

Exercise 268. Use DrRacket's stepper to calculate out how it evaluates

```
(sup (list 2 1 3))
```

where `sup` is the function from figure 68 equipped with a `local` expression. ■

For the explanation of the lifting step, we use a toy example that gets to the heart of the issue, namely, that functions are now values:

```
((local ((define (f x) (+ (* 4 (sqr x)) 3))) f)
  1)
```

Deep down we know that this is equivalent to `(f 1)` where

```
(define (f x) (+ (* 4 (sqr x)) 3))
```

but the rules of pre-algebra don't apply. The key is that **functions can be the result of expressions, including `local` expressions**. And the best way to think of this is to move such `local` definitions to the top and to deal with them like ordinary definitions. Doing so renders the definition visible for every step of the calculation. By now you also understand that the renaming step makes sure that the lifting of definitions does not accidentally conflate names or introduce conflicting definitions.

Here are the first two steps of the calculation:

```
((local ((define (f x) (+ (* 4 (sqr x)) 3))) f)
  1)
==
((local ((define (f-1 x) (+ (* 4 (sqr x)) 3))) f-1)
  1)
==
(define (f-1 x) (+ (* 4 (sqr x)) 3))
⊕
(f-1 1)
```

Remember that the second step of the `local` rule replaces the `local` expression with its body. In this case, the body is just the name of the function and its surrounding is an application to `1`. The rest is arithmetic:

```
(f-1 1) == (+ (* 4 (sqr 1)) 3) == 7
```

Exercise 269. Use DrRacket's stepper to fill in any gaps in the above calculation. ■

Exercise 270. Use DrRacket's stepper to find out how it evaluates this expression

```
((local ((define (f x) (+ x 3))
         (define (g x) (* x 4)))
  (if (odd? (f (g 1)))
      f
      g))
  2)
```

18.5 Using Abstractions, by Example

Now that you understand `local`, you are ready to use the abstractions from figure 71. Let us look at some examples, starting with this one:

Sample Problem: Design the function `add-3-to-all`. The function consumes a list of `Posns` and adds 3 to the x-coordinates of each of them.

If we follow the design recipe and take the problem statement as a purpose statement, we can quickly step through the first three steps:

```
; [List-of Posn] -> [List-of Posn]
; adds 3 to each x-coordinate on the given list

(check-expect (add-3-to-all (list (make-posn 30 10) (make-posn 0 0)))
               (list (make-posn 33 10) (make-posn 3 0)))

(define (add-3-to-all lop) '())
```

While you can run the program, doing so signals a failure in the one test case because the function returns the default value `'()`.

At this point, we stop and ask what kind of function we are dealing with. Clearly, `add-3-to-all` is clearly a list-processing function. The question is whether it is like any of the functions in figure 71. The signature tells us that `add-3-to-all` is a list-processing function that consumes and produces a list. In figure 71, we have several functions with similar signatures: `map`, `filter`, and `sort`.

The purpose statement and example also tell you that `add-3-to-all` deals with each `Posn` separately and assembles the results into a single list. Some reflection says that also confirms that the resulting list contains as many items as the given list. All this thinking points to one function—`map`—because the point of `filter` is to drop items from the list and `sort` has an extremely specific purpose.

Here is `map`'s signatures again:

```
; [X -> Y] [List-of X] -> [List-of Y]
```

It tells us that `map` consumes a function from `X` to `Y` and a list of `X`s. Given that `add-3-to-all` consumes a list of `Posns`, we know that `X` stands for `Posn`. Similarly, `add-3-to-all` is to produce a list of `Posns`, and this means we replace `Y` with `Posn`.

From the analysis of the signature we conclude that `map` could do the job of `add-3-to-all` if we can find an appropriate function from `Posns` to `Posns`. With `local`, we can express this idea as a template for `add-3-to-all`:

```
; [List-of Posn] -> [List-of Posn]
; adds 3 to each x-coordinate on the given list

(check-expect (add-3-to-all (list (make-posn 30 10) (make-posn 0 0)))
               (list (make-posn 33 10) (make-posn 3 0)))

(define (add-3-to-all lop)
```

```
(local (; Posn -> Posn
  ; ...
  (define (a-fun-from-posn-to-posn p)
    ... p ...))
  (map a-fun-from-posn-to-posn lop)))
```

Doing so reduces the problem to defining a function on `Posns`.

Given the example for `add-3-to-all` and the abstract example for `map`, you can actually imagine how the evaluation proceeds:

```
(add-3-to-all (list (make-posn 30 10) (make-posn 0 0)))
== 
(map a-fun (list (make-posn 30 10) (make-posn 0 0)))
== 
(list (a-fun (make-posn 30 10)) (a-fun (make-posn 0 0))))
```

And that shows how `a-fun` is applied to every single `Posn` on the given list, meaning it is its job to add `3` to the x-coordinate.

From here, it is straightforward to wrap up the definition:

```
; [List-of Posn] -> [List-of Posn]
; adds 3 to each x-coordinate on the given list

(check-expect (add-3-to-all (list (make-posn 30 10) (make-posn 0 0)))
              (list (make-posn 33 10) (make-posn 3 0)))

(define (add-3-to-all lop)
  (local (; Posn -> Posn
           ; adds 3 to the x-coordinate of the given Posn
           (define (add-3-to-one p)
             (make-posn (+ (posn-x p) 3) (posn-y p))))
    (map add-3-to-one lop)))
```

We chose `add-3-to-one` as the name for the local function because the name tells you what it computes. It adds `3` to (the x-coordinate of) one `Posn`—as opposed to `add-3-to-all`, which adds `3` to `all` given `Posns`. It is `map`'s task to apply `add-3-to-one` to each of the `Posns` on `lop` and to assemble the results into one list.

You may think now that using abstractions is hard. Keep in mind, though, that this first example spells out every single detail and that it does so because we wish to teach you how to pick the proper abstraction. Let us take a look at a second example a bit more quickly:

Sample Problem: Design a function that eliminates all `Posns` from a list that have a y-coordinate of larger than `100`.

The first two steps of the design recipe yield this:

```
; [List-of Posn] -> [List-of Posn]
; eliminates Posns whose y-coordinate is larger than 100

(check-expect (keep-good (list (make-posn 0 110) (make-posn 0 60)))
              (list (make-posn 0 60)))

(define (keep-good lop) '())
```

By now you may have guessed that this function is like `filter` whose purpose is to separate the “good” from the “bad.”

With `local` thrown in, the next step is also straightforward:

```
; [List-of Posn] -> [List-of Posn]
; eliminates Posns whose y-coordinate is larger than 100

(check-expect (keep-good (list (make-posn 0 110) (make-posn 0 60)))
               (list (make-posn 0 60)))

(define (keep-good lop)
  (local (; Posn -> Boolean
          ; should this Posn stay on the list
          (define (good? p) #true))
    (filter good? lop)))
```

The `local` function definition introduces the helper function needed for `filter` and the body of the `local` expression applies `filter` to this local function and the given list. The local function is called `good?` because `filter` retains all those items of `lop` for which `good?` produces `#true`.

Before you read on, analyze the signature of `filter` and `keep-good` and determine why the helper function consumes individual `Posns` and produces `Booleans`.

Putting all of our ideas together yields this definition:

```
; [List-of Posn] -> [List-of Posn]
; eliminates Posns whose y-coordinate is larger than 100

(check-expect (keep-good (list (make-posn 0 110) (make-posn 0 60)))
               (list (make-posn 0 60)))

(define (keep-good lop)
  (local (; Posn -> Posn
          ; should this Posn stay on the list
          (define (good? p)
            (not (> (posn-y p) 100))))
    (filter good? lop)))
```

Explain the definition of `good?` and simplify it.

Before we spell out a design recipe, let us deal with one more example:

Sample Problem: Design a function that determines whether any of a list of `Posns` is close to some given position `pt` where “close” means a distance of at most `5` pixels.

This problem clearly consists of two distinct parts: one concerns processing the list and another one calls for a function that determines whether the distance between a point and `pt` is “close.” Since this second part is unrelated to the reuse of abstractions for list traversals, we assume the existence of an appropriate function:

```
; Posn Posn Number -> Boolean
; is the distance between p and q less than d
(define (close-to p q d) ...)
```

You should complete this definition on your own.

As required by the problem statement, the function consumes two arguments—the list of **Posns** and the “given” point **pt**—and produces a **Boolean**:

```
; [List-of Posn] Posn -> Boolean
; is any Posn on lop close to pt

(check-expect (close? (list (make-posn 47 54) (make-posn 0 60))
                           (make-posn 50 50)))
               #true)

(define (close? lop pt) #false)
```

The latter point immediately differentiates this example from the preceding ones.

The **Boolean** range also gives away a clue with respect to [figure 71](#). Only two functions in this list produce **Boolean** values—**andmap** and **ormap**—and they must be primary candidates for defining **close?**’s body. While the explanation of **andmap** says that some property must hold for every item on the given list, the purpose statement for **ormap** tells us that it looks for only **one** such item. Given that **close?** just checks whether one of the **Posns** is close to **pt**, we should try **ormap** first.

Let us apply our standard “trick” of adding a **local** whose body uses the chosen abstraction on some locally defined function and the given list argument:

```
; [List-of Posn] Posn -> Boolean
(define (close? lop pt)
  (local (; Posn -> Boolean
          ; ...
          (define (is-one-close? p)
            ...))
    (ormap close-to? lop)))
```

Following the description of **ormap**, the local function consumes one item of the list at a time. This accounts for the **Posn** part of its signature. Also, the local function is expected to produce **#true** or **#false**, and **ormap** checks these results until it finds **#true**.

Here is a comparison of the signature of **ormap** and **close?**, starting with the former:

```
; [X -> Boolean] [List-of X] -> Boolean
```

In our case, the list argument is a list of **Posns**. Hence **X** stands for **Posn**, which explains what **is-one-close?** consumes. Furthermore, it determines that the result of the local function must be **Boolean** so that it can work as the first argument to **ormap**.

The rest of the work requires just a bit more thinking. While **is-one-close?** consumes one argument—a **Posn**—the **close-to** function consumes three: two **Posns** and a “tolerance” value. While the argument of **is-one-close?** is one of the two **Posns**, it is also obvious that the other one is **pt**, the argument of **close?** itself. Naturally the “tolerance” argument is **5**, as stated in the problem:

```
; [List-of Posn] -> Boolean
(define (close? lop pt)
  (local (; Posn -> Boolean
          ; is one shot close to pt
```

```

(define (is-one-close? p)
  (close-to p pt CLOSENESS)))
(ormap is-one-close? lop))

(define CLOSENESS 5)

```

Note two properties of this definition. First, we stick to the rule that constants deserve definitions. Second, the reference to `pt` in `is-one-close?` signals that this `Posn` stays the same for the entire traversal of `lop`.

18.6 Designing with Abstractions

Three examples suffice for formulating a design recipe for reusing abstractions:

1. Step 1 is to follow the **design recipe for functions** for three steps. Specifically, you should distill the problem statement into a signature, a purpose statement, an example, and a stub definition.

Consider the problem of defining a function that places small red circles on a 200 by 200 canvas for a given list of `Posns`. The first three steps design recipe yield this much:

```

; [List-of Posn] -> Image
; adds the Posns on lop to the empty scene

(check-expect (dots (list (make-posn 12 31)))
                (place-image DOT 12 32 BACKGROUND))

(define (dots lop)
  BACKGROUND)

(define BACKGROUND (empty-scene 200 200))
(define DOT (circle 5 "solid" "red"))

```

2. Next we exploit the signature and purpose statement to find a matching abstraction. To **match** means to pick an abstraction whose purpose is more general than the one for the function to be designed; it also means that the signatures are related. It is often best to start with the desired output and to find an abstraction that has the same or a more general output.

For our running example, the desired output is an `Image`. While none of the available abstractions produces an image, two of them have a variable to the right of

```

; foldr : [X Y -> Y] Y [List-of X] -> Y
; foldl : [X Y -> Y] Y [List-of X] -> Y

```

meaning we can plug in any data collection we want. If we do use `Image`, the signature on the left of `->` demands a helper function that consumes an `X` together with an `Image` and produces an `Image`. Furthermore, since the given list contains `Posns`, `X` does stand for the `Posn` collection.

3. Write down **a template**. For the reuse of abstractions a template uses `local` for two different purposes. The first one is to note which abstraction to use and how in the body of the `local` expression. The second one is to write down a stub for the helper function: its signature, its purpose (optionally), and its header. Keep in mind that the signature comparison in the preceding step suggests most of the signature for the auxiliary

function.

Here is what this template looks like for our running example if we choose `foldr`:

```
(define (dots lop)
  (local (; Posn Image -> Image
            (define (add-one-dot p scene) ...))
    (foldr add-one-dot BACKGROUND lop)))
```

The `foldr` description calls for a “base” `Image` value, to be used if or when the list is empty. In our case, we clearly want the empty canvas for this case. Otherwise, `foldr` uses a helper function and traverses the list of `Posns`.

4. Finally, it is time to define the helper function inside `local`. In most cases, this auxiliary function consumes relatively simple kinds of data, like those encountered in [Fixed-Size Data](#). You know how to design those in principle. The only difference is that now you may not only use the function’s arguments and global constants but also the arguments of the surrounding function.

In our running example, the purpose of the helper function is to add one dot to the given scene, which you can (1) guess or (2) derive from the example:

```
(define (dots lop)
  (local (; Posn Image -> Image
            ; adds a DOT at p to scene
            (define (add-one-dot p scene)
              (place-image DOT (posn-x p) (posn-y p) scene)))
    (foldr add-one-dot BACKGROUND lop)))
```

5. The last step is **to test** the definition in the usual manner.

For abstract functions, it is occasionally possible to use the abstract example of their purpose statement to confirm their workings at a more general level. You may wish to use the abstract example for `foldr` to confirm that `dots` does add one dot after another to the background scene.

In the third step, we picked `foldr` without further ado. Experiment with `foldl` to see how it would help complete this function. Functions like `foldl` and `foldr` are well-known and are spreading in usage in various forms. Becoming familiar with them is a good idea and the next section will help.

18.7 Finger Exercises: Abstraction

Each of the following set of exercises suggests small practice problems for specific abstractions in ISL.

Exercise 271. Use `map` to define the function `convert-euro`, which converts a list of US\$ amounts into a list of € amounts based on an exchange rate of €1.22 per US\$.

Also use `map` to define `convertFC`, which converts a list of Fahrenheit measurements to a list of Celsius measurements.

Finally, try your hands at `translate`, a function that translates a list of `Posns` into a list of list of pairs of numbers, i.e., [List-of [list Number Number]]. ▀

Exercise 272. An inventory record specifies the name of an item, a description, the

acquisition price, and the recommended sales price.

Define a function that sorts a list of inventory records by the difference between the two prices. ▀

Exercise 273. Define `eliminate-exp`, which consumes a number, `ua` and a list of inventory records, and it produces a list of all those structures whose sales price is below `ua`.

Then use `filter` to define `recall`, which consumes the name of an inventory item, called `ty`, and a list of inventory records and which produces a list of inventory records that do not use the name `ty`.

In addition, define `selection`, which consumes two lists of names and selects all those from the second one that are also on the first. ▀

Exercise 274. Use `build-list` to define functions that

1. creates the list `(list 0 ... (- n 1))` for any natural number `n`;
2. creates the list `(list 1 ... n)` for any natural number `n`;
3. creates the list `(list 1 1/10 1/100 ...)` of `n` numbers for any natural number `n`;
4. creates the list of the first `n` even numbers;
5. creates a list of lists of `0` and `1` in a diagonal arrangement; see [exercise 266](#). ▀

Finally, define `tabulate` from [exercise 254](#) using `build-list`. ▀

Exercise 275. Use `ormap` to define `find-name`. The function consumes a name and a list of names. It determines whether any of the names on the latter are equal to or an extension of the former.

With `andmap` you can define a function that checks all names on a list of names start with the letter "`a`".

Should you use `ormap` or `andmap` to define a function that ensures that no name on some list exceeds some given width? ▀

Exercise 276. Recall that the `append` function in ISL concatenates the items of two lists or, equivalently, replaces `'()` at the end of the first list with the second list:

```
(equal? (append (list 1 2 3) (list 4 5 6 7 8))
        (list 1 2 3 4 5 6 7 8))
```

Use `foldr` to define `append-from-fold`. What happens if you replace `foldr` with `foldl`?

Now use one of the fold functions to define functions that compute the sum and the product, respectively, of a list of numbers.

With one of the fold functions, you can define a function that horizontally composes a list of `Images`. **Hints** (1) Look up `beside` and `empty-image`. Can you use the other fold function? Also define a function that stacks a list of images vertically. (2) Check for `above` in the libraries. ▀

Exercise 277. The fold functions are so powerful that you can define almost any list-processing functions with them. Use `fold` to define `map`. ▀

Exercise 278. Use existing abstractions to define the `prefixes` and `suffixes` functions from [exercise 192](#). Ensure that they pass the same tests as the original function. ▀

18.8 Projects: Abstraction

Now that you have some experience with the existing list-processing abstractions in ISL, it is time to tackle some of the small projects for which you already have programs. The challenge is to look for two kinds of improvements. First, inspect the programs for functions that traverse lists. For these functions, you already have signatures, purpose statements, tests, and working definitions that pass the tests. Change the definitions to use abstractions from [figure 71](#). Second, also determine whether there are opportunities to create new abstractions. Indeed, you might be able to abstract across these classes of programs and provide generalized functions that help you write additional programs.

You may wish to tackle these exercises again after studying [Nameless Functions](#).

Exercise 279. Real-world Data: Dictionaries deals with relatively simple tasks relating to English dictionaries. Two of them call for re-formulation with existing abstractions:

- Design `most-frequent`. The function consumes a `Dictionary` and produces the `Letter-Count` for the letter that is most frequently used as the first one in the words of the given `Dictionary`.
- Design `words-by-first-letter`. The function consumes a `Dictionary` and produces a list of `Dictionary`s, one per `Letter`. Do **not** include '()' if there are no words for some letter; ignore the empty grouping instead.

For the data definitions, see [figure 58](#). ▀

Exercise 280. Real-world Data: iTunes explains how to analyze the information in an iTunes XML library.

- Design `select-album-date`. The function consumes the title of an album, a date, and an `LTracks`. It extracts from the latter the list of tracks that belong to the given album and have been played after the given date.
- Design `select-albums`. The function consumes an `LTracks`. It produce a list of `LTracks`, one per album. Each album is uniquely identified by its title and shows up in the result only once.

See [figure 60](#) for the data definitions and functions that the `2htdp/itunes` library exports. ▀

Exercise 281. Full Space War spells out a game of space war. In the basic version, a UFO descends and a player defends with a tank. One additional suggestion is to equip the UFO with charges that it can drop at the tank; the tank is destroyed if a charge comes close enough.

Inspect the code of your project for places where it can benefit from existing abstraction, that is, processing lists of shots or charges.

Once you have simplified the code with the use of existing abstractions look for opportunities to create abstractions. Consider moving lists of objects as one example where abstraction may pay off. ▀

Exercise 282. *Feeding Worms* explains how another one of the oldest computer games work. The game features a worm that moves at a constant speed in a player-controlled direction. When it encounters food, it eats the food and grows. When it runs into the wall or into itself, the game is over.

This project can also benefit from the abstract list-processing in ISL. Look for places to use them and replace existing code one piece at a time, relying on the tests to ensure the program works. ■

19 Nameless Functions

Using abstract functions needs functions as arguments. Occasionally these functions are existing primitive functions, library functions, or functions that you defined:

- (`(build-list n add1)`) creates (`(list 1 ... n)`);
- the expression (`(foldr cons another-list one-list)`) concatenates the items on `one-list` and `another-list` into a single list; and
- (`(foldr above a-list-of-images)`) stacks the images on the given list.

At other times, it requires the definition of a simple helper function, a definition that often consists of a single line. Consider this use of `filter`:

```
; [List-of IR] String -> Boolean
(define (find aloir threshold)
  (local (; IR -> Boolean
          (define (acceptable? ir)
            (<= (ir-price ir) threshold)))
         (filter acceptable? aloir)))
```

It finds all items on an inventory list whose price is below `threshold`. The auxiliary function is nearly trivial yet its definition takes up three lines.

This situation calls for an improvement to the language. Programmers should be able to create such small and insignificant functions without much effort. The next level in our hierarchy of teaching languages,

“Intermediate Student Language with `lambda`” solves the problem—with `lambda`—and this chapter introduces the concept: its syntax, its meaning, and its pragmatics. With `lambda`, the above definition is, conceptually speaking, a one-liner:

In DrRacket, choose “Intermediate Student Language with `lambda`” from the “How to Design Programs” submenu in the “Language” menu.—The history of `lambda` is intimately involved with the early history of programming and programming language design. When you have time, you should chase down the origins of `lambda`.

```
; [List-of IR] String -> Boolean
(define (find aloir threshold)
  (filter (lambda (ir) (<= (ir-price ir) threshold)? aloir)))
```

The first two sections focus the mechanics of `lambda`; the remaining ones use `lambda` for instantiating abstractions, for testing and specifying, and for representing infinite data.

19.1 Functions from `lambda`

A `lambda` expression creates a nameless function. Its syntax is straightforward:

```
| (lambda (variable-1 ... variable-N) expression)
```

Its distinguishing characteristic is the keyword `lambda`. The keyword is followed by a sequence of variables, enclosed in a pair of parentheses. The last piece is an arbitrary expression, and it computes the result of the function when it is given values for its parameters.

Here are three simple examples, all of which consume one argument:

1. `(lambda (x) (* 10 x))`, which assumes that the argument is a number and computes the exponent of 10 to the number;
2. `(lambda (name) (string-append "hello, " name ", how are you?"))`, which consumes a string and creates a greeting with `string-append`; and
3. `(lambda (ir) (<= (ir-price ir) threshold))`, which is a function on an IR struct that extracts the price and compares it with some `threshold` value.

This way of thinking about `lambda` shows one more time why the rule for computing with `local` is complicated.

One way to understand how `lambda` works, is to view it as an abbreviation for a `local` expression. For example,

```
| (lambda (x) (* 10 x))
```

is short for

```
| (local ((define some-random-name (lambda (x) (* 10 x))))
       some-random-name)
```

This “trick” works in general as long as `some-random-name` does not appear in the body of the function. What this means is that `lambda` creates a function with a name that nobody knows. If nobody knows the name, it might as well be nameless or anonymous.

A nameless function is a value like any other value. The only interesting parts are the function parameters—the sequence of names in parentheses—and the function body—the expression in the third position.

To use a function created from `lambda` expression, you apply it to the correct number of arguments. It works as expected:

```
> ((lambda (x) (* 10 x)) 2)
20
> ((lambda (name rst) (string-append name ", " rst)) "Robby" "etc.")
"Robby, etc."
> ((lambda (ir) (<= (ir-price ir) threshold)) (make-ir "bear" 10))
#true
```

Note how the second sample function requires two arguments and that the last example assumes a definition for `threshold` in the definitions window such as this one:

```
| (define threshold 20)
```

The result of the last example is `#true` because the price field of the inventory record contains `10` and `10` is less than `20`.

Now the important point is that these nameless functions can be used wherever a function is required, including with the abstractions from figure 71:

```
> (map (lambda (x) (* 10 x))
      '(1 2 3))
'(10 20 30)
> (foldl (lambda (name rst) (string-append name ", " rst)) "etc."
      '("Matthew" "Robby"))
"Robby, Matthew, etc."
> (filter (lambda (ir) (<= (ir-price ir) threshold))
      (list (make-ir "bear" 10) (make-ir "doll" 33)))
'(#<ir>)
```

Once again, the last example assumes a definition for `threshold`.

Exercise 283. Decide which of the following phrases are legal `lambda` expressions:

1. `(lambda (x y) (x y y))`
2. `(lambda () 10)`
3. `(lambda (x) x)`
4. `(lambda (x y) x)`
5. `(lambda x 10)`

Explain why they are legal or illegal. If in doubt, experiment in the interactions area. ▀

Exercise 284. Calculate the result of the following expressions:

1. `((lambda (x y)
 (+ x (* x y)))
 1 2)`
2. `((lambda (x y)
 (+ x
 (local ((define z (* y y)))
 (+ (* 3 z)
 (/ 1 x)))))

 1 2)`
3. `((lambda (x y)
 (+ x
 ((lambda (z)
 (+ (* 3 z)
 (/ 1 z)))
 (* y y))))
 1 2)`

Check your results in DrRacket. ▀

Exercise 285. Write down a `lambda` expression that

1. consumes a number and decides whether it is less than `10`;

2. consumes two numbers, multiplies them, and turns the result into a string;
3. consumes two inventory records and compares them by price;
4. consumes a natural number and produces 0 if it is even and 1 if it is odd; and
5. consumes an `Posn` p together with a rectangular `Image` and adds a red 3-pixel dot to the image at p .

Demonstrate how to use these functions in the interactions area. ▶

19.2 Computing with `lambda`

The insight that `lambda` abbreviates a certain kind of `local` also connects constant definitions and function definitions. Instead of viewing function definitions as given, we can take `lambda`s as another fundamental concept and say that a function definition abbreviates a plain constant definition with a `lambda` expression on the right-hand side.

Alonzo Church, who invented `lambda` in the late 1920s, aimed to create a unifying theory of functions in mathematics. From his work we know that from a purely theoretical perspective, a language does not need `local` once it has `lambda`. But the margin of this page is too small to explain this idea properly. If you are curious, read up on the [Y combinator](#).

It's best to look at some concrete examples:

<code>(define (f x)</code>	is short for	<code>(define f</code>
<code>(* 10 x))</code>		<code>(lambda (x)</code>
		<code>(* 10 x)))</code>

What this line says, is that a function definition consists of two steps: the creation of the function and its naming. Here, the `lambda` on the right-hand side creates a function of one argument x that computes $10 \cdot x$; it is `define` that names the `lambda` expression f . We do give names to functions for two distinct reasons. On one hand, a function is often called more than once from other functions, and we wouldn't want to spell out the function with a `lambda` each time it is called. On the other hand, functions are often recursive because they process recursive forms of data, and naming functions makes it easy to create recursive functions.

Exercise 286. Experiment with the above definitions in DrRacket.

Also add

```
;; Number -> Boolean
(define (compare x)
  (= (f-plain x) (f-lambda x)))
```

to the definitions area after renaming the left-hand f to $f\text{-plain}$ and the right-hand one to $f\text{-lambda}$. Then run

```
(compare (random 100000))
```

a few times to make sure the two functions agree on all kinds of random inputs. ▶

If function definitions are just abbreviations for constant definitions, we can replace the

function name by its `lambda` expression wherever we see it:

```
(f (f 42))
== 
((lambda (x) (* 10 x)) ((lambda (x) (* 10 x)) 42))
```

Strangely though, this substitution appears to create an expression that violates the grammar as we know it. To be precise, it generates an application expressions whose function position is a `lambda` expression.

The point is that ISL+'s differs from ISL in **two** aspects: it obviously comes with `lambda` expressions but it also allows arbitrary expressions to show up in the function position of an application. This means that you may need to evaluate the function position before you can proceed with an application, but you know how to evaluate most expressions. Of course, the real difference is that the evaluation of an expression may yield a `lambda` expression. Functions really are values. The following grammar revises the one from [Intermezzo: BSL](#) to summarize these differences:

```
expr = ...
| (expr expr ...)

value = ...
| (lambda (variable variable ...) expr)
```

What you really need to know, is how to evaluate the application of a `lambda` expression to arguments, and that is surprisingly straightforward:

Church stated the *beta axiom* roughly like this.

```
((lambda (x-1 ... x-n) f-body) v-1 ... v-n) == f-body
; with all occurrences of x-1 ... x-n
; replaced with v-1 ... v-n, respectively
```

That is, the application of a `lambda` expression proceeds just like that of an ordinary function. We replace the parameters of the function with the actual argument values and compute the value of the function body.

Here is how to use this law on the first few examples in this chapter:

1.

```
((lambda (x) (* 10 x)) 2)
==
(* 10 2)
==
20
```
2.

```
((lambda (name rst) (string-append name " " rst)) "Robby" "etc.")
 ==
(string-append "Robby" " " "etc.")
 ==
"Robby, etc."
```

```

3. ((lambda (ir) (<= (ir-price ir) threshold)) (make-ir "bear" 10))
== 
(<= (ir-price (make-ir "bear" 10)) threshold)
== 
(<= 10 threshold)
== 
#true

```

assuming that `threshold` is larger than or equal to `10`.

Exercise 287. Confirm that DrRacket's stepper can deal with `lambda` expressions. ■

Exercise 288. Use DrRacket's stepper to determine how it evaluates these expressions:

```

(map (lambda (x) (* 10 x))
  '(1 2 3))

(foldl (lambda (name rst) (string-append name ", " rst)) "etc."
  '("Matthew" "Robby"))

(filter (lambda (ir) (<= (ir-price ir) threshold))
  (list (make-ir "bear" 10) (make-ir "doll" 33)))

```

Exercise 289. Step through the evaluation of this expression:

```
((lambda (x) x) (lambda (x) x))
```

Explain what this means.

Now step through this one:

```
((lambda (x) (x x)) (lambda (x) x))
```

Stop! What do you think we should try next?

Yes, try to evaluate

```
((lambda (x) (x x)) (lambda (x) (x x)))
```

Be ready to hit *STOP*. ■

19.3 Abstracting with `lambda`

Although it may take you a bit to get used to `lambda` notation, you will soon notice that `lambda` makes short functions much more readable than `local` definitions. Indeed, you will find that you can adapt step 4 of the design recipe from [Designing with Abstractions](#) to use `lambda` instead of `local`. Consider the running example from that section. Its template based on `local` is this:

```

(define (dots lop)
  (local (; Posn Image -> Image
          (define (add-one-dot p scene) ...))
    (foldr add-one-dot BACKGROUND lop)))

```

If you spell out the parameters so that their names include signatures, you can easily pack all the information from `local` into a single `lambda`:

```
(define (dots lop)
  (foldr (lambda (one-posn scene) ...) BACKGROUND lop))
```

From here, you should be able to complete the definition as well as from the original template:

```
(define (dots lop)
  (foldr (lambda (one-posn scene)
    (place-image DOT (posn-x one-posn) (posn-y one-posn) scene))
  BACKGROUND lop))
```

Let us illustrate `lambda` some more via examples from [Using Abstractions, by Example](#):

- the purpose of the first function is to add 3 to each x-coordinate on a given list of `Posns`:

```
; [List-of Posn] -> [List-of Posn]
; (add-3-to-all (list (make-posn 30 10) (make-posn 0 0)))
; should deliver
; (list (make-posn 33 10) (make-posn 3 0))

(define (add-3-to-all lop)
  (map (lambda (p) (make-posn (+ (posn-x p) 3) (posn-y p))) lop))
```

Because `map` expects a function of one argument, we use `(lambda (p) ...)`. The function then de-structures the given `Posn`, adds 3 to the x-coordinate, and repackages the data into a `Posn`.

- the second one eliminates `Posns` whose y-coordinate is larger than 100:

```
; [List-of Posn] -> [List-of Posn]
; (keep-good (list (make-posn 0 110) (make-posn 0 60)))
; should deliver
; (list (make-posn 0 60))

(define (keep-good lop)
  (filter (lambda (p) (<= (posn-y p) 100)) lop))
```

Here we know that `filter` needs a function of one argument that produces a `Boolean`. First, the `lambda` function extracts the y-coordinate from the `Posn` to which `filter` applies the function. Second, it checks whether it is less than or equal to 100, the desired limit.

- and the third one determines whether any `Posn` on `lop` is close to some given point:

```
; [List-of Posn] -> Boolean
; (close? (list (make-posn 3 4) (make-posn 9 9)) (make-posn 0 0))
; should be
; #true

(define (close? lop pt)
  (ormap (lambda (p) (close-to p pt CLOSENESS)) lop))

(define CLOSENESS 5)
```

Like the preceding two examples, `ormap` is a function that expects a function of one argument and applies this functional argument to every item on the given list. If any result is `#true`, `ormap` returns `#true`, too; if all results are `#false`, `ormap` produces

```
#false.
```

It is probably best to compare the definitions from [Using Abstractions, by Example](#) and the definitions above side by side. When you do so, you should notice how easy the transition from `local` to `lambda` is and how concise the `lambda` version is in comparison to the `local` version. Thus, if you are ever in doubt, design with `local` first and then convert this tested version into one that uses `lambda`. Keep in mind, however, that `lambda` is not a cure-all. The locally defined function comes with a name that explains its purpose and, if it is long, the use of an abstraction with a named function is much easier to understand than one with a large `lambda`.

The following exercises request that you solve the problems from [Finger Exercises: Abstraction with lambda](#) in ISL+ .

Exercise 290. Use `map` to define the function `convert-euro`, which converts a list of US\$ amounts into a list of € amounts based on an exchange rate of €1.22 per US\$.

Also use `map` to define `convertFC`, which converts a list of Fahrenheit measurements to a list of Celsius measurements.

Finally, try your hands at `translate`, a function that translates a list of `Posns` into a list of list of pairs of numbers, i.e., [List-of [list Number Number]]. ▀

Exercise 291. An inventory record specifies the name of an inventory item, a description, the acquisition price, and the recommended sales price.

Define a function that sorts a list of inventory records by the difference between the two prices. ▀

Exercise 292. Use `filter` to define `eliminate-exp`. The function consumes a number, `ua` and a list of inventory records (containing name and price), and it produces a list of all those structures whose acquisition price is below `ua`.

Then use `filter` to define `recall`, which consumes the name of an inventory item, called `ty`, and a list of inventory records and which produces a list of inventory records that do not use the name `ty`.

In addition, define `selection`, which consumes two lists of names and selects all those from the second one that are also on the first. ▀

Exercise 293. Use `build-list` and `lambda` to define functions that

1. creates the list `(list 0 ... (- n 1))` for any natural number `n`;
 2. creates the list `(list 1 ... n)` for any natural number `n`;
 3. creates the list `(list 1/10 1/100 ...)` of `n` numbers for any natural number `n`;
 4. creates the list of the first `n` even numbers;
 5. creates a list of lists of `0` and `1` in a diagonal arrangement; see [exercise 266](#).
- ▀

Exercise 294. Use `ormap` to define `find-name`. The function consumes a name and a list of names. It determines whether any of the names on the latter are equal to or an extension of the former.

With `andmap` you can define a function that checks all names on a list of names start with the letter "a".

Should you use `ormap` or `andmap` to define a function that ensures that no name on some list exceeds some given width? ■

Exercise 295. Recall that the `append` function in ISL concatenates the items of two lists or, equivalently, replaces `'()` at the end of the first list with the second list:

```
(equal? (append (list 1 2 3) (list 4 5 6 7 8))
        (list 1 2 3 4 5 6 7 8))
```

Use `foldr` to define append-from-fold. What happens if you replace `foldr` with `foldl`? ■

Now use one of the fold functions to define functions that compute the sum and the product, respectively, of a list of numbers.

With one of the fold functions, you can define a function that horizontally composes a list of Images. **Hints** (1) Look up `beside` and `empty-image`. Can you use the other fold function? Also define a function that stacks a list of images vertically. (2) Check for `above` in the libraries. ■

Exercise 296. The fold functions are so powerful that you can define almost any list-processing functions with them. Use `fold` to define map-via-fold, which simulates `map`. ■

19.4 Specifying with lambda

Figure 73 shows a generalized sorting function that consumes a list of values and a comparison function for such values. The details of the definitions don't matter for this section, and to help the discussion figure 77 displays the essence of the sorting function. The body of `sort-cmp` introduces two `local` auxiliary functions: `isort` and `insert`. In addition, the figure also comes with two test cases that illustrate the workings of `sort-cmp`. One demonstrates how the function works on strings and the other one for numbers.

```
; [List-of Number] [Number Number -> Boolean] -> [List-of Number]
; sort alon0 according to cmp

(check-expect (sort-cmp '("a" "c" "b") string<?) '("a" "b" "c"))
(check-expect (sort-cmp '(2 1 3 4 6 5) <) '(1 2 3 4 5 6))

(define (sort-cmp alon0 cmp)
  (local (; [List-of Number] -> [List-of Number]
          ; produces a variant of alon sorted by cmp
          (define (isort alon) ...)

          ; Number [List-of Number] -> [List-of Number]
          ; inserts n into the sorted list of numbers alon
          (define (insert n alon) ...))
    (isort alon0)))
```

Figure 77: A general sorting function

Now take a quick look at exercise 188. It asks you to formulate `check-satisfied` tests for `sort>` using `sorted>?`. The former is a function that sorts lists of numbers in descending order; the latter is a function that determines whether a list of numbers is sorted in

descending order. Hence, the solution of this exercise is

```
(check-satisfied (sort> '()) sorted>?)
(check-satisfied (sort> '(12 20 -5)) sorted>?)
(check-satisfied (sort> '(3 2 1)) sorted>?)
(check-satisfied (sort> '(1 2 3)) sorted>?)
```

The question is how to reformulate the tests for `sort-cmp` in an analogous manner.

Since `sort-cmp` consumes a comparison function together with a list, the generalized version of `sorted>?` must take one too. If so, the following test cases might look like this:

```
(check-satisfied (sort-cmp '("a" "c" "b") string<?) (sorted string<?))
(check-satisfied (sort-cmp '(2 1 3 4 6 5) <) (sorted <))
```

Both `(sorted string<?)` and `(sorted <)` must produce predicates. The first one checks whether some list of strings is sorted according to `string<?`, and the second one whether a list of numbers is sorted according to `<`.

We have thus worked out the desired signature and purpose of `sorted`:

```
; [X X -> Boolean] -> [ [List-of X] -> Boolean]
; produces a function that determines whether
; some list is sorted according to cmp
(define (sorted cmp)
  ...)
```

What we need to do now, is to go through the rest of the design process.

Let's first finish the header. Remember that the header produces a value that matches the signature and is likely to break most of the tests/examples. Here we need `sorted` to produce a function that consumes a list and produces a `Boolean`. With `lambda`, that's actually straightforward:

```
(define (sorted cmp)
  (lambda (l)
    #true))
```

Note this is your first function-producing function. Stop! Read the definition again. Can you explain this definition in your own words?

Next we need examples. According to our analysis of `sort-cmp`, `sorted` consumes predicates such as `string<?` and `<`, but clearly, `>` or `<=` should be acceptable, too, as should be comparison functions that you define. At first glance, this suggests test cases of the shape

```
(check-expect (sorted string<?) ...)
(check-expect (sorted <) ...)
```

but `(sorted ...)` produces a function and `check-expect` cannot compare functions. Indeed, according to [exercise 248](#) it impossible to compare functions. To formulate reasonable test cases, we need to apply the result of `(sorted ...)` to appropriate lists.

Here, then, are some test cases for `sorted`:

```
(check-expect [(sorted string<?) '("a" "b" "c")] #true)
(check-expect [(sorted <) '(1 2 3 4 5 6)] #true)
```

Notes (1) The square brackets are replaceable with parentheses, but they highlight that the first expression produces a function, which is then applied to arguments. (2) These test cases are derived from those for `sort-cmp` in [figure 77](#).

From this point on, the design is quite conventional. What we basically wish to design is a generalization of `sorted?>` from [Non-empty Lists](#); let's call this function `sorted/l`. What is unusual about `sorted/l`, is that it "lives" in the body of a `lambda` inside of `sorted`:

```
(define (sorted cmp)
  (lambda (l0)
    (local ((define (sorted/l l) ...))
      ...)))
```

In other words, `sorted/l` is defined locally and refers to `cmp`.

Exercise 297. Design the function `sorted?`, which comes with the following signature and purpose statement:

```
; [X X -> Boolean] [NEList-of X] -> Boolean
; determine whether l is sorted according to cmp

(check-expect (sorted? < '(1 2 3)) #true)
(check-expect (sorted? < '(2 1 3)) #false)

(define (sorted? cmp l)
  #false)
```

The wish list even includes examples. ■

[Figure 78](#) shows the result of the design process. The `sorted` function consumes a comparison function `cmp` and produces a predicate. The latter consumes a list `l0` and uses a locally defined function to determine whether all the items in `l0` are ordered via `cmp`. Specifically, the locally defined function checks a non-empty list; in the body of `local`, `sorted` first checks whether `l0` is empty, in which case it simply produces `#true` because the empty list is sorted.

Stop! Could you re-define `sorted` to use `sorted?` from [exercise 297](#)? Explain why `sorted/l` does not consume `cmp` as an argument?

```
; [X X -> Boolean] -> [[List-of X] -> Boolean]
; is the given list l0 sorted according to cmp
(define (sorted cmp)
  (lambda (l0)
    (local (; [NEList-of X] -> Boolean
            ; is l sorted according to cmp
            (define (sorted/l l)
              (cond
                [(empty? (rest l)) #true]
                [else (and (cmp (first l) (second l))
                           (sorted/l (rest l))))])
            (if (empty? l0) #true (sorted/l l0)))))
```

Figure 78: A curried predicate for checking the ordering of a list

The `sorted` function in [figure 78](#) is a *curried* version of a function that consumes two arguments: `cmp` and `l0`. Instead of

consuming two arguments at once, a curried function consumes one argument and then returns a function that consumes the second one.

The verb “curry” honors Haskell Curry, the second person to invent the idea. The first one was Moses Schönfinkel.

[Exercise 188](#) asks how to formulate a test case that exposes mistakes in sorting functions. Consider this definition:

```
; List-of-numbers -> List-of-numbers
; produces a sorted version of l
(define (sort-cmp/bad l)
  '(9 8 7 6 5 4 3 2 1 0))
```

Formulating such a test case with `check-expect` is straightforward. Doing the same with `check-satisfied` is difficult; it calls for a much stronger predicate than `sorted`.

To design a predicate that exposes `sort-cmp/bad` as flawed, we need to understand the purpose of `sort-cmp` or sorting in general. It clearly is unacceptable to throw away the given list and to produce some other list in its place. That’s why the purpose statement of `isort` says that the function “produces a **variant** of” the given list. “Variant” means that the function does not throw away any of the items on the given list.

With these thoughts in mind, we can now say that we want a predicate that checks whether the result is sorted **and** contains all the items from the given list:

```
; [List-of X] [X X -> Boolean] -> [ [List-of X] -> Boolean]
; is l0 sorted according to cmp
; are all items in list k members of list l0
(define (sorted-variant-of k cmp)
  (lambda (l0)
    #false))
```

The two lines of the purpose statement suggest examples:

```
(check-expect [(sorted-variant-of '(1 3 2) <) '(1 2 3)] #true)
(check-expect [(sorted-variant-of '(1 3 2) <) '(1 3)] #false)
```

Like `sorted`, `sorted-variant-of` consumes arguments and produces a function. For the first case, `sorted-variant-of` produces `#true`, because the `'(1 2 3)` is sorted and it contains all numbers in `'(1 3 2)`. In contrast, the function produces `#false` in the second case, because `'(1 3)` lacks `2` from the originally given list.

A two-line purpose statement suggests two tasks, and two tasks means that the function itself is a combination of two functions:

```
(define (sorted-variant-of k cmp)
  (lambda (l0)
    (and (sorted? cmp l0)
         (contains? l0 k))))
```

The body of the function is an `and` expression that combines two function calls. With the call to the `sorted?` function from [exercise 297](#), the function realizes the first line of the purpose statement. The second call, `(contains? k l0)`, is an implicit wish for an auxiliary function.

We immediately give the full definition:

```

; [Listof X] [Listof X] -> Boolean
; are all items in list k members of list l

(check-expect (contains? '(1 2 3) '(2 1 4 3)) #false)
(check-expect (contains? '(1 2 3 4) '(2 1 3)) #true)

(define (contains? l k)
  (andmap (lambda (item-in-k) (member? item-in-k l)) k))

```

On one hand, we have never explained how to systematically design a function that consumes two lists, and it actually needs its own chapter; see [Simultaneous Processing](#). On the other hand, the function definition clearly satisfies the purpose statement. The `andmap` expression checks that every item in `k` is a `member?` of `l`, which is what the purpose statement promises.

Sadly, `sorted-variant-of` fails to describe sorting functions properly. Consider this variant of a sorting function:

```

; [List-of Number] -> [List-of Number]
; produces a sorted version of l
(define (sort-cmp/worse l)
  (local ((define sorted-version (sort-cmp l <)))
    (cons (- (first sorted-version) 1) sorted-version)))

```

It is again easy to expose a flaw in this function with a `check-expect` test that it—like any correct sorting function—ought to pass but clearly fails:

```
(check-expect (sort-cmp/worse '(1 2 3)) '(1 2 3))
```

Amazingly, a `check-satisfied` test based on `\scheme{sorted-variant-of}` succeeds, however:

```
(check-satisfied (sort-cmp/worse '(1 2 3)) (sorted-variant-of '(1 2 3) <))
```

Indeed, such a test will succeed for any list of numbers, not just `'(1 2 3)`. The problem is that the predicate generator merely checks that all the items on the original list are members of the resulting list; it **fails** to check whether all the items on the resulting list are also members of the original list.

The easiest way to add this third check to `sorted-variant-of` is to add a third sub-expression to the `and` expression:

```

(define (sorted-variant-of.v2 k cmp)
  (lambda (l0)
    (and (sorted? cmp l0)
         (contains? l0 k)
         (contains? k l0))))

```

We choose to re-use `contains?` but with its arguments flipped; an alternative to make sure the two lists have the same length.

At this point, you may wonder why we are bothering with the development of such a predicate when we can rule out bad sorting functions with plain `check-expect` tests. The difference is that `check-expect` checks only that our sorting functions work on specific lists. With a predicate such as `sorted-variant-of.v2`, we can articulate the claim that a sorting function works for all possible inputs:

```
(define a-list (generate-a-list-of-random-numbers 500))

(define check-satisfied (sort-cmp a-list <) (sorted-variant-of.v2 a-list <))
```

Let's take a close look at these two lines. The first line generates a list of 500 numbers. Every time you ask DrRacket to evaluate this test, it is likely to generate a list never seen before. The second line is a test case that says sorting this generated list produces a list that (1) is sorted, (2) contains all the numbers on the generated list, and (3) contains nothing else. In other words, it is almost like saying that **for all** possible lists, `sort-cmp` produces outcomes that `sorted-variant-of.v2` blesses.

Computer scientists call `sorted-variant-of.v2` a *specification* of a sorting function. The idea that **all** lists of numbers pass the above test case is a **theorem** about the relationship between the specification of the sorting function and its implementation. If a programmer can prove this theorem with a mathematical argument, we say that the function is **correct** with respect to its specification. How to prove functions or programs correct is beyond the scope of this book, but a good computer science curriculum shows you in a follow-up course how to construct such proofs.

Exercise 298. Develop `found?`, a specification for the `find` function:

```
; X [List-of X] -> [Maybe [List-of X]]
; produces the first sublist of l that starts with x, #false otherwise
(define (find x l)
  (cond
    [(empty? l) #false]
    [else (if (equal? (first l) x) l (find x (rest l))))]))
```

Use `found?` to formulate a `check-satisfied` test for `find`. ■

Exercise 299. Develop `is-index?`, a specification for the `index` function:

```
; X [List-of X] -> [Maybe N]
; determine the (0-based) index of the first occurrence of x in l,
; #false otherwise
(define (index x l)
  (cond
    [(empty? l) #false]
    [else (if (equal? (first l) x)
              0
              (local ((define i (index x (rest l))))
                    (if (boolean? i) i (+ i 1))))]))
```

Use `is-index?` to formulate a `check-satisfied` test for `index`. ■

Exercise 300. Develop `n-inside-playground?`, a function that generates a predicate that ensures that the length of the given list is `k` and that all `Posns` in this list are within a `WIDTH` by `HEIGHT` rectangle:

```
(define WIDTH 300)
(define HEIGHT 300)

; N -> [List-of Posn]
; generate n random Posns in a WIDTH by HEIGHT rectangle
(define check-satisfied (random-posns 3) (n-inside-playground? 3))
```

```
(define (random-posns n)
  (build-list n (lambda (i) (make-posn (random WIDTH) (random HEIGHT)))))
```

Define `random-posns/bad` that satisfies `n-inside-playground?` and does not live up to the expectations implied by the above purpose statement. **Note** This specification is **incomplete**. Although the word “partial” might come to mind, computer scientists reserve the phrase “partial specification” for a different purpose. ■

19.5 Representing with lambda

Because functions are first-class values in ISL+, we may think of them as another form of data and use them for data representation. This section provides a taste of this idea; the next few chapters do not rely on it. Its title uses “abstracting” because people consider data representations that use functions as abstract.

As always, we start from a representative problem:

Sample Problem: Navy strategists represent fleets of ships as rectangles (the ships themselves) and circles (their weapons’ reach). The coverage of a fleet of ships is the combination of all these shapes. Design a data representation for rectangles, circles, and combinations of shapes. Then design a function that determines whether some point is within a shape.

This problem is also solvable with a self-referential data representation that says a shape is either a circle, a rectangle, or a combination of two shapes. See the next part for this design choice.

The problem comes with all kinds of concrete interpretations, which we leave out here. A slightly more complex version was the subject of a programming competition in the mid-1990s run by Yale University on behalf of the US Department of Defense.

One mathematical approach considers shapes as predicates on points. That is, a shape is a function that maps a Cartesian point to a Boolean value. Let’s translate these English words into a data definition:

```
; Shape is a function:
; [Posn -> Boolean]
; interpretation if s is a shape and p a Posn, (s p) produces
; #true if the given Posn is inside of s, #false otherwise
```

Its interpretation part is extensive because this data representation is so unusual. Such an unusual representation calls for an immediate exploration with examples. We delay this step for a moment, however, and instead define a function that checks whether a point is inside some shape:

```
; Shape Posn -> Boolean
(define (inside? s p)
  (s p))
```

Doing so is straightforward because of the given interpretation. It also turns out that it is simpler than creating examples and, surprisingly, the function is helpful for formulating data examples.

Stop! Explain how and why `inside?` works.

Now let us return to the problem of elements of `Shape`. Here is a simplistic element of the

class:

```
; Posn -> Boolean
(lambda (p) (and (= (posn-x p) 3) (= (posn-y p) 4)))
```

As required, it consumes a `Posn` `p`, and its body compares the coordinates of `p` to those of the point $(3,4)$, meaning this function represents a single point. While the data representation of a point as a `Shape` might seem silly, it suggests how we can define functions that create elements of `Shape`:

```
; Number Number -> Shape
(define (make-point x y)
  (lambda (p)
    (and (= (posn-x p) x) (= (posn-y p) y)))))

(define a-sample-shape (make-point 3 4))
```

Stop again! Convince yourself—perhaps by using DrRacket’s stepper—that the last line creates a data representation of $(3,4)$.

If we were to **design** such a function, we would formulate a purpose statement and provide some illustrative examples. For the purpose we could go with the obvious:

```
; creates a data representation for a point at (x,y)
```

or, more concisely,

```
; represents a point at (x,y)
```

For the examples we want to go with the interpretation of `Shape`. To illustrate, `(make-point 3 4)` is supposed to evaluate to a function that returns `#true` if, and only if, it is given `(make-posn 3 4)`. Using `inside?`, we can express this statement via tests:

```
(check-expect (inside? (make-point 3 4) (make-posn 3 4)) #true)
(check-expect (inside? (make-point 3 4) (make-posn 3 -4)) #false)
```

In short, to make a point representation, we define a constructor-like function that consumes the point’s two coordinates. Instead of a record, this function uses `lambda` to construct another function. The function that it creates consumes a `Posn` and determines whether its `x` and `y` fields are equal to the originally given coordinates.

Next we generalize this idea from simple points to shapes such as circles and rectangles. In your geometry courses, you learn that a circle is a collection of points that all have the same distance to the center of the circle—the radius. For points inside the circle, the distance is smaller or equal to the radius. Hence, a function that creates a `Shape` representation of a circle must consume three pieces: the two coordinates for its center and the radius:

```
; Number Number Number -> Shape
; creates a data representation for a circle of radius r
; located at (center-x, center-y)
(define (make-circle center-x center-y r)
  ...)
```

Like `make-point`, it produces a function via a `lambda`. The function that is returned determines whether some given `Posn` is inside the circle. Here are some examples, again formulated as tests:

```
(check-expect (inside? (make-circle 3 4 5) (make-posn 0 0)) #true)
(check-expect (inside? (make-circle 3 4 5) (make-posn 0 -1)) #false)
(check-expect (inside? (make-circle 3 4 5) (make-posn -1 3)) #true)
```

The origin, `(make-posn 0 0)`, is exactly five steps away from (3,4) the center of the circle; see [Defining Structure Types](#). Stop! Explain the second and third example.

Exercise 301. Use compass-and-pencil drawings to check the tests. ■

Mathematically, we say that a `Posn` p is inside a circle if the distance between p and the circle's center is smaller than the radius r . Let's wish for the right kind of helper function and write down what we have.

```
(define (make-circle center-x center-y r)
  ; [Posn -> Boolean]
  (lambda (p)
    (<= (distance-between center-x center-y p) r)))
```

The `distance-between` function is a straightforward exercise.

Exercise 302. Design the function `distance-between`. It consumes two numbers and a `Posn`: x , y , and p . The function computes the distance between the points (x, y) and p .

Domain Knowledge The distance between (x_0, y_0) and (x_1, y_1) is

$$\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

i.e., the distance of $(x_0 - y_0, x_1 - y_1)$ to the origin. ■

The data representation of a rectangle is expressed in a similar manner:

```
; Number Number Number Number -> Shape
; represent a width by height rectangle whose upper-left
; corner is located at (upper-left-x, upper-left-y)
(check-expect (inside? (make-rectangle 0 0 10 3) (make-posn 0 0)) #true)
(check-expect (inside? (make-rectangle 2 3 10 3) (make-posn 4 5)) #true)
(check-expect (inside? (make-rectangle 0 3 10 3) (make-posn -1 3)) #false)
(define (make-rectangle upper-left-x upper-left-y width height)
  (lambda (p)
    (and (<= upper-left-x (posn-x p) (+ upper-left-x width))
         (<= upper-left-y (posn-y p) (+ upper-left-y height))))))
```

Its constructor receives four numbers: the position of the upper, left corner, its width, and its height. The result is again a `lambda` expression, i.e., a function. As for circles, this function consumes a `Posn` and produces a `Boolean`, checking whether the x and y fields of the `Posn` are in the proper intervals.

At this point, we have only one task left, namely, the design of function that maps two `Shape` representations to their combination. The signature and the header are easy:

```
; Shape Shape -> Shape
; combines two shapes into one
(define (make-combination s1 s2)
  ; Posn -> Boolean
  (lambda (p)
    #false))
```

Indeed, even the default value is straightforward. We know that a shape is represented as a function from `Posn` to `Boolean`, so we write down a `lambda` that consumes some `Posn` and produces `#false`, meaning it says no point is in the combination.

So suppose we wish to combine the circle and the rectangle from above:

```
(define circle1 (make-circle 3 4 5))
(define rectangle1 (make-rectangle 0 3 10 3))
(define union1 (make-combination circle1 rectangle1))
```

We know that some points are inside and others are outside of this combination:

```
(check-expect (inside? union1 (make-posn 0 0)) #true)
(check-expect (inside? union1 (make-posn 0 -1)) #false)
(check-expect (inside? union1 (make-posn -1 3)) #true)
```

Since `(make-posn 0 0)` is inside both, there is no question that is inside the combination of the two. In a similar vein, `(make-posn 0 -1)` is in neither shape, and so it isn't in the combination. Finally, `(make-posn -1 3)` is in `circle1` but not in `rectangle1`. But the point must be in the combination of the two shapes because every point that is in one or the other shape is in their combination.

This analysis of examples implies the obvious revision of `make-combination`:

```
; Shape Shape -> Shape
(define (make-combination s1 s2)
  ; Posn -> Boolean
  (lambda (p)
    (or (inside? s1 p) (inside? s2 p))))
```

The `or` expression says that the result is `#true` if one of two expression produces `#true`: `(inside? s1 p)` or `(inside? s2 p)`. The first expression determines whether `p` is in `s1` and the second one whether `p` is in `s2`. And that is precisely a translation of our above explanation into ISL+ .

Exercise 303. Design `my-animate`. Recall that the `animate` function consumes the representation of a *stream* of images, one per natural number. Since streams are infinitely long, ordinary compound data cannot represent them. Instead, we use functions:

```
; An ImageStream is a function:
; [N -> Image]
; interpretation a stream s represents a time-series of images
```

Here is a data example:

```
; ImageStream
(define (create-rocket-scene height)


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

You may recognize this as one of the first pieces of code in [Prologue: How to Program](#).

The task of `(my-animate s n)` is to display the images `(s 0)`, `(s 1)`, and so on at a rate of 30 images per second up to `n` images total. Its result is the number of clock ticks passed since launched.

Note This case is an example where it is possible to write down examples/test cases easily but these examples/tests per se do not inform the design process of this **big-bang** function. Using functions as data representations calls for more design ideas than this book supplies. ■

Exercise 304. Design a data representation for finite and infinite sets so that you can represent the sets of all odd numbers, all even numbers, all numbers divisible by 10, etc.

Hint Mathematicians sometimes interpret sets as functions that consume a potential element ed and produce #**true** if the ed belongs to the set and #**false** if it doesn't.

Design the functions

1. `add-element`, which adds an element to a set;
2. `union`, which combines the elements of two sets; and
3. `intersect`, which collects all elements common to two sets;

Keep in mind the analogy between sets and shapes. ■

20 Summary

This third part of the book is about the role of abstraction in program design. Abstraction has two sides: creation and use. It is therefore natural if we summarize the chapter as two lessons:

1. **Repeated code patterns call for abstraction.** To abstract means to factor out the repeated pieces of code—the abstraction—and to parameterize over the differences. With the design of proper abstractions, programmers save themselves future work and headaches because mistakes, inefficiencies, and other problems are all in one place. One fix to the abstraction thus eliminates any specific problem once and for all. In contrast, the duplication of code means that a programmer must find all copies and fix all of them when a problem is found.
2. Most languages come with a large collection of abstractions. Some are contributions by the language design and implementation team; others are added by programmers who use the language. To enable **effective reuse of these abstractions**, their creators must supply the appropriate pieces of documentation—a **purpose statement, a signature, and good examples**—and programmers use them to apply abstractions.

All programming languages come with the means to build abstractions though some means are better than others. All programmers must get to know the means of abstractions and the abstractions that a language provides. A discerning programmer will learn to distinguish programming languages along these axes.

In addition to abstraction, this third part of the book also introduces the idea that

functions are values, and they can represent information as data.

While the idea is ancient for the Lisp family of programming languages (such as ISL+) and for specialists in programming language research, it has only recently gained acceptance in most modern mainstream languages—C#, C++, Java, JavaScript, Perl, Python.

v.6.3.0.2

Intermezzo: Scope and Abstraction

While the preceding part gets away with explaining `local` and `lambda` in an informal manner, the introduction of such abstraction mechanisms really requires additional terminology to facilitate such discussions. In particular, these discussions need words to delineate regions within programs and to refer to specific uses of variables.

This intermezzo starts with a section that defines the new terminology, most importantly, scope, binding variables, and bound variables. It immediately uses this new capability to introduce two abstraction mechanisms often found in programming languages: `for` loops and pattern matching. The former is an alternative to functions such as `map`, `build-list`, `andmap`, etc; the latter abstracts over the conditional in the functions of the first three parts of the book. Both require not only the definition of functions but the creation of entirely new language constructs, meaning they are not something programmers can usually design and add to their vocabulary.

While the *2htdp/abstraction* library is useful for the remaining parts of the book, the latter do not rely on any of its features. Instructors may wish to use the library anyway, though they should also explain then how the principles of design apply to languages without such features.

Scope

Consider the following two definitions:

```
(define (f x) (+ (* x x) 25))
(define (g x) (+ (f (+ x 1)) (f (- x 1))))
```

Clearly, the occurrences of `x` in `f` are completely unrelated to the occurrences of `x` in the definition of `g`. We could systematically replace the shaded occurrences with `y` and the function would still compute the exact same result. In short, the shaded occurrences of `x` have meaning only inside the definition of `f` and nowhere else.

At the same time, the first occurrence of `x` in `f` is different from the others. When we evaluate `(f n)`, the occurrence of `f` completely disappears while those of `x` are replaced with `n`. To distinguish these two kinds of variable occurrences, we call the `x` in the function header a *binding occurrence* and those in the function's body the *bound occurrences*. We also say that the binding occurrence of `x` binds all occurrences of `x` in the body of `f`. Indeed, people who study programming languages even have a name for the region where a binding occurrence works, namely, its *lexical scope*.

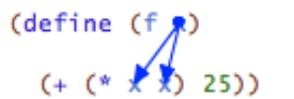
The definitions of `f` and `g` bind two more names: `f` and `g`. Their scope is called *top-level scope* because we think of scopes as nested (see below).

The word *free occurrence* applies to a variable without any binding occurrence. It is a name without definition, i.e., neither the language nor its libraries nor the program associates it with some value. For example, if you were to put the above program into a definitions area by itself and run it, entering `f`, `g`, and `h` at the prompt of the interactions would show that

the first two are defined and the last one is not:

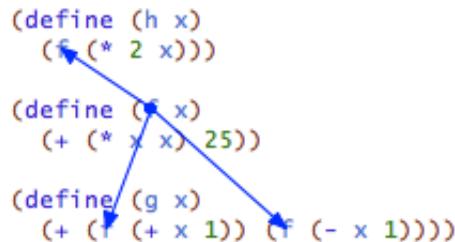
```
> f
#<procedure:f>
> g
#<procedure:g>
> h
h: this variable is not defined
```

The description of lexical scope suggests a pictorial representation of f's definition:



DrRacket's "Check Syntax" functionality draws these diagrams.

Here is an arrow diagram for top-level scope:



Note that the scope of f includes all definitions above and below its definition. The bullet over the first occurrence indicates that it is a binding occurrence. The arrows from the binding occurrence to the bound occurrences suggest the flow of values. When the value of a binding occurrence becomes known, the bound occurrences receive their values from there.

Along similar lines, these diagrams also explain how renaming works. If you wish to rename a function parameter, you search for all bound occurrences in scope and replace them. For example, renaming f's x to y in the program above means

```
(define (f x) (+ (* x x) 25))
(define (g x) (+ (f (+ x 1)) (f (- x 1))))
```

changes only two occurrences of x:

```
(define (f y) (+ (* y y) 25))
(define (g x) (+ (f (+ x 1)) (f (- x 1))))
```

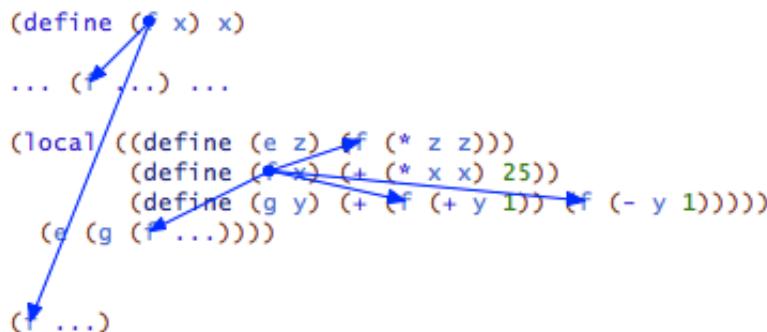
Exercise 305. Here is a simple ISL+ program:

```
(define (p1 x y)
  (+ (* x y)
     (+ (* 2 x)
        (+ (* 2 y) 22))))
(define (p2 x)
  (+ (* 55 x) (+ x 11)))
(define (p3 x)
  (+ (p1 x 0)
     (+ (p1 x 1) (p2 x))))
```

Draw arrows from p1's x parameter to all its bound occurrences. Draw arrows from p1 to

all bound occurrences of p1. Check the results with DrRacket “Check Syntax” functionality.

In contrast to top-level function definitions, the scope of the definitions in a `local` is limited. Specifically, the scope of local definitions is the `local` expression. Consider the definition of an auxiliary function `f` in a `local` expression. It binds all occurrences within the `local` expression but none that occur outside:



The two occurrences outside of `local` are not bound by the local definition of `f`. As always, the parameters of a function definition, local or not, are only bound in the function’s body.

Since the scope of a function name or a function parameter is a textual region, people also draw box diagrams to indicate scope. More precisely, for parameters a box is drawn around the body of a function:

```
(define (f x)
  (+ (* 2 x) 10))
```

In the case of a local definition, the box is drawn around the entire `local` expression:

```
(define (h z)
  (local ((define (f x) (+ (* x x) 55))
          (define (g y) (+ (f y) 10)))
        (f z)))
```

In this example, the box describes the scope of the definitions of `f` and `g`.

Using a box for a scope, we can also easily understand what it means to reuse the name of function inside a `local` expression:

```
(define (a-function y)
  (local ((define (f x y) (+ (* x y) (+ x y)))
          (define (g z)
            (local ((define (f x) (+ (* x x) 55))
                  (define (g y) (+ (f y) 10)))
                  (f z)))
            (define (h x) (f x (g x))))
          (h y)))
```

The inner box describes the scope of the inner definition of `f`; the outer box is the scope of the outer definition of `f`. Accordingly, all occurrences of `f` in the inner box refer to the inner `local`; all those in the outer box refer to the definition in the outer `local`. In other words, the scope of the outer definition of `f` has a *hole*, namely, the inner box.

Holes can also occur in the scope of a parameter definition. Here is an example:

```
(define (f x)
  (local ((define (g x) (+ x (* x 2)) ) )
    (g x)) )
```

In this function, the parameter x is used twice: for f and g . The scope of the latter is nested in the scope of the former and is thus a hole for the scope of the outer use of x .

In general, if the same name occurs more than once in a function, the boxes that describe the corresponding scopes never overlap. In some cases the boxes are nested within each other, which gives rise to holes. Still, the picture is always that of a hierarchy of smaller and smaller nested boxes.

Exercise 306. Here is a simple ISL+ function:

```
; [List-of X] -> [List-of X]
; creates a version of the given list that is sorted in descending order
(define (insertion-sort alon)
  (local ((define (sort alon)
            (cond
              [(empty? alon) '()]
              [else (add (first alon) (sort (rest alon))))]))
    (define (add an alon)
      (cond
        [(empty? alon) (list an)]
        [else
          (cond
            [(> an (first alon)) (cons an alon)]
            [else (cons (first alon) (add an (rest alon))))])))))
  (sort alon)))
```

Draw a box around the scope of each binding occurrence of `sort` and `alon`. Then draw arrows from each occurrence of `sort` to the appropriate binding occurrence. Now repeat the exercise for the following variant:

```
(define (sort alon)
  (local ((define (sort alon)
            (cond
              [(empty? alon) '()]
              [else (add (first alon) (sort (rest alon))))]))
    (define (add an alon)
      (cond
        [(empty? alon) (list an)]
        [else
          (cond
            [(> an (first alon)) (cons an alon)]
            [else (cons (first alon) (add an (rest alon))))])))))
  (sort alon)))
```

Do the two functions differ other than in name? ■

Exercise 307. Recall that each occurrence of a variable receives its value from the corresponding binding occurrence. Consider the following definition:

```
(define x (cons 1 x))
```

Where is the shaded occurrence of x bound? Since the definition is a constant definition

and not a function definition, we need to evaluate the right-hand side if we wish to work with this function. What should be the value of the right-hand side according to our rules?

As discussed in [Functions from lambda](#), a `lambda` expression is just a short-hand for a `local` expression, i.e.,

```
(lambda (x-1 ... x-n) exp)
```

is short for

```
(local ((define (a-new-name x-1 ... x-n) exp))
       a-new-name) ; if a-new-name does not occur in exp.
```

The short-hand explanation suggests that

```
(lambda (x-1 ... x-n) exp)
```

introduces $x-1, \dots, x-n$ as binding occurrences and that the scope of parameters is `exp`, e.g.,

```
(define f
  (lambda (x)
    (+ (* x x) 25)))
```

Of course, if `exp` contains further binding constructs (say, a nested `local` expression), then the scope of the variables may have a hole.

Exercise 308. Draw arrows from the shaded occurrences of `x` to their binding occurrences in each of the following three `lambda` expressions:

1.

```
(lambda (x y)
      (+ x (* x y)))
```

2.

```
(lambda (x y)
      (+ x
          (local ((define x (* y y)))
            (+ (* 3 x)
                (/ 1 x)))))
```

3.

```
(lambda (x y)
      (+ x
          ((lambda (x)
              (+ (* 3 x)
                  (/ 1 x)))
            (* y y))))
```

Also draw a box for the scope of each shaded `x` and holes in the scope as necessary.

For Loops

Even though it never mentions the word, [Abstraction](#) introduces loops. Abstractly, a *loop* traverses compound data, processing one piece at a time. In the process, loops also synthesize data. For example, `map` traverses a list, applies a function to each item, and collects the results in a list. Similarly, `build-list` enumerates the sequence of predecessors of a natural number (from `0` to `(- n 1)`), maps each of these to some value,

and also gathers the results in a list.

The loops of ISL+ differ from those in conventional languages in two ways. First, a conventional loop does not directly create new data; in contrast, abstractions such as `map` and `build-list` are all about computing new data from traversals. Second, conventional languages often provide only a fixed number of loops; an ISL+ programmer defines new loops as needed. Put differently, conventional languages view loops as syntactic constructs akin to `local` or `cond`, and their introduction requires a detailed explanation of their vocabulary, grammar, scope, and meaning.

Loops as syntactic constructs have two advantages over functional loops. On one hand, their shape tends to signal intentions more directly than a composition of functions. On the other hand, language implementations typically translate syntactic loops into faster commands for computers than functional loops. It is therefore common that even functional programming languages—with all their emphasis on functions and function compositions—provide syntactic loops.

Use the `2htdp/abstraction` library for this section and the next one.

In this section, we introduce a few of Racket’s so-called `for` loops. The goal is to illustrate how to think about conventional loops as linguistic constructs and to indicate how programs built with abstractions may use loops instead. Figure 79 spells out the grammar of our selected `for` loops as an extension of BSL’s grammar from Intermezzo: BSL. Every loop is an expression and, like all compound constructs, is marked with a keyword. The latter is followed by a parenthesized sequence of so-called *comprehension clauses* and a single expression. The clauses introduce so-called *loop variables*, and the expression at the end is the *loop body*.

```

expr = ...
| (for/list (clause clause ...) expr)
| (for*/list (clause clause ...) expr)
| (for/and (clause clause ...) expr)
| (for*/and (clause clause ...) expr)
| (for/or (clause clause ...) expr)
| (for*/or (clause clause ...) expr)
| (for/sum (clause clause ...) expr)
| (for*/sum (clause clause ...) expr)
| (for/product (clause clause ...) expr)
| (for*/product (clause clause ...) expr)
| (for/string (clause clause ...) expr)
| (for*/string (clause clause ...) expr)

clause = (variable expr)

```

Figure 79: ISL+ extended with `for` loops

Even a cursory look at the grammar shows that the dozen looping constructs come in six pairs: a `for` and `for*` variant for each of `list`, `and`, `or`, `sum`, `product`, and `string`. All `for` loops bind the variables of their clauses in the body; the `for*` variants also bind variables in the subsequent clauses. The following two, near-

Racket comes with many more loops than this, and its loops come with more functionality than those presented here.

identical code snippets illustrate the difference between these two scoping rules:



The syntactic difference is that the left one uses `for/list` and the right one `for*/list`. In terms of scope, the two strongly differ as the arrows indicate. While both pieces introduce two loop variables—`width` and `height`—the left one uses an externally defined variable for `height`'s initial value and the right one uses the first loop variable.

Semantically, a `for/list` expression evaluates the expressions in its clauses to generate sequences of values. If an expression in a clause evaluates to a

- list, its items make up the sequence values;
- natural number n , the sequence of values consists of the numbers $0, 1, \dots, (- n 1)$;
- string, its one-character strings are the sequence items.

Finally, `for/list` evaluates the loop body with the loop variables successively bound to the values of the sequences generated by the clause expressions, and it collects the values of its body into a list. Note that the evaluation of a `for/list` expression stops when the shortest sequence is exhausted.

Terminology Each evaluation of a loop body is called an *iteration*. Similarly, a loop is said to *iterate* over the values of its loop variables.

Based on this explanation, the following, first example generates the list from 0 to 9 :

```

> (for/list ((i 10)
             i)
  '(0 1 2 3 4 5 6 7 8 9)
> (build-list 10 (lambda (i) i))
'(0 1 2 3 4 5 6 7 8 9)
  
```

For comparison, the second expression shows how to create the same list with `build-list` and `lambda`. Here is a second example:

```

> (for/list ((i 2) (j '(a b)))
             (list i j))
'((0 a) (1 b))
> (local ((define i-s (build-list 2 (lambda (i) i)))
             (define j-s '(a b)))
      (map list i-s j-s))
'((0 a) (1 b))
  
```

Again, the second expression generates the same list using ISL+'s abstractions.

Let's finish the introduction of `for/list` with a practical example:

Sample Problem: Design `enumerate`. The function consumes a list and produces

a list of the same items paired with their relative index.

Stop! Design this function systematically using ISL+'s abstractions.

With `for/list`, this problem has a straightforward solution:

```
; [List-of X] -> [List-of [List N X]]
; pair each item in l with its index
(check-expect (enumerate '(a b c)) '((1 a) (2 b) (3 c)))
(define (enumerate l)
  (for/list ((item l) (ith (length l)))
    (list (+ ith 1) item)))
```

The function's body uses `for/list` to iterate over the given list and a list of numbers from `0` to `(- (length l) 1)`; the loop body combines the index with the list item.

In terms of semantics, `for*/list` differs from `for/list` in a subtle way. Unlike the latter, the former iterates over the sequences of values in a nested fashion. That is, a `for*/list` expression basically unfolds into a nest of loops:

```
> (for*/list ((i 2) (j '(a b)))
  (list i j))
'((0 a) (0 b) (1 a) (1 b))
> (for/list ((i 2))
  (for/list ((j '(a b)))
    (list i j)))
'(((0 a) (0 b)) ((1 a) (1 b)))
```

Nested loops, however, produce nested results; to collect those results into a single list, `for*/list` concatenates the nested lists using `foldl` and `append`.

Here is a problem that illustrates this use of `for*/list` in context:

Sample Problem: Design `cross`. The function consumes two lists, `l1` and `l2`, and produces pairs of all items from `l1` and `l2`.

Stop! Take a moment to design the function using existing abstractions.

If you designed `cross` properly, you worked through a table such as this one:

	'a	'b	'c
1	(list 'a 1)	(list 'b 1)	(list 'c 1)
2	(list 'a 2)	(list 'b 2)	(list 'c 2)

The first row displays `l1` as given, while the left-most column shows `l2`. Each cell in the table corresponds to one of the pairs that `cross` must generate.

Since the purpose of `for*/list` is an enumeration of all such pairs, defining `cross` via `for*/list` is straightforward:

```
; [List-of X] [List-of Y] -> [List-of [List X Y]]
; generate all possible pairs of items from l1 and l2
(check-satisfied (cross '(a b c) '(1 2)) (lambda (c) (= (length c) 6)))
(define (cross l1 l2)
  (for*/list ([item1 l1][item2 l2])
    (list item1 item2)))
```

We use `check-satisfied` instead of `check-expect` because we do not wish to predict

the exact order in which `for*/list` generates the pairs.

Note Figure 80 shows another in-context use of `for*/list`. It displays a compact solution of the extended design problem covered by [Word Games, the Heart of the Problem](#), namely,

We thank Mr. Mark Engelberg for suggesting this exhibition of expressive power.

given a word, create all possible re-arrangements of the letters in a list.

The extended exercise is a direct guide to the design of the main function and two auxiliaries, where the design of the latter requires the creation of two more helper functions. In contrast, figure 80 uses the combined power of `for*/list` and an unusual form of recursion to define the same program as a single, five-line function definition. The code is listed here merely to exhibit the power of these abstractions; for the yet-unknown design method, see [Generative Recursion](#) and especially [exercise 479](#). **End**

```
; [List-of X] -> [List-of [List-of X]]
; creates a list of all rearrangements of the items in w
(define (arrangements w)
  (cond
    [(empty? w) '()]
    [else (for*/list ([item w]
                      [arrangement-without-item
                       (arrangements (remove item w))])
                  (cons item arrangement-without-item)))))

; test:
; [List-of X] -> Boolean
(define (all-words-from-rat? w)
  (and (member? (explode "rat") w)
       (member? (explode "art") w)
       (member? (explode "tar") w)))

(check-satisfied (arrangements '("r" "a" "t")) all-words-from-rat?)
```

Figure 80: A compact definition of arrangements with `for*/list`

At this point, the difference between `for/list` and `for*/list` is pretty clear, but here is a pair of interactions that turns the difference in scope into different behavior:

```
> (define width 2)
> (for/list ([width 3][height width])
  (list width height))
'((0 0) (1 1))
> (for*/list ([width 3][height width])
  (list width height))
'((1 0) (2 0) (2 1))
```

To understand the first one, remember that `for/list` traverses the two sequences in parallel and stops when the shorter one is exhausted. Here, the two sequences are

width	=	0	1	2
height	=	0		1
body	=	(list 0 0)	(list 1 1)	

The first two rows show the values of the two loop variables, which change in tandem. The

last row shows the result of each iteration, which explains the first result. Now contrast this situation with `for*/list`, which proceeds in a nested fashion:

```
width   =  0      1      2
height  =          0      0, 1
body    =  (list 1 0)  (list 2 0), (list 2 1)
```

While the first row is like the one for `for/list`, the second one now displays sequences of numbers in its cells. The implicit nesting of `for*/list` means that each iteration with a specific value of `width` re-initializes `height` and thus creates a distinct **sequence** of `height` values. This explains why the first cell of `height` values is empty; after all, there are no natural numbers between 0 (inclusive) and 0 (exclusive). Finally, each nested `for` loop yields a sequences of pairs, which, as mentioned above, are collected into a single list of pairs.

It is also clear now that `.../list` signals that the loop creates a list by collecting the results in the order in which they are generated. By analogy, `for` and `for*` loops for

- `.../and` apply an operation like `and` to all of the generated values:

```
> (for-and ((i 10)) (> (- 9 i) 0))
#false
> (for-and ((i 10)) (if (>= i 0) i #false))
9
```

For pragmatic reasons, the loops return the last generated value or `#false` if any of the results are `#false`.

- `.../or` apply an operation like `or` to all of the generated values:

```
> (for-or ((i 10)) (if (= (- 9 i) 0) i #false))
9
> (for-or ((i 10)) (if (< i 0) i #false))
#false
```

These loops return the first value that is not `#false` unless all the values are `#false`.

- `.../sum` add up the numbers that the iterations generate:

```
> (for/sum ((c "abc")) (string->int c))
294
```

- `.../product` multiply the numbers that the iterations generate

```
> (for/product ((c "abc")) (+ (string->int c) 1))
970200
```

- `.../string` create `Strings` from the `1Strings` that the iterations generate:

```
> (define a (string->int "a"))
> (for/string ((j 10)) (int->string (+ a j)))
"abcdefghijklmnopqrstuvwxyz"
```

Racket also comes with `for/fold`, which combines the results of the loop bodies via a program-defined operation; this loop is not included in the library.

Stop! It is an instructive exercise to re-formulate all of the above examples using the existing abstractions in ISL+. Doing so also indicates how to design functions with `for`

loops instead of abstract functions. **Hint** Design `and-map` and `or-map`, which work like `andmap` and `ormap`, respectively, but return the appropriate non-`#false` values.

```
; N -> sequence?
; construct an infinite sequence of natural numbers starting at n
(define (in-naturals n) ...)

; N N N -> sequence?
; construct the finite sequence of natural numbers starting with
;   start
;   (+ start step)
;   (+ start step step)
;   ...
;   until the number exceeds end
(define (in-range start end step) ...)
```

Figure 81: Constructing sequences of natural numbers

Looping over numbers isn't always a matter of enumerating `0` through `(- n 1)`. Often programs need to step through non-sequential sequences of numbers; other times, an unlimited supply of numbers is needed. To accommodate this form of programming, Racket comes with functions that generate sequences, and [figure 81](#) lists two that are provided in the abstraction library for ISL+.

With the first one, we can simplify the `enumerate` function a bit:

```
(define (enumerate.v2 l)
  (for/list ((item l) (ith (in-naturals 1)))
    (list ith item)))
```

Here `in-naturals` is used to generate the infinite sequence of natural numbers starting at `1`; the `for` loop stops when `l` is exhausted. **Note** This version of `enumerate` is also a bit faster than the original because it traverses `l` only once.

With the second one, it is, for example, possible to step through the even numbers among the first `n`:

```
; N -> Number
; add the even numbers between 0 and n (exclusive)
(check-expect (sum-evens 2) 0)
(check-expect (sum-evens 4) 2)
(define (sum-evens n)
  (for/sum ([i (in-range 0 n 2)]) i))
```

Although this use may appear trivial, many problems originating in mathematics call for just such loops, which is precisely why concepts such as `in-range` are found in many programming languages.

Exercise 309. Use loops to define `convert-euro`, a function that converts a list of US\$ amounts into a list of € amounts based on an exchange rate of €1.08 per US\$. Compare with [exercise 271](#). ■

Exercise 310. Use loops to define a function that

1. creates the list `0 ... (n - 1)` for any natural number `n`;

2. creates the list $1 \dots n$ for any natural number n ;
3. creates the list $(\text{list } 1 \ 1/10 \ 1/100 \ \dots)$ of n numbers for any natural number n ;
4. creates the list of the first n even numbers;
5. creates a list of lists of 0 and 1 in a diagonal arrangement, e.g.,

```
(equal? (diagonal 3)
  (list
    (list 1 0 0)
    (list 0 1 0)
    (list 0 0 1)))
```

Finally, use loops to define `tabulate` from [exercise 254](#). See [exercise 274](#). ■

Exercise 311. Define `find-name`. The function consumes a name and a list of names. It retrieves the first name on the latter that is equal to, or an extension of, the former.

Define a function that ensures that no name on some list of names exceeds some given width. Compare with [exercise 275](#). ■

Pattern Matching

When we design a function for a data definition with six clauses, we use a six-pronged `cond` expression. When we formulate one of the `cond` clauses, we use a predicate to determine to determine whether this clause should process the given value and, if so, selectors to deconstruct any compound values. The first three parts of this book explain this idea over and over again, and many of its code snippets exhibit just this pattern.

The *2htdp/abstraction* library also supports the definition of algebraic data types with `define-type` and their deconstruction using `type-case`. The latter is like `match` but also ensures that the conditional has the correct number and kinds of clauses.

Repeated patterns call for abstraction. While [Abstraction](#) explains how programmers can create some of these abstractions, the predicate-selector pattern can be addressed only by a language designer. In particular the designers of functional programming languages have recognized the need for abstracting the repetitive uses of predicates and selectors. These languages therefore provide *pattern matching* as a linguistic construct that combines and simplifies these `cond` clauses.

This section presents Racket's pattern matcher as provided by the *2htdp/abstraction* library. [Figure 82](#) displays the grammatical extension for `match`. Clearly, `match` is a syntactically complex construct. While its basic outline resembles that of `cond`, the conditions are replaced by patterns, which come with their own vocabulary and grammatical rule.

```
expr = ...
| (match expr [pattern expr] ...)

pattern = variable
| literal-constant
| (cons pattern pattern)
```

```

| (structure-name pattern ...)
| (? predicate-name)

```

Figure 82: ISL+ match expressions

Roughly speaking,

```

(match expr
  [pattern1 expr1]
  [pattern2 expr2]
  ...
)
```

proceeds like a `cond` expression in that it evaluates `expr` and sequentially tries to match its result with `pattern1`, `pattern2`, ... until it succeeds with `patterni`. At that point, it determines the value of `expri`, which is also the result of the entire `match` expression.

The key difference is that `match`, unlike `cond`, introduces a new scope, which is best illustrated with a screen shot from DrRacket:

```

(define (sum-items a-lon)
  (match a-lon
    [(cons fst '()) fst]
    [(cons fst rst)
     (+ fst (sum-items rst))]))

```

As the image shows, each pattern clause of this function binds variables. Furthermore, the scope of a variable is the body of the clause, so even if two patterns introduce the same variable binding—as it is the case in the above code snippet—their bindings cannot interfere with each other.

Syntactically, a pattern resembles a piece of nested, structural data whose leafs are literal constants, variables, or predicate patterns of the shape `(? predicate-name)`. In the latter, `predicate-name` must refer to a predicate function in scope, that is, a function that consumes one value and produces a `Boolean`.

Semantically, a pattern is `matched` to a value `v`. If the pattern is

- a `literal-constant`, it matches only that literal constant

```

> (match 4
  ['four 1]
  ['"four" 2]
  [#true 3]
  [4 "hello world"])
"hello world"

```

- a `variable`, it matches any value, and it is associated with this value during the evaluation of the body of the corresponding `match` clause

```

> (match 2
  [3 "one"]
  [x (+ x 3)])
5

```

Since `2` does not equal the first pattern, which is the literal constant `3`, `match` matches `2` with the second pattern, which is a plain variable and thus matches any value. Hence,

`match` picks the second clause and evaluates its body, with `x` standing for `2`.

- (`cons pattern1 pattern2`), it matches only an instance of `cons`, assuming its first field matches `pattern1` and its rest field matches `pattern2`

```
> (match (cons 1 '()))
  [(cons 1 tail) tail]
  [(cons head tail) head])
'( )
> (match (cons 2 '()))
  [(cons 1 tail) tail]
  [(cons head tail) head])
2
```

These interactions show how `match` first deconstructs `cons` and then uses literal constants and variables for the leafs of the given list.

- (`structure-name pattern1 ... patternn`), it matches only a `structure-name` structure, assuming its field values match `pattern1, ..., patternn`

```
> (define p (make-posn 3 4))
> (match p
  [(posn x y) (sqrt (+ (sqr x) (sqr y)))])
5
```

Obviously, matching an instance of `posn` with a pattern is just like matching a `cons` pattern. Note though, how the pattern uses `posn` for the pattern, not the name of the constructor.

Matching also works for programmer-introduced structure type definitions:

```
> (define-struct phone [area switch four])
> (match (make-phone 713 664 9993)
  [(phone x y z) (+ x y z)])
11370
```

Again, the pattern uses the name of the structure, `phone`, not that of the constructor.

Finally, matching also works across several layers of data constructions:

```
> (match (cons (make-phone 713 664 9993) '())
  [(cons (phone area-code 664 9993) tail) area-code])
713
```

This `match` expression extracts the area code from a phone number in a list if the switch code is `664` and the last four digits are `9993`.

- (`? predicate-name`), it matches only when `(predicate-name v)` produces `#true`

```
> (match (cons 1 '())
  [(cons (? symbol?) tail) tail]
  [(cons head tail) head])
1
```

This expression produces `1`, the result of the second clause, because `1` is not a symbol.

Stop! Experiment with `match` before you read on.

At this point, it is time to demonstrate the usefulness of `match` in context:

Sample Problem: Design the function `last-item`, which retrieves the last item on a non-empty list. Recall that non-empty lists are defined as follows:

```
; A [Non-empty-list X] is one of:  
; - (cons X '())  
; - (cons X [Non-empty-list X])
```

Stop! [Arbitrarily Large Data](#) deals with this problem. Look up the solution.

With `match`, a designer can eliminate three selectors and two predicates from the solution using `cond`:

```
; [Non-empty-list X] -> X  
; retrieve the last item of ne-l  
(check-expect (last-item '(a b c)) 'c)  
(check-error (last-item '()))  
(define (last-item ne-l)  
  (match ne-l  
    [(cons lst '()) lst]  
    [(cons fst rst) (last-item rst)])))
```

Instead of predicates and selectors, this solution uses patterns that are just like those found in the data definition. For each self-reference and occurrence of the set parameter in the data definition, the patterns use program-level variables. The bodies of the `match` clauses no longer extract the relevant parts from the list with selectors but simply refer to these names. As before, the function recurs on the `rest` field of the given `cons` because the data definition refers to itself in this position. In the base case, the answer is `lst`, the variable that stands for the last item on the list.

Let's take a look at a second problem from [Arbitrarily Large Data](#):

Sample Problem: Design the function `depth`, which measures the number of layers surrounding Russian doll. Recall the corresponding data definition:

```
(define-struct layer [color doll])  
; An RD (russian doll) is one of:  
; - "wooden doll"  
; - (make-layer String RD)
```

Stop! Look up the solution and try to modify it to use `match`.

Here is a definition of `depth` using `match`:

```
; RD -> N  
; how many dolls are a part of an-rd  
(check-expect (depth (make-layer "red" "wooden doll")) 1)  
(define (depth a-doll)  
  (match a-doll  
    ["wooden doll" 0]  
    [(layer c inside) (+ (depth inside) 1)]))
```

While the pattern in the function's first `match` clause looks for the literal string `"wooden doll"`, the second one matches any `layer` structure, associating `c` with the value in the `color` field and `inside` with the value in the `doll` field. In short, `match` again simplifies the conditional expression needed by eliminating predicates and selectors.

The final problem is an excerpt from a generalized version of our UFO game:

Sample Problem: Use `match` to design the function `move-right`. The latter consumes a list of `Posns`, which represent the positions of objects on a canvas, plus a number. It adds the latter to each x-coordinate, which represents an rightward movement of these objects.

Stop! Look up the function `tock` in [figure 48](#), which solves a variant of this sample problem.

Our solution to the problem combines the abstract `map` function introduced in the preceding part of the book with `match`:

```
; [List-of Posn] -> [List-of Posn]
; moves each object right by delta-x pixels
(check-expect (move-right (list (make-posn 1 1) (make-posn 10 14)) 3)
                (list (make-posn 4 1) (make-posn 13 14)))
(define (move-right lop delta-x)
  (map (lambda (p)
         (match p
           [(posn x y) (make-posn (+ x delta-x) y)])))
    lop))
```

You may wish to define the version that uses `cond` and the selectors and the one that uses `match` only—both without using `map`. We leave it to you to judge which version is the most readable one.

Exercise 312. Use `match` to design the function `replace`, which substitutes the area code [713](#) with [281](#) in a list of phone records. For a structure type definition of phone records, see above. Formulate a suitable data definition first. If you are stuck, look up your solution for [exercise 172](#). ■

Exercise 313. [Figure 54](#) displays a function that determines the number of words per nested list in a list of list of strings. Using the notation from [Abstraction](#), the function's signature, purpose and header can be formulated like this:

```
; [List-of [List-of String]] -> [List-of Number]
; determines the number of words on each line
(define (words-on-line lls)
  '())
```

Define the function using `match`. ■

v.6.3.0.2

IV Intertwined Data

You might think that the data definitions for lists and natural numbers are quite unusual. These data definitions refer to themselves, and in all likelihood, they are the first such definitions you have ever encountered. As it turns out, many classes of data require even more complex data definitions than these two. Common generalizations involve many self-references in one data definition or a bunch of data definitions that refer to each other. These forms of data have become ubiquitous and it is therefore critical for a programmer to learn to cope with **any** collection of data definitions. And that's what the design recipe is all about.

This part starts with a generalization of the design recipe so that it works for all forms of structural data definitions. Next, it introduces the concept of iterative refinement because complex data definitions are not developed in one fell swoop but in several stages. Indeed, the use of iterative refinement is one of the reasons why all programmers are little scientists and why our discipline uses the word “science” in its American name. Two last chapters illustrate these ideas: one explains how to design an interpreter for BSL and another is about processing XML, a data exchange language for the Web. The last chapter expands the design recipe one more time, re-working it for functions that process two complex arguments at the same time.

22 The Poetry of S-expressions

Programming resembles poetry in many ways. For example, programmers—like poets—practice their skill on seemingly pointless ideas. In this spirit, this chapter introduces increasingly complex forms of data without a true purpose. Even when we provide a motivational background, the chosen kinds of data are “purist,” and it is unlikely that you will ever encounter these particular kinds of data again.

Nevertheless, this chapter shows the full power of the design recipe and introduces you to the kinds of data that real-world programs cope with. To connect this material with what you will encounter in your life as a programmer, we label each section with appropriate names: trees, forests, XML. The last one is a bit misleading, because it is really about S-expressions; the connection between S-expressions and XML is clarified in [The Commerce of XML](#), which in contrast to this chapter, comes much closer to real-world uses of complex forms of data.

22.1 Trees

All of us have a family tree. One way to draw a family tree is to add a node every time a child is born. From the node, we can draw connections to the father and mother of the child. For those people in the tree whose parents are unknown, there is no connection to draw. The result is a so-called ancestor family tree because, given any person in the tree, we can find all of the person's known ancestors.

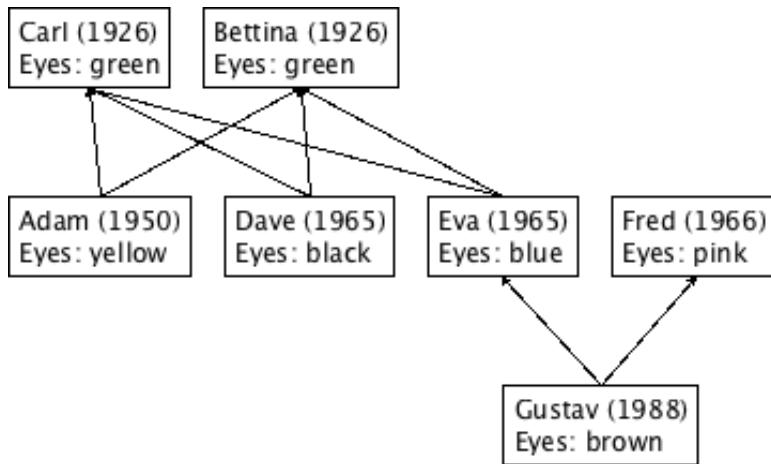


Figure 83: A family tree

[Figure 83](#) displays a three-tier family tree. Gustav is the child of Eva and Fred, while Eva is the child of Carl and Bettina. In addition to people's names and family relationships, the tree also records years of birth and eye colors. Based on this sketch, you can easily imagine a family tree reaching back many generations and one that records other kinds of information.

Once a family tree is large, it makes sense to represent it as data and to design programs that process this kind of data. Given that a node in a family tree combines five pieces of information—the father, the mother, the name, the birth date, and the eye color—we should define a structure type:

```
(define-struct child [father mother name date eyes])
```

The structure type definition calls a data definition:

```
; A Child is a structure:  
; (make-child Child Child String N String)
```

While this data definition looks straightforward, it is also useless. It refers to itself but, because it doesn't have any clauses, there is no way to create a proper instance `Child`. Roughly, we would have to write

```
(make-child (make-child (make-child ...) ...)) ...)
```

without end. To avoid such pointless data definitions, we demand that a self-referential data definition has several clauses and that at least one of them does not refer back to the data definition.

Let's postpone the data definition for a moment; let's experiment instead. Suppose we are about to add a child to an existing family tree and that we already have representations for the parents. In that case, we can simply construct a new `child` structure. For example, to represent Adam in a program that already represents Carl and Bettina, it suffices to add the following `child` structure:

```
(define Adam (make-child Carl Bettina "Adam" 1950 "yellow"))
```

assuming `Carl` and `Bettina` stand for representations of Adam's parents.

One problem is that we don't always know a person's parents, and its solution points to the correct data definitions. In the family tree of [figure 83](#), we don't know Bettina's parents. Yet, even if we don't know a person's father or mother, we must still use some piece of data for the parent fields in a `child` representation of Bettina. Whatever data we choose, it

must signal an absence of information. On the one hand, we could use `#false`, `"none"`, or `'()` from the pool of existing values. On the other hand, we should really say that it is missing information about a family tree, and we can achieve this objective best with the introduction of a structure type with an appropriate name:

```
(define-struct no-parent [])
```

Now, to construct a `child` structure for Bettina, we say

```
(make-child (make-no-parent) (make-no-parent) "Bettina" 1926 "green")
```

Of course, if only one piece of information is missing, we fill just that field with this special value.

The experiment suggests two insights. First, we are **not** looking for a data definition that describes how to generate instances of `child` structures but for a data definition that describes how to represent family trees. Second, the data definition consists of two clauses, one for the nodes describing unknown family trees and another one for known family trees:

```
(define-struct no-parent [])
(define-struct child [father mother name date eyes])
; A FT (family tree) is one of:
; - (make-no-parent)
; - (make-child FT FT String N String)
```

Since the “no parent” tree is going to show up a lot in our programs, we define `MTFT` as a short-hand and revise the data definition a bit:

```
(define MTFT (make-no-parent))
; A FT (family tree) is one of:
; - MTFT
; - (make-child FT FT String N String)
```

The use of a constant in the data definition should not bother you. You can replace a name by its value.

Following the design recipe from [Designing with Self-Referential Data Definitions](#), we use the data definition to create examples of family trees. Specifically, we translate the family tree in [figure 83](#) into our data representation. The information for Carl is easy to translate into data:

```
(make-child MTFT MTFT "Carl" 1926 "green")
```

Bettina and Fred are represented with similar instances of `child`. The case for Adam calls for nested children, one for Carl and one for Bettina:

```
(make-child (make-child MTFT MTFT "Carl" 1926 "green"))
          (make-child MTFT MTFT "Bettina" 1926 "green")
          "Adam"
          1950
          "hazel")
```

Since the records for Carl and Bettina are also needed to construct the records for Dave and Eva, it is better to introduce definitions that name specific instances of `child` and to use the variable names elsewhere. [Figure 84](#) illustrates this approach for the complete data representation of the family tree from [figure 83](#). Take a close look; the tree serves as our running example for the following design exercise.

```

; Oldest Generation:
(define Carl (make-child MTFT MTFT "Carl" 1926 "green"))
(define Bettina (make-child MTFT MTFT "Bettina" 1926 "green"))

; Middle Generation:
(define Adam (make-child Carl Bettina "Adam" 1950 "hazel"))
(define Dave (make-child Carl Bettina "Dave" 1955 "black"))
(define Eva (make-child Carl Bettina "Eva" 1965 "blue"))
(define Fred (make-child MTFT MTFT "Fred" 1966 "pink"))

; Youngest Generation:
(define Gustav (make-child Fred Eva "Gustav" 1988 "brown"))

```

Figure 84: A data representation of the sample family tree

Instead of designing a concrete function on family trees, let us first look at the generic organization of such a function. That is, let us work through the design recipe as much as possible without having a concrete task in mind. We start with the header material, i.e., step 2 of the recipe:

```

; FT -> ???
; ...
(define (fun-for-FT a-ftree) ...)

```

Even though we aren't stating the purpose of the function, we do know that it consumes a family tree and that this form of data is the main input. The “???” in the signature says that we don't know what kind of data the function produces; the “...” remind us that we don't know its purpose.

The lack of purpose implies that we cannot make up functional examples. Nevertheless, we can exploit the organization of the data definition for FT to design a template. Since it consists of two clauses, the template must consist of a `cond` expression with two clauses:

```

; FT -> ???
(define (fun-for-FT a-ftree)
  (cond
    [(no-parent? a-ftree) ...]
    [else ; (child? a-ftree)
     ...]))

```

The first deals with `MTFT`, the second with `child` structures.

In case the argument to `fun-for-FT` satisfies `no-parent?`, the structure contains no additional data, so the first clause is complete. For the second clause, the input contains five pieces of data, which we indicate with five selectors in the template:

```

; FT -> ???
(define (fun-for-FT a-ftree)
  (cond
    [(no-parent? a-ftree) ...]
    [else
     (... (child-father a-ftree) ...
          ... (child-mother a-ftree) ...
          ... (child-name a-ftree) ...
          ... (child-date a-ftree) ...
          ... (child-eyes a-ftree) ...)]))

```

The last addition to templates concerns self-references. If a data definition refers to itself, the function is likely to recur and templates indicate so with suggestive natural recursions. Unlike lists, the data definition for family trees requires two self-references and the template therefore needs two recursions:

```
; FT -> ???
(define (fun-for-FT a-ftree)
  (cond
    [(no-parent? a-ftree) ...]
    [else
      (... (fun-for-FT (child-father a-ftree)) ...)
      (... (fun-for-FT (child-mother a-ftree)) ...)
      (... (child-name a-ftree) ...)
      (... (child-date a-ftree) ...)
      (... (child-eyes a-ftree) ...)])))
```

Specifically, `fun-for-FT` is applied to the data representation for fathers and mothers in the second `cond` clause, because the second clause of the data definition contains corresponding self-references.

Let us now turn to a concrete function, `blue-eyed-child?`. Its purpose is to determine whether any `child` structure in a given family tree has blue eyes. You may copy, paste, and rename `fun-for-FT` to get its template; we replace “`???`” with `Boolean` and add a purpose statement;

```
; FT -> Boolean
; does a-ftree contain a child
; structure with "blue" in the eyes field
(define (blue-eyed-child? a-ftree)
  (cond
    [(no-parent? a-ftree) ...]
    [else
      (... (child-name a-ftree) ...)
      (... (child-date a-ftree) ...)
      (... (child-eyes a-ftree) ...)
      (... (blue-eyed-child? (child-father a-ftree)) ...)
      (... (blue-eyed-child? (child-mother a-ftree)) ...)])))
```

When you work in this fashion, you need to replace the template’s generic name with a specific one.

Checking with our recipe, we realize that we need to backtrack and develop some examples before we move on to the definition step. If we start with Carl, the first person in the family tree, we see that Carl’s family tree does not contain a node with a “`blue`” eye color. Specifically, the node representing Carl says the eye color is “`green`” and, given that Carl’s ancestor trees are empty, they cannot possibly contain a node with “`blue`” eye color:

```
(check-expect (blue-eyed-child? Carl) #false)
```

In contrast, Gustav contains a node for Eva who does have blue eyes:

```
(check-expect (blue-eyed-child? Gustav) #true)
```

Now we are ready to define the actual function. The function distinguishes between two cases: a `no-parent` node and a `child` node. For the first case, the answer should be

obvious even though we haven't made up any examples. Since the given family tree does not contain any child node whatsoever, it cannot contain one with "blue" as eye color. Hence the result in the first `cond` clause is `#false`.

For the second `cond` clause, the design requires a lot more work. Again following the design recipe, we first remind ourselves what the expressions in the template accomplish:

1. `(child-name a-ftree)` extracts the child's name;
2. `(child-date a-ftree)` extracts the child's date of birth
3. `(child-eyes a-ftree)`, which extracts the child's eye color
4. according to the purpose statement for the function,

`| (blue-eyed-child? (child-father a-ftree))`

determines whether some node in the father's FT has "blue" eyes; and

5. `(blue-eyed-child? (child-mother a-ftree))` determines whether someone in the mother's FT has blue eyes;

It is now time to combine the values of these expressions in an appropriate manner.

Clearly, if the given child structure contains "blue" in the eyes field, the function's answer must be true. Furthermore, the expressions concerning names and birth dates are useless, which leaves us with the two recursive calls. As stated, `(blue-eyed-child? (child-father a-ftree))` processes the father's family tree and `(blue-eyed-child? (child-mother a-ftree))` the mother's. If either of these expressions returns `#true`, `a-ftree` contains a child with "blue" eyes—even if we don't know where exactly this node is located. Since the consideration of the father's tree, the mother's tree, and the child node itself covers all nodes in `a-ftree`, there are no other possibilities.

Our analysis suggests that the result should be `#true` if one of the following three expressions is `#true`:

- `(string=? (child-eyes a-ftree) "blue")`
- `(blue-eyed-child? (child-father a-ftree))`
- `(blue-eyed-child? (child-mother a-ftree))`

which, in turn, means we need to combine these expressions with `or`:

`| (or (string=? (child-eyes a-ftree) "blue")
 (blue-eyed-child? (child-father a-ftree))
 (blue-eyed-child? (child-mother a-ftree)))`

Figure 85 pulls everything together in a single definition.

```
; FT -> Boolean
; does a-ftree contain a child
; structure with "blue" in the eyes field

(check-expect (blue-eyed-child? Carl) #false)
(check-expect (blue-eyed-child? Gustav) #true)
```

```
(define (blue-eyed-child? a-ftree)
  (cond
    [(no-parent? a-ftree) #false]
    [else (or (string=? (child-eyes a-ftree) "blue")
              (blue-eyed-child? (child-father a-ftree))
              (blue-eyed-child? (child-mother a-ftree)))]))
```

Figure 85: Finding a blue-eyed child in an ancestor tree

Since this function is the very first one to use two recursions, we simulate the Stepper's action for `(blue-eyed-child? Carl)` to give you an impression of how it all works:

```
(blue-eyed-child? Carl)
== (blue-eyed-child? (make-child MTFT MTFT "Carl" 1926 "green"))
==
(cond
  [(no-parent? (make-child MTFT MTFT "Carl" 1926 "green")) #false]
  [else
    (or (string=?
          (child-eyes (make-child MTFT MTFT "Carl" 1926 "green"))
          "blue")
        (blue-eyed-child?
          (child-father (make-child MTFT MTFT "Carl" 1926 "green"))))
        (blue-eyed-child?
          (child-mother (make-child MTFT MTFT "Carl" 1926 "green"))))]
==
(or (string=?
      (child-eyes (make-child MTFT MTFT "Carl" 1926 "green"))
      "blue")
    (blue-eyed-child?
      (child-father (make-child MTFT MTFT "Carl" 1926 "green"))))
    (blue-eyed-child?
      (child-mother (make-child MTFT MTFT "Carl" 1926 "green"))))
==
(or (string=? "green" "blue")
    (blue-eyed-child?
      (child-father (make-child MTFT MTFT "Carl" 1926 "green"))))
    (blue-eyed-child?
      (child-mother (make-child MTFT MTFT "Carl" 1926 "green"))))
==
(or #false
    (blue-eyed-child? MTFT)
    (blue-eyed-child? MTFT))
== (or #false #false #false)
== #false
```

The calculation uses several steps where we replace equals by equals, assuming appropriate auxiliary calculations, something the Stepper wouldn't do. For example:

```
(blue-eyed-child? MTFT)
== (blue-eyed-child? (make-no-parent))
==
(cond
  [(no-parent? (make-no-parent)) #false]
  [else
    (or (string=? (child-eyes (make-no-parent)) "blue")
        (blue-eyed-child? (child-father (make-no-parent)))
        (blue-eyed-child? (child-mother (make-no-parent))))])
==
(cond
```

```
[#true #false]
[else
 (or (string=? (child-eyes (make-no-parent)) "blue")
      (blue-eyed-child? (child-father (make-no-parent)))
      (blue-eyed-child? (child-mother (make-no-parent))))]
== #false
```

We trust you have seen such auxiliary calculations in math.

Exercise 314. Develop `count-persons`. The function consumes a family tree node and counts the `child` structures in the tree. ■

Exercise 315. Develop the function `average-age`. It consumes a family tree node and the current year. It produces the average age of all `child` structures in the family tree. ■

Exercise 316. Develop the function `eye-colors`, which consumes a family tree node and produces a list of all eye colors in the tree. An eye color may occur more than once in the resulting list.

Hint Use the `append` operation to concatenate lists:

```
(append (list 'a 'b 'c) (list 'd 'e))
== (list 'a 'b 'c 'd 'e)
```

We discuss the development of functions such as `append` in section [Simultaneous Processing](#). ■

Exercise 317. Suppose we need the function `blue-eyed-ancestor?`. It is like `blue-eyed-child?` but responds with `#true` only when an ancestor, not the given `child` node, has blue eyes.

Even though the functions' purposes clearly differ, their signatures are the same:

```
; FT -> Boolean
(define (blue-eyed-ancestor? a-ftree) ...)
```

Stop! Formulate a purpose statement for the `blue-eyed-ancestor?` function.

To appreciate the difference, we take a look at Eva:

```
(check-expect (blue-eyed-child? Eva) #true)
```

Eva is blue-eyed. Because she does not have a blue-eyed ancestor, however, we get

```
(check-expect (blue-eyed-ancestor? Eva) #false)
```

In contrast, Gustav is Eva's son, he does have a blue-eyed ancestor:

```
(check-expect (blue-eyed-ancestor? Gustav) #true)
```

Now suppose a friend comes up with this solution:

```
(define (blue-eyed-ancestor? a-ftree)
  (cond
    [(no-parent? a-ftree) #false]
    [else (or (blue-eyed-ancestor? (child-father a-ftree))
              (blue-eyed-ancestor? (child-mother a-ftree))))])
```

Explain why this function fails one of its tests. What is the result of `(blue-eyed-ancestor? A)` no matter which `A` you choose? Can you fix your friend's solution? ■

22.2 Forests

It is a short step from a family tree to an entire forest of information about families:

```
; A FF (family forest) is one of:
; - '()
; - (cons FT FF)
```

A family forest could represent the information a town maintains about all its families.

Here are some trees excerpts from [figure 83](#) arranged as forests:

```
(define ff1 (list Carl Bettina))
(define ff2 (list Fred Eva))
(define ff3 (list Fred Eva Carl))
```

The first two forests contain two unrelated families, and the third one illustrates that unlike in real forests, trees in family forests can overlap.

Here is a representative problem concerning family trees:

Sample Problem: Design the function `blue-eyed-child-in-forest?`, which determines whether a family forest contains a child with "blue" in the eyes field.

The straightforward solution is displayed in [figure 86](#).

```
; FF -> Boolean
; does the forest contain any child node with "blue" eyes

(check-expect (blue-eyed-child-in-forest? ff1) #false)
(check-expect (blue-eyed-child-in-forest? ff2) #true)
(check-expect (blue-eyed-child-in-forest? ff3) #true)

(define (blue-eyed-child-in-forest? a-forest)
  (cond
    [(empty? a-forest) #false]
    [else (or (blue-eyed-child? (first a-forest))
              (blue-eyed-child-in-forest? (rest a-forest))))]))
```

Figure 86: Finding a blue eyed child in a family forest

We leave it to you to figure out the signature, the purpose statement, and the examples for this solution. As for the template, the design can employ the list template because `blue-eyed-child-in-forest?` consumes lists. If each item on the list were a structure with an `eyes` field and nothing else, the function would iterate over those structures using the selector function for the `eyes` field and a string comparison. In this case, each item is a family tree but luckily, we already know how to process family trees.

Let us step back for a moment and inspect how we explained [figure 86](#). The starting point is a **pair** of data definitions where the second refers to the first and both refer to themselves. The result is a **pair** of functions where the second refers to the first and both refer to themselves. In other words, the function definitions refer to each other the same way the data definitions refer to each other. To some extent, this relationship is a simple generalization of what we have seen so far. The difference is that **two** data definitions and **two** functions are involved, but not counting the self-references of both data definitions, we have encountered this situation before in some of our projects; we just happen to gloss

over it. Instead of dwelling on the idea in the context of trees, we move on to S-expressions and articulate a generalized design recipe afterwards.

Exercise 318. Reformulate the data definition for `FF` with the `List-of` abstraction. Now do so for the signature of `blue-eyed-child-in-forest?`. Finally, define `blue-eyed-child-in-forest?` using one of the list abstractions from the preceding chapter. ■

Exercise 319. Design the function `average-age`. It consumes a family forest and a year, specified as a natural number. From this data, it produces the average age of all `child` nodes in the forest. **Note** If the trees in this forest overlap, the result isn't a true average because some people may contribute more than others. For this exercise, act as if the trees don't overlap. ■

22.3 S-expressions

While [Intermezzo: Quote, Unquote](#) introduced S-expressions on an informal basis, it is possible to describe them with a combination of three data definitions:

```
; An S-expr (S-expression) is one of:  
; - Atom  
; - SL  
; An SL (S-list) is one of:  
; - '()  
; - (cons S-expr SL)  
; An Atom is one of:  
; - Number  
; - String  
; - Symbol
```

Recall that `Symbols` look like strings with a single quote at the beginning and with no quote at the end.

The idea of S-expressions is due to John McCarthy and his Lispers, who created S-expressions in 1958 so that they could process Lisp programs with other Lisp programs. This seemingly circular reasoning may sound esoteric but as mentioned in [Intermezzo: Quote, Unquote](#), S-expressions are a versatile form of data that is often rediscovered, most recently with applications to the world wide web. Working with S-expressions thus prepares a discussion of how to design functions for highly intertwined data definitions.

Exercise 320. Define the `atom?` function. ■

Up to this point in this book, no data has required a data definition as complex as the one for S-expressions. And yet, with one extra hint, you can design functions that process S-expressions if you follow the design recipe. To demonstrate this point, let us work through a specific example:

Sample Problem: Design the function `count`, which determines how many times some symbol `sy` occurs in some S-expression `sexp`.

While the first step calls for data definitions and appears to have been completed, remember that it also calls for the creation of data examples, especially when the definition is complex.

A data definition is supposed to be a prescription of how to create data, and its “test” is whether it is usable. One point that the data definition for `S-expr` makes is that every `Atom`

is an element of **S-expr**, and you know that **Atoms** are easy to fabricate:

```
'hello
20.12
"world"
```

In the same vein, every **SL** is an **S-expr**, and you know how to make up lists:

```
'()
(cons 'hello (cons 20.12 (cons "world" '())))
(cons (cons 'hello (cons 20.12 (cons "world" '())))
      '())
```

The first two are obvious, the third one deserves a second look. It repeats the second **S-expr** but nested inside `(cons ... '())`. What this means is that it is a list that contains a single item, namely, the second example. You can simplify the example with **list**:

```
(list (cons 'hello (cons 20.12 (cons "world" '()))))
; or
(list (list 'hello 20.12 "world"))
```

Indeed, with the quotation mechanism of **Intermezzo: Quote, Unquote** it is even easier to write down S-expressions. Here are the last three repeated in this notation:

```
> '()
'()
> '(hello 20.12 "world")
'(hello #i20.12 "world")
> '((hello 20.12 "world"))
'((hello #i20.12 "world"))
```

To help you out, we evaluate these examples in the interactions area of DrRacket so that you can see the result, which is closer to the above constructions than the **quote** notation.

With quote, it is quite easy to make up complex examples:

```
> '(define (f x)
      (+ (* 3 x x) (* -2 x) 55))
'(define (f x) (+ (* 3 x x) (* -2 x) 55))
```

This example may strike you as odd because it looks like a definition in BSL but, as the interaction with DrRacket shows, it is just a piece of data. Here is another one:

```
> '((6 f)
  (5 e)
  (4 d))
'((6 f) (5 e) (4 d))
```

This piece of data looks like a table, associating letters with numbers. The last example is a piece of art:

```
> '(wing (wing (wing body wing) wing) wing)
'(wing (wing (wing body wing) wing) wing)
```

It is now time to write down the rather obvious header for count:

```
; S-expr Symbol -> N
; counts all occurrences of sy in sexp
(define (count sexp sy))
```

0)

Since the header is obvious, we move on to functional examples. If the given [S-expr](#) is `'world` and the to-be-counted symbol is `'world`, the answer is obviously `1`. Here are some more examples, immediately formulated as tests:

```
(check-expect (count 'world 'hello) 0)
(check-expect (count '(world hello) 'hello) 1)
(check-expect (count '(((world) hello) hello) 'hello) 2)
```

You can see how convenient quotation notation is for test cases. When it comes to templates, however, thinking in terms of [quote](#) or even [list](#) would be disastrous.

Before we move on to the template step, we need to give you one hint:

For intertwined data definitions, create one template per data definition. Create them in parallel. Make sure they refer to each other in the same way the data definitions do.

This hint sounds more complicated than it is. For our problem, it means we need three templates:

1. one for `count`, which counts occurrences of symbols in [S-exprs](#);
2. one for a function that counts occurrences of symbols in [SLs](#);
3. and one for a function that counts occurrences of symbols in [Atoms](#);

And here are three partial templates, with conditionals as suggested by the three data definitions:

```
(define (count sexp sy)
  (cond
    [(atom? sexp) ...]
    [else ...])))

(define (count-sl sl sy)
  (cond
    [(empty? sl) ...]
    [else ...])))

(define (count-atom at sy)
  (cond
    [(number? at) ...]
    [(string? at) ...]
    [(symbol? at) ...])))
```

The template for `count` contains a two-pronged conditional, because the data definition for [S-expr](#) has two clauses. It uses the `atom?` function to distinguish the case for [Atoms](#) from the case for [SLs](#). The template named `count-sl` consumes an element of [SL](#) and a symbol, and because [SL](#) is basically a list, `count-sl` also contains a two-pronged `cond`. Finally, `count-atom` is supposed to work for [Atoms](#) and [Symbols](#). And this means that its template checks for the three distinct forms of data mentioned in the data definition of [Atom](#).

The next step in the template creation process is to take apart compound pieces of data:

```
(define (count sexp sy)
  (cond
    [(atom? sexp) ...]
    [else ...])))

(define (count-sl sl sy)
  (cond
    [(empty? sl) ...]
    [else (... (first sl) ... (rest sl) ...)]))

(define (count-atom at sy)
  (cond
    [(number? at) ...]
    [(string? at) ...]
    [(symbol? at) ...])))
```

Since only `count-sl` deals with constructed data, we add just two selector expressions.

The last step in the template creation process calls for an inspection of self-references in the data definitions. In our context, this means self-references **and** references from one data definition to another and (possibly) back. Let us inspect the `cond` lines in the three templates:

1. The `atom?` line in `count` corresponds to the first line in the definition of **S-expr**, which points to **Atom**. To indicate this reference from one data definition to the other inside the template, we add `(count-atom sexp sy)`, meaning we interpret `sexp` as an **Atom** and let the appropriate function deal with it.
2. Following the same line of thought, the second `cond` line in `count` calls for the addition of `(count-sl sexp sy)`.
3. The `empty?` line in `count-sl` corresponds to a line in the data definition that makes no reference to another data definition.
4. In contrast, the `else` line contains two selector expressions, and each of them extracts a different kind of value. Specifically, `(first sl)` is an element of **S-expr**, which means that we should wrap it in `(count ...)`. After all, `count` is responsible for counting inside of arbitrary **S-exps**. In the same vein, `(rest sl)` corresponds to a self-reference, and we know that we need to deal with those via recursive function calls.
5. Finally, all three cases in **Atom** refer to atomic forms of data. Therefore the `count-atom` function does not need to change.

```
(define (count sexp sy)
  (cond
    [(atom? sexp) (count-atom sexp sy)]
    [else (count-sl sexp sy)]))

(define (count-sl sl sy)
  (cond
    [(empty? sl) ...]
    [else (... (count (first sl) sy)
               ... (count-sl (rest sl) sy) ...)]))

(define (count-atom at sy)
```

```
(cond
  [(number? at) ...]
  [(string? at) ...]
  [(symbol? at) ...]))
```

Figure 87: A template for S-expressions

Figure 87 presents the three complete templates. Filling in the blanks in these templates is straightforward:

```
; S-expr Symbol -> N
; counts all occurrences of sy in sexp
(define (count sexp sy)
  (cond
    [(atom? sexp) (count-atom sexp sy)]
    [else (count-sl sexp sy)]))

; SL Symbol -> N
; counts all occurrences of sy in sl
(define (count-sl sl sy)
  (cond
    [(empty? sl) 0]
    [else (+ (count (first sl) sy) (count-sl (rest sl) sy))]))

; Atom Symbol -> N
; counts all occurrences of sy in at
(define (count-atom at sy)
  (cond
    [(number? at) 0]
    [(string? at) 0]
    [(symbol? at) (if (symbol=? at sy) 1 0)]))
```

With this idea in mind, you can easily explain any random line in these definitions:

- `[(atom? sexp) (count-atom sexp sy)]` checks whether `sexp` is an atom. Hence, it interprets the `S-expr` as an `Atom` and calls `count-atom` on it. Doing so means the latter function counts how often `sy` occurs in `sexp`—which is precisely what we want, but specialized to the appropriate kind of data.
- `[else (+ (count (first sl) sy) (count-sl (rest sl) sy))]` means the given list consists of two parts: an `S-expr` and an `SL`. By using `count` and `count-sl`, the corresponding functions are used to count how often `sy` appears in each part, and the two numbers are added up—yielding the total number of `sys` in all of `sexp`.
- `[(symbol? at) (if (symbol=? at sy) 1 0)]` tells us that if an `Atom` is a `Symbol`, `sy` occurs once if it is equal to `sexp` and otherwise it does not occur at all. Since the two pieces of data are atomic, there is no other possibility.

Voilà, you have systematically designed a function for S-expressions.

Exercise 321. In a sense, we designed one program that consists of three connected functions. To express this fact, we can use `local` to organize the definitions:

```
; S-expr Symbol -> N
; counts all occurrences of sy in sexp
(define (count sexp sy)
```

```
(local ( ; S-expr Symbol -> N
        ; the main function
      (define (count-sexp sexp sy)
        (cond
          [(atom? sexp) (count-atom sexp sy)]
          [else (count-sl sexp sy)]))

      ; SL Symbol -> N
      ; counts all occurrences of sy in sl
      (define (count-sl sl sy)
        (cond
          [(empty? sl) 0]
          [else (+ (count-sexp (first sl) sy)
                    (count-sl (rest sl) sy))]))

      ; Atom Symbol -> N
      ; counts all occurrences of sy in at
      (define (count-atom at sy)
        (cond
          [(number? at) 0]
          [(string? at) 0]
          [(symbol? at) (if (symbol=? at sy) 1 0)])))

  ; - IN -
  (count-sexp sexp sy)))

```

Notice that we also renamed the first function to indicate that its primary argument is an **S-expr**.

Copy the above definition into DrRacket and validate with the test suite for `count` that it works properly.

The second argument to the local functions, `sy`, never changes. It is always the same as the original symbol. Hence you can eliminate it from the local function definitions and function applications. Do so and re-test the revised definition. In [Simplifying Functions](#), we show you how to simplify these kinds of definitions even more—which is easy to do when you have designed functions systematically. ■

Exercise 322. Design `depth`. The function consumes an S-expression and determines its depth. An atom has a depth of 1. The depth of a list of S-expressions is the maximum depth of its items plus 1. ■

Exercise 323. Design the `substitute` function. It consumes an S-expression `s` and two symbols, `old` and `new`. The result is like `s` with all occurrences of `old` replaced by `new`. ■

Exercise 324. Reformulate the data definition for **S-expr** so that the first clause is expanded into the three clauses of **Atom** and the second clause uses the **List-of** abstraction.

Re-design the `count` function for this data definition. **Note** This kind of simplification is not always possible though experienced programmers tend to recognize and use such opportunities. ■

Exercise 325. Abstract the data definitions for **S-expr** and **SL** so that they abstract over the kinds of atoms that may appear. ■

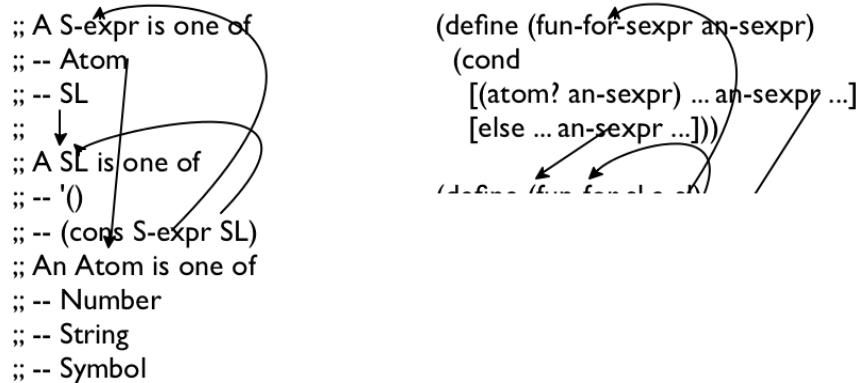


Figure 88: Arrows for nests of data definitions and templates

22.4 Designing with Intertwined Data

The jump from self-referential data definitions to collections of data definitions with mutual references is far smaller than the one from data definitions for finite data to self-referential data definitions. Indeed, the design recipe for self-referential data definitions—see [Designing with Self-Referential Data Definitions](#)—needs only minor adjustments to apply to this seemingly complex situation:

1. The need for “nests” of mutually related data definitions is similar to the one for the need for self-referential data definitions. The problem statement deals with many distinct kinds of information, and one form of information refers to other kinds.

Before you proceed in such situations, use arrows to connect references to definitions. For example, see the left side of [figure 88](#). It displays the definition for `S-expr`, which contains references to `SL` and `Atom` and which are connected to their respective definitions via arrows. Similarly, the definition of `SL` contains one self-reference and one reference back to `SL`; again, both are connected by appropriate arrows.

Like self-referential data definitions, these nests also call for validation. At a minimum, you must be able to construct some examples for every individual data definition.

Before you even start, make sure some of the data definitions contain clauses that do not refer to any of the other data definitions that come with this nest. Keep in mind that the definition may be invalid if it is impossible to generate examples from them.

2. The key change is that you must design as many functions in parallel as there are data definitions. Each function specializes for one of the data definitions; all remaining arguments remain the same. Based on that, you start with a signature, a purpose statement, and a dummy definition **for each function**.
3. Be sure to work through functional examples that use all mutual references in the nest of data definitions.
4. For each function design the template according to its primary data definition. Use [figure 43](#) to guide the template creation up to the last step.

The last template creation step now calls for a check for all self references and cross references. Use the data definitions annotated with arrows to guide this step. For each arrow in the data definitions, include an arrow in the templates. See the right side of [figure 88](#) for the arrow-annotated version of the templates.

Now replace the arrows with actual function calls. As you gain experience, you will naturally skip the arrow-drawing step and use function calls directly.

Note Note how both nests—the one for data definitions and the one for function templates—contain four arrows, and note how pairs of arrows correspond to each other. Researchers call this correspondence a *xsymmetry*. It is evidence that the design recipe provides a natural way for going from problems to solutions.

5. For the design of the body we start with those `cond` lines that do not contain natural recursions or calls to other functions. They are called *base cases*. The corresponding answers are typically easy to formulate or are already given by the examples. After that, you deal with the self-referential cases and the cases of cross-function calls. Let the questions and answers of [figure 44](#) guide you.
5. Run the tests when all definitions are completed. If an auxiliary function is broken, you may get two error report, one for the main function and another one for the flawed auxiliary definition. A **single** fix should eliminate both. Do make sure that running the tests covers all the pieces of the function.

Finally, if you are stuck in step 5, remember the table-based approach to guessing the “combine function.” In the case of intertwined data, you may need not only a table per case but a table per case and per function to work out the combination.

22.5 Project: BSTs

Programmers often work with tree representations of data to improve the performance of their functions. A particularly well-known form of tree is the **binary search tree** because it is a good way to store and retrieve information quickly.

To be concrete, let’s discuss binary trees that manage information about people. Instead of the `child` structures in family trees, a binary tree contains nodes:

```
(define-struct no-info [])
(define NONE (make-no-info))

(define-struct node [ssn name left right])
; A BinaryTree (short: BT) is one of:
; - NONE
; - (make-node Number Symbol BT BT)
```

The corresponding data definition is like the one for family trees with `NONE` indicating a lack of information, and each node recording a social security number, a name, and two other binary trees. The latter are like the parents of family trees, though the relationship between a node and its `left` and `right` trees is not based on family relationships.

Here are two binary trees:

<pre>(make-node 15 'd NONE)</pre>	<pre>(make-node 15 'd (make-node 87 'h NONE NONE))</pre>
---	--

```
(make-node 24 'i NONE NONE))
```

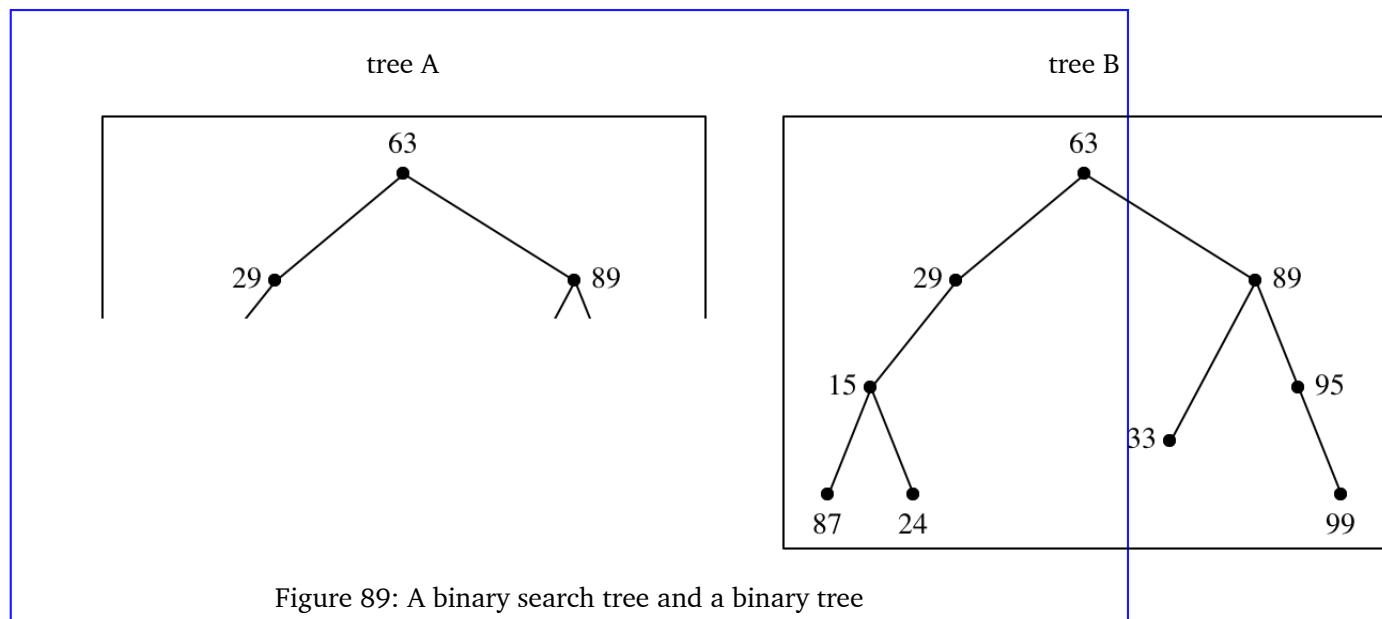
```
NONE)
```

[Figure 89](#) shows how we should think about such trees. The trees are drawn upside down, that is, with the root at the top and the crown of the tree at the bottom. Each circle corresponds to a node, labeled with the `ssn` field of a corresponding node structure. The trees omit `NONE`.

Exercise 326. Draw the above two trees in the manner of [figure 89](#). Then design `contains-bt?`, which determines whether a given number occurs in some given BT. ▀

Exercise 327. Design `search-bt`. The function consumes a number `n` and a BT. If the tree contains a node structure whose `ssn` field is `n`, the function produces the value of the `name` field in that node. Otherwise, the function produces `#false`.

Hint Use either `contains-bt?` to produce `#false` if called for or `boolean?` to find out whether `search-bt` is successfully used on a subtree. ▀



If we read the numbers in the two trees in [figure 89](#) from left to right we obtain two different sequences:

tree A	10	15	24	29	63	77	89	95	99
tree B	87	15	24	29	63	33	89	95	99

The sequence for tree A is sorted in ascending order, the one for B is not. A binary tree that has such an ordered sequence of information is a **binary search tree**. Every binary search tree is a binary tree, but not every binary tree is a binary search tree. More concretely, we formulate a condition—or data invariant—that distinguishes a binary search tree from a binary tree:

The BST Invariant

A *binary search tree* (short: *BST*) is a `BT` according to the following conditions:

- `NONE` is always a **BST**.
- `(make-node ssn0 name0 L R)` is a **BST** if
 - `L` is a **BST**,
 - `R` is a **BST**,

- all `ssn` fields in `L` contain numbers that are smaller than `ssn0`,
- all `ssn` fields in `R` contain numbers that are larger than `ssn0`.

In other words, to check whether a `BT` also belongs to `BST`, we must inspect all numbers in the given subtrees and ensure that they are smaller (or larger) than some given number. This places an additional burden on the construction of sample data but as the rest of this exercise shows, it's well worth it.

Exercise 328. Design the function `inorder`. It consumes a binary tree and produces the sequence of all the `ssn` numbers in the tree as they show up from left to right when looking at a tree drawing.

Hint Use `append`, which concatenates lists like thus:

```
(append (list 1 2 3) (list 4) (list 5 6 7))
==

(list 1 2 3 4 5 6 7)
```

What does `inorder` produce for a binary search tree? ■

Looking for a node with a given `ssn` in a `BST` may exploit the `BST` invariant. To find out whether a `BT` contains a node with a specific `ssn`, a function may have to look at every node of the tree. In contrast, to find out whether a binary `search` tree contains the same `ssn`, a function may eliminate one of two subtrees for every node it inspects.

Let's illustrate this idea with an example. Suppose we are given the `BST`:

```
(make-node 66 'a L R)
```

If we are looking for `66`, we have found the node we are looking for. Now, if we are looking for a smaller number, say `63`, we can focus the search on `L` because **all** nodes with `ssn` fields smaller than `66` are in `L`. Similarly, if we were to look for `99`, we would ignore `L` and focus on `R` because **all** nodes with `ssns` larger than `66` are in `R`.

Exercise 329. Design `search-bst`. The function consumes a number `n` and a `BST`. If the tree contains a node whose `ssn` field is `n`, the function produces the value of the `name` field in that node. Otherwise, the function produces `NONE`. The function organization must exploit the `BST` invariant so that the function performs as few comparisons as necessary.

Compare searching in binary search trees with searching in sorted lists from [exercise 191](#). ■

Building a binary tree is easy; building a binary search tree is complicated. Given any two `BTs`, a number, and a name, we simply apply `make-node` to these values in the correct order, and voilà, we get a new `BT`. This same procedure fails for `BSTs` because the result would typically not be a `BST`. For example, if one `BST` contains nodes with `ssn` fields `3` and `5` in the correct order, and the other one contains `ssn` fields `2` and `6`, simply combining the two trees with another social security number and a name does not produce a `BST`.

The remaining two exercises explain how to build a `BST` from a list of numbers and names. Specifically, the first exercise calls for a function that inserts a given `ssn0` and `name0` into a `BST`; that is, it produces a `BT` like the one it is given with one more node inserted containing `ssn0`, `name0`, and `NONE` subtrees. The second exercise then requests a function that can deal with a complete list of numbers and names.

Exercise 330. Design the function `create-bst`. It consumes a `BST` `B`, a number `N`, and a symbol `S`. It produces a `BST` that is just like `B` and that in place of one `NONE` subtree

contains the node structure

```
(make-node N S NONE NONE)
```

Once you have completed the design, create tree A from [figure 89](#) using `create-bst`.

Exercise 331. Design the function `create-bst-from-list`. It consumes a list of numbers and names and produces a binary search tree by repeatedly applying `create-bst`. Here is the signature:

```
; [List-of [List Number Symbol]] -> BST
```

Once you have completed the design, create a [BST](#) from the following sample input:

```
(define sample
  '((99 o)
    (77 l)
    (24 i)
    (10 h)
    (95 g)
    (15 d)
    (89 c)
    (29 b)
    (63 a)))
```

The result is tree A in [figure 89](#) if you follow the structural design recipe. If you use an existing abstraction, you may still get this tree but you may also get an “inverted” one.

Why?

22.6 Simplifying Functions

[Exercise 321](#) shows how to use `local` to organize a function that deals with an intertwined form of data. This organization also helps simplify functions once we know that the data definition is final. To demonstrate this point, we explain how to simplify the solution of [exercise 323](#).

Here is a complete definition of the `substitute` function, using `local` and three auxiliary functions as suggested by the data definition:

```
; S-expr Symbol Atom -> S-expr
; replaces all occurrences of old in sexp with new

(check-expect (substitute 'world 'hello 0) 'world)
(check-expect (substitute '(world hello) 'hello 'bye) '(world bye))
(check-expect (substitute '(((world) bye) bye) 'bye '42) '(((world) 42) 42))

(define (substitute sexp old new)
  (local (; S-expr -> S-expr
         (define (subst-sexp sexp)
           (cond
             [(atom? sexp) (subst-atom sexp)]
             [else (subst-sl sexp)])))

         ; SL -> S-expr
         (define (subst-sl sl)
```

```
(cond
  [(empty? sl) '()]
  [else (cons (subst-sexp (first sl)) (subst-sl (rest sl))))]

; Atom -> S-expr
(define (subst-atom at)
  (cond
    [(number? at) at]
    [(string? at) at]
    [(symbol? at) (if (symbol=? at old) new at)])))

; - IN -
(subst-sexp sexp)))
```

We have included test cases so that you can re-test the function after each edit.

Exercise 332. Copy and paste the above definition into DrRacket, including the test suite. Run and validate that the test suite passes. As you read along the remainder of this section, perform the edits and re-run the test suites to confirm the validity of our arguments.

Since we know that `SL` describes lists of `S-expr`, we can use `map` to simplify `subst-sl`:

```
(define (substitute.v1 sexp old new)
  (local (; S-expr -> S-expr
          (define (subst-sexp sexp)
            (cond
              [(atom? sexp) (subst-atom sexp)]
              [else (subst-sl sexp)]))

          ; SL -> S-expr
          (define (subst-sl sl)
            (map subst-sexp sl))

          ; Atom -> S-expr
          (define (subst-atom at)
            (cond
              [(number? at) at]
              [(string? at) at]
              [(symbol? at) (if (symbol=? at old) new at)])))

          ; - IN -
          (subst-sexp sexp))))
```

Its original definition says that `subst-sexp` is applied to every item on `sl`; the new definition expresses the same idea more succinctly with `map`.

For the second simplification step, we need to introduce `eq?`, another built-in function of our programming language. All you need to know for now is that `eq?` determines whether two pieces of atomic data are equal. In other words, if `s` is a symbol and `t` is a number, `(eq? s t)` evaluates to `#false` because a string and a number can never be equal; but `(eq? s t)` produces `#true`, if `t` is also a symbol and spells exactly like `s`.

With this in mind, the third `local` function becomes a one-liner:

```
(define (substitute.v2 sexp old new)
  (local (; S-expr -> S-expr
          (define (subst-sexp sexp)
            (cond
```

```

[(atom? sexp) (subst-atom sexp)]
[else (subst-sl sexp))]

; SL -> S-expr
(define (subst-sl sl)
  (map subst-sexp sl))

; Atom -> S-expr
(define (subst-atom at)
  (if (eq? at old) new at))
; - IN -
(subst-sexp sexp))

```

At this point the last two `local` definitions consist of a single line. Furthermore, neither definition is recursive. Hence we can *in-line* the functions in `subst-sexp`, the first `local` function. In-lining means replacing `(subst-atom sexp)` with `(if (eq? sexp old) new sexp)`, that is, we replace the parameter `at` with the actual argument `sexp`. Similarly, for `(subst-sl sexp)` we put in `(map subst-sexp sexp)`:

While `sexp` is also a parameter, this substitution is really acceptable because it, too, stands in for an actual value.

```

(define (substitute.v3 sexp old new)
  (local (; S-expr -> S-expr
          (define (subst-sexp sexp)
            (cond
              [(atom? sexp) (if (eq? sexp old) new sexp)]
              [else (map subst-sexp sexp))]))
        ; - IN -
        (subst-sexp sexp)))

```

All we are left with now is a function whose definition introduces one `local` function, which is called on the same major argument. If we systematically supplied the other two arguments, we would immediately see that the `local` function can be used in lieu of the outer one.

Here is the result of translating this last thought into code:

```

(define (substitute sexp old new)
  (cond
    [(atom? sexp) (if (eq? sexp old) new sexp)]
    [else (map (lambda (s) (substitute s old new)) sexp)]))

```

Stop! Explain why we had to use `lambda` for this last simplification.

23 Incremental Refinement

When you develop real-world programs, you may confront complex forms of information and the problem of representing them with data. The best strategy to approach this task is to use *iterative refinement*, a well-known scientific process. A scientist's problem is to represent a part of the real world using some form of mathematics. The result of the effort is called a model. The scientist then tests the model in many ways, in particular by predicting the outcome of experiments. If the discrepancies between the predictions and the measurements are too large, the model is refined with the goal of improving the

predictions. This iterative process continues until the predictions are sufficiently accurate.

Consider a physicist who wishes to predict a rocket's flight path. While a "rocket as a point" representation is simple, it is also quite inaccurate, failing to account for air friction. In response, the physicist may add the rocket's rough contour and introduce the necessary mathematics to represent friction. This second model is a *refinement* of the first model. In general, a scientist *iterates* this process until the model predicts the rocket's flight path with sufficient accuracy.

A programmer trained in a computer science department should proceed like this physicist. The key is to find an accurate data representation of the real-world information and functions that process them appropriately. Complicated situations call for a refinement process to get to a sufficient data representation combined with the proper functions. The process starts with the essential pieces of information and adds others as needed.

Sometimes a programmer must refine a model **after** the program has been deployed because users request additional functionality.

So far we have used iterative refinement for you when it came to complex forms of data. This chapter illustrates iterative refinement as a principle of program development with an extended example, representing and processing (portions of) a computer's file system. We start with a brief discussion of the file system and then iteratively develop three data representations. Along the way, we propose some programming exercises so that you see how the design recipe also helps modify existing programs.

23.1 Data Analysis

Before you turn off DrRacket, you want to make sure that all your work—your programs, your data—are stashed away somewhere. Otherwise you have to reenter everything when you fire up DrRacket next. So you ask your computer to save programs and data in *files*. A file is a sequence of small pieces of *characters*, also called *bytes*. For our purposes here, a file resembles a string.

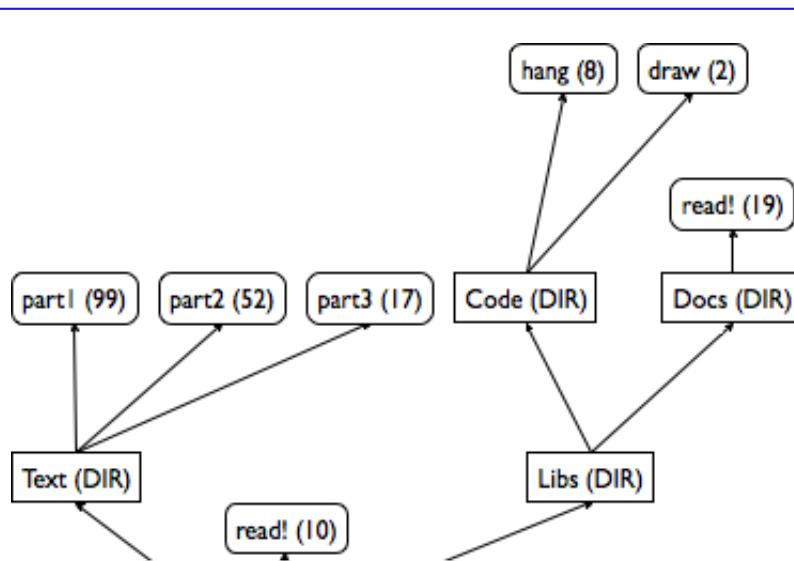


Figure 90: A sample directory tree

On most computer systems, files are organized in *directories* or *folders*. Roughly speaking, a

directory contains some files and some more directories. The latter are called subdirectories and may contain yet more subdirectories and files, and so on. The entire collection is also called a *directory tree*.

[Figure 90](#) contains a graphical sketch of a small directory tree, and the picture explains why computer scientists call them trees. Contrary to computer scientists, the figure shows the tree growing upwards, with a root directory named `TS`. The root directory contains one file, called `read!`, and two subdirectories, called `Text` and `Libs`, respectively. The first subdirectory, `Text`, contains only three files; the latter, `Libs`, contains only two subdirectories, each of which contains at least one file. Finally each box has one of two annotations: a directory is annotated with `DIR`, and a file is annotated with a number, its size.

Exercise 333. How many times does a file name `read!` occur in the directory tree `TS`? Can you describe the path from the root directory to the two occurrences? What is the total size of all the files in the tree? What is the total size of the directory if each directory node has size [1](#)? How many levels of directories does it contain? ■

23.2 Refining Data Definitions

[Exercise 333](#) lists some of the questions that users routinely ask about directories. To answer such questions, the computer's operating system provides programs that can answer just such questions. If you want to design such programs, you need to develop a data representation for directory trees.

In this section, we use iterative refinement to develop three such data representations. For each stage, we need to decide which attributes to include and which to ignore. Consider the directory tree in [figure 90](#) and imagine how it is created. When a user first creates a directory, it is empty. As time goes by, the user adds files and directories. In general, a user refers to files by names but mostly thinks of directories as containers of other things.

Model 1 Our thought experiment suggests that our first model should focus on files as atomic entities with a name and directories as containers. Here is a data definition that deals with directories as lists and files as symbols, i.e., their names:

```
; A Dir.v1 (short for directory) is one of:  
; - '()  
; - (cons File.v1 Dir.v1)  
; - (cons Dir.v1 Dir.v1)  
  
; A File.v1 is a Symbol.
```

The names have a `.v1` suffix to distinguish them from future refinements.

Exercise 334. Translate the directory tree in [figure 90](#) into a data representation according to model 1. ■

Exercise 335. Design the function `how-many`, which determines how many files a given `Dir.v1` contains. Remember to follow the design recipe; [exercise 334](#) provides you with data examples.

Model 2 If you solved [exercise 335](#), you know that this first data definition is still reasonably simple. But, it also obscures the nature of directories. With this first representation, we would not be able to list all the names of the subdirectories of some given directory. To model directories in a more faithful manner than containers, we must

introduce a structure type that combines a name with a container:

```
(define-struct dir [name content])
```

This new structure type, in turn, suggests the following revision of the data definition:

```
; A Dir.v2 is a structure:  
;   (make-dir Symbol LOFD)  
  
; A LOFD (short for list of files and directories) is one of:  
; - ()  
; - (cons File.v2 LOFD)  
; - (cons Dir.v2 LOFD)  
  
; A File.v2 is a Symbol.
```

Note how the data definition for `Dir.v2` refers to the definition for `LOFDs` and the one for `LOFDs` refers back to that of `Dir.v2`. The two definitions are mutually recursive.

Exercise 336. Translate the directory tree in [figure 90](#) into a data representation according to model 2.

Exercise 337. Design the function `how-many`, which determines how many files a given `Dir.v2` contains. [Exercise 336](#) provides you with data examples. Compare your result with that of [exercise 335](#).

Exercise 338. Show how to equip a directory with two more attributes: size and readability. The former measures how much space the directory itself (as opposed to its files and subdirectories) consumes; the latter specifies whether anyone else besides the user may browse the content of the directory.

Model 3 Like directories, files have attributes. To introduce these, we proceed just as above. First, we define a structure for files:

```
(define-struct file [name size content])
```

Second, we provide a data definition:

```
; A File.v3 is a structure:  
;   (make-file Symbol N String)
```

As indicated by the field names, the symbol represents the name of the file, the natural number its size, and the string its content.

Finally, let us split the content field of directories into two pieces: a list of files and a list of subdirectories. This change requires a revision of the structure type definition:

```
(define-struct dir.v3 [name dirs files])
```

Here is the refined data definition:

```
; A Dir.v3 is a structure:  
;   (make-dir.v3 Symbol Dir* File*)  
  
; A Dir* is one of:  
; - ()  
; - (cons Dir.v3 Dir*)  
  
; A File* is one of:
```

```
| ; - '()
| ; - (cons File.v3 File*)
```

Following a convention in computer science, the use of `*` as the ending of a name suggests “many” and is a marker distinguishing the name from similar ones: `File.v3` and `Dir.v3`.

Exercise 339. Translate the directory tree in [figure 90](#) into a data representation according to model 3. Use `""` for the content of files.

Exercise 340. Design the function `how-many`, which determines how many files a given `Dir.v3` contains. [Exercise 339](#) provides you with data examples. Compare your result with that of [exercise 337](#).

Given the complexity of the data definition, contemplate how anyone can design correct functions. Why are you confident that `how-many` produces correct results?

Exercise 341. Use [List-of](#) to simplify the data definition `Dir.v3`. Then use ISL+’s list processing functions from [figure 71](#) to simplify the function definition(s) for the solution of [exercise 340](#).

Starting with a simple representation of the first model and refining it step by step, we have developed a reasonably accurate data representation for directory trees. Indeed, this third data representation captures the nature of a directory tree much more faithfully than the first two. Based on this model, we can create a number of other functions that users expect from a computer’s operating system.

23.3 Refining Functions

To make the following exercises somewhat realistic, DrRacket comes with the `dir.ss` library from the first edition of this book. This teachpack introduces the two structure type definitions from model 3, though without the `.v3` suffix. Furthermore, the teachpack provides a function that creates model 3-style data representations of directory trees on your computer:

```
| ; String -> Dir.v3
| ; creates a data representation of the directory that a-path identifies
| (define (create-dir a-path) ...)
```

For example, if you open DrRacket and enter the following three lines into the definitions area:

```
| (require htdp/dir)
| (define d0 (create-dir "/Users/...")) ; on OS X
| (define d1 (create-dir "/var/log/")) ; on Linux/Unix
| (define d2 (create-dir "C:\\\\Users\\\\...")) ; on Windows
```

you get data representations of directories on your computer after you **save** and then run the program. Indeed, you could use `create-dir` to map the entire file system on your computer to an instance of `Dir.v3`. Warnings: (1) For large directory trees, DrRacket may need a lot of time to build a representation. Use `create-dir` on small directory trees first. (2) Do **not** define your own `dir` structure type. The teachpack already defines them, and you must not define a structure type twice.

The function informs you via print outs about inaccessible directories.

Although `create-dir` delivers only a representation of a directory tree, it is sufficiently realistic to give you a sense of what it is like to design programs at that level. The following exercises illustrate this point. They use `Dir` to refer to the generic idea of a data representation for directory trees. Use the simplest data definition of `Dir` that allow you to complete the respective exercise. Feel free to use the data definition from [exercise 341](#) and the functions from [figure 71](#).

Exercise 342. Use `create-dir` to create data representations of some sample directories on your computer. Then use `how-many` from [exercise 340](#) to count how many files they contain. Why are you confident that `how-many` produces correct results for these directories?

Exercise 343. Design `find?`. The function consumes a `Dir` and a file name and determines whether or not a file with this name occurs in the directory tree.

Exercise 344. Design the function `ls`, which lists the names of all files and directories in a given `Dir`.

Exercise 345. Design `du`, a function that consumes a `Dir` and computes the total size of all the files in the entire directory tree. Assume that storing a directory in a `Dir` structure costs 1 file storage unit. In the real world, a directory is basically a special file and its size depends on how large its associated directory is.

The remaining exercises rely on the notion of a path, which for our purposes, is a list of names:

```
; Path = [List-of Symbol]
; interpretation directions on how to find a file in a directory tree
```

For example, the path from `TS` to `part1` in [figure 90](#) is (`list 'TS 'Text 'part1`). Similarly, the path from `TS` to `Code` is (`list 'TS 'Libs 'Code`).

Exercise 346. Design `find`. The function consumes a directory `d` and a file name `f`. If `(find? d f)` is true, `find` produces a path to a file with name `f`; otherwise it produces `#false`.

Hint While it is tempting to first check whether the file name occurs in the directory tree, you have to do so for every single subdirectory. Hence it is better to combine the functionality of `find?` and `find`.

Challenge: The `find` function discovers only one of the two files named `read!` file in [figure 90](#). Design `find-all`, which is generalizes `find` and produces the list of all paths that lead to `f` in `d`. What should `find-all` produce when `(find? f d)` is `#false`? Is this part of the problem really a challenge compared to the basic problem?

Exercise 347. Design the function `ls-R`, which lists the paths to all files in a given `Dir`.

Challenge: Modify `ls-R` so that its result includes all paths to directories, too.

Exercise 348. Re-design `find-all` from [exercise 346](#) using `ls` from [exercise 347](#). This is design by composition, and if you solved the challenge part of [exercise 347](#) your new function can find directories, too.

24 Refining Interpreters

DrRacket is a program. As you can imagine, it is a complex program, dealing with many

different kinds of data. Like most complex programs, DrRacket also consists of many functions: one that allows programmers to edit text; another one that acts like the interactions area; a third one checks whether definitions and expressions are grammatical; and so on.

In this chapter, we show you how to design the function that implements the heart of the interactions area. Naturally, we use iterative refinement to design this evaluator. As a matter of fact, the very idea of focusing on the evaluator part of DrRacket is another instance of refinement, namely, the obvious one of implementing only one piece of functionality for a complex program.

Simply put, the interactions area performs the task of determining the values of expressions that you enter. After you click *RUN*, the interactions area knows about all the definitions. It is then ready to accept an expression that may refer to these definitions, to determine the value of this expression, and to repeat this cycle as often as you wish. For this reason, many people also refer to the interactions area as the *read-eval-print loop*, where *eval* is short for *evaluator*, a function is also called an *interpreter*.

Like this book, our refinement process starts with numeric BSL expressions. They are simple; they do not assume an understanding of definitions; and even your brother in fourth grade can determine their value. Once you understand this first step, you know the difference between a BSL expression and its representation. Next we move on to expressions with variables. The last step is to add definitions.

24.1 Interpreting Expressions

Our first task is to agree on a data representation for BSL programs, that is, we must figure out how to represent a BSL expression as a piece of BSL data. This sounds strange and unusual, but it is not difficult. Suppose we just want to represent numbers, additions, and multiplications for a start. Clearly, numbers can stand for numbers. Additions and multiplications, however, call for a class of compound data because they consist of an operator and two pieces.

Following the design recipes of this book, a straightforward way to represent additions and multiplications is to define two structure types: one for additions and another one for multiplications, each with two fields:

```
(define-struct add [left right])
(define-struct mul [left right])
```

The intention is that the `left` field contains one operand—the one to the “left” of the operator—and the `right` field contains the other operand. The following table shows three examples:

BSL expression	representation of BSL expression
3	3
(+ 1 1)	(make-add 1 1)
(* 300001 100000)	(make-mul 300001 100000)

The next question is how we should represent an expression with sub-expressions:

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

The surprisingly simple answer is that fields may contain any value. In this particular case, `left` and `right` may contain representations of expressions; and you may nest this as deep as you wish as the next table shows:

BSL expression	representation of BSL expression
(+ (* 1 1) 10)	(make-add (make-mul 1 1) 10)
(+ (* 3 3))	(make-add (make-mul 3 3))
(* 4 4))	(make-mul 4 4))
(+ (* (+ 1 2) 3) (* (* (+ 1 1) 2) 4))	(make-add (make-mul (make-add 1 2) 3) (make-mul (make-mul (make-add 1 1) 2) 4))

As you can see, these examples cover all cases: numbers, variables, simple expressions, and nested expressions.

Exercise 349. Formulate a data definition for the representation of BSL expressions based on the structure type definitions of `add` and `mul`. Let us use *BSL-expr* in analogy for *S-expr* for the new class of data.

Translate the following expressions into data:

1. $(+ \ 10 \ -10)$
 2. $(+ \ (* \ 20 \ 3) \ 33)$
 3. $(+ \ (* \ 3.14 \ (* \ 2 \ 3)) \ (* \ 3.14 \ (* \ -1 \ -9)))$

Interpret the following data as expressions:

1. (make-add -1 2)
 2. (make-add (make-mul -2 -3)
33)
 3. (make-mul (make-add 1 (make-
mul 2 3)) (make-mul 3.14
12))

Remember that “interpret” here means “translate from data into information.” In contrast, the word “interpreter” in the title of this chapter refers to a program that consumes the representation of a program and produces its value. While the two ideas are related, they are not the same.

Now that you have a data representation for BSL programs, it is time to design an evaluator. This function consumes a representation of a BSL expression and produces its value. Again, this function is unlike any you have ever designed so it pays off to experiment with some examples. To this end, you can either use the rules of arithmetic to figure out what the value of an expression is or you can “play” in the interactions area of DrRacket. Take a look at the following table for our examples:

BSL expression	its representation	its value
3	3	3
(+ 1 1)	(make-add 1 1)	2
(* 3 10)	(make-mul 3 10)	30
(+ (* 1 1) 10)	(make-add (make-mul 1 1) 10)	11

Exercise 350. Formulate a data definition for the class of values to which a representation of a BSL expression can evaluate.

Exercise 351. Design eval-expression. The function consumes a representation of a BSL expression (according to exercise 349) and computes its value.

Exercise 352. Develop a data representation for boolean BSL expressions constructed from `#true`, `#false`, `and`, `or`, and `not`. Then design `eval-bool-expression`. The function

consumes a representative of boolean BSL expression and computes its value. What is the value of a Boolean expression?

Exercise 353. S-expressions offer a convenient way to express BSL:

```
> (+ 1 1)
2
> '(+ 1 1)
'(+ 1 1)
> (+ (* 3 3) (* 4 4))
25
> '(+ (* 3 3) (* 4 4))
'(+ (* 3 3) (* 4 4))
```

By simply putting a quote in front of an expression, we get a piece of ISL+ data. This representation is convenient only for the person who types the representation of a BSL expression on a keyboard. Interpreting an S-expression representation is clumsy, mostly because not all S-expressions represent [BSL-exprs](#):

```
"hello world"
'(+ x 1)
'(* (- "hello" "world") 10)
```

People invented *parsers* to check whether some piece of data conforms to a data definition. A parser consumes a “raw” piece of input and, if it does conform, it produces a *parse tree*, which in our specific case is the corresponding [BSL-expr](#). If not, it signals an error, like the checked functions from [Input Errors](#).

[Figure 91](#) presents a BSL parser for S-expressions. Specifically, the `parse` function consumes an [S-expr](#) and produces an [BSL-expr](#)—if and only if the given S-expression is the result of quoting a BSL expression that has a [BSL-expr](#) representative.

Create test cases for the `parse` function until DrRacket tells you that all expressions in the definitions area are covered during the test run.

What is unusual about the definition of this program with respect to the design recipe?

Note One unusual aspect is that the function uses `length` on the list argument. Real parsers do not use `length` because it slows the functions down.

Discuss: should a programming language be designed for the convenience of the programmer who uses it or for the convenience of the programmer who implements it?

```
(define WRONG "wrong kind of S-expression")

(define-struct add [left right])
(define-struct mul [left right])

; S-expr -> BSL-expr
; creates representation of a BSL expression for s (if possible)
(define (parse s)
  (local (; S-expr -> BSL-expr
         (define (parse s)
           (cond
             [(atom? s) (parse-atom s)]
             [else (parse-sl s)]))

         ; SL -> BSL-expr
```

```

(define (parse-sl s)
  (local ((define L (length s)))
    (cond
      [(< L 3)
       (error WRONG)]
      [(and (= L 3) (symbol? (first s)))
       (cond
         [(symbol=? (first s) '+)
          (make-add (parse (second s)) (parse (third s)))]
         [(symbol=? (first s) '*)
          (make-mul (parse (second s)) (parse (third s)))]
         [else (error WRONG)])]
      [else
       (error WRONG)])))

; Atom -> BSL-expr
(define (parse-atom s)
  (cond
    [(number? s) s]
    [(string? s) (error "strings not allowed")]
    [(symbol? s) (error "symbols not allowed")]))
(parse s)))

```

Figure 91: From S-expr to BSL-expr

24.2 Interpreting Variables

Since the first section ignores constant definitions, an expression does not have a value if it contains a variable. Indeed, if we do not know what x stands for, it makes no sense to evaluate $(+ 3 x)$. Hence, one first refinement of the evaluator is to add variables to the expressions that we wish to evaluate. The assumption is that the definitions area contains a definition such as

```
| (define x 5)
```

and that someone wants to evaluate expressions containing x in the interactions area:

```

> x
5
> (+ x 3)
8
> (* 1/2 (* x 3))
7.5

```

Indeed, you could imagine a second constant definition, say `(define y 3)`, and interactions that involve two variables:

```

> (+ (* x x)
      (* y y))
34

```

Exercise 353 implicitly proposes symbols as representations for variables. After all, if you were to choose quoted S-expressions to represent expressions with variables, symbols would appear naturally:

```
| > 'x
```

```
'x
> '(* 1/2 (* x 3))
'(* 0.5 (* x 3))
```

The obvious alternative is a string, so that "`x`" would represent `x`. Other representations are also possible but this book is not about designing evaluators, so we stick with symbols. From this decision, it follows how to modify the data definition from [exercise 349](#):

```
; A BSL-var-expr is one of:
; - Number
; - Symbol
; - (make-add BSL-var-expr BSL-var-expr)
; - (make-mul BSL-var-expr BSL-var-expr)
```

We simply add one clause to the data definition.

As for data examples, the following table shows some BSL expressions with variables and their `BSL-var-expr` representation:

BSL expression	representation of BSL expression
<code>x</code>	<code>'x</code>
<code>(+ x 3)</code>	<code>(make-add 'x 3)</code>
<code>(* 1/2 (* x 3))</code>	<code>(make-mul 1/2 (make-mul 'x 3))</code>
<code>(+ (* x x))</code>	<code>(make-add (make-mul 'x 'x))</code>
<code>(* y y))</code>	<code>(make-mul 'y 'y))</code>

They are all taken from the interactions above, meaning you know the results when `x` is `5` and `y` `3`.

One way to determine the value of variable expressions is to replace all variables with the values that they represent. This is the way you know from mathematics classes in school, and it is perfectly fine way.

Exercise 354. Design `subst`. The function consumes a `BSL-var-expr` `ex`, a `Symbol` `x`, and a `Number` `v`. It produces a `BSL-var-expr` like `ex` with all occurrences of `x` replaced by `v`.

Exercise 355. Design the function `numeric?`, which determines whether a `BSL-var-expr` is also a `BSL-expr`. Here we assume that your solution to [exercise 349](#) is the definition for `BSL-var-expr` without the line for `Symbol`.

Exercise 356. Design `eval-variable`. The function consumes a `BSL-var-expr` and determines its value if `numeric?` is true. Otherwise it signals an error, saying that it is impossible to evaluate an expression that contains a variable.

In general, a program defines many constants in the definitions area and expressions contain more than one variable. To evaluate such expressions, we need a representation of the definition area when it contains a series of constant definitions. For this exercise we use association lists:

```
; An AL (association list) is [List-of Association].
; An Association is (cons Symbol (cons Number '())).
```

Make up elements of `AL`.

Design `eval-variable*`. The function consumes a `BSL-var-expr` `ex` and an association list `da`. Starting from `ex`, it iteratively applies `subst` to all associations in `da`. If `numeric?` holds for the result, it determines its value; otherwise it signals the same error as `eval-variable`. **Hint** Think of the given `BSL-var-expr` as an atomic value and traverse the given

association list instead. Or use a loop from [figure 71](#). We provide this hint because the creation of this function requires a little design knowledge from [Simultaneous Processing](#).

Exercise 357. Modify the parser in [figure 91](#) so that it creates `BSL-var-expr` if it is an appropriate BSL expression.

[Exercise 356](#) relies on the mathematical approach to constant definitions. If a name is defined to stand for some value, then all occurrences of the name can be replaced with the value. Substitution performs this replacement once and for all before the evaluation process even starts.

An alternative approach is to mingle substitution and replacement. That is, the evaluator starts processing the expression immediately but also carries along the representation of the definitions area. Every time the evaluator encounters a variable, it looks in the definitions area for its value and uses it.

Exercise 358. Design `lookup-con`. The function consumes an `AL da` and a `Symbol x`. It produces the value of `x` in `da`—if there is a matching [Association](#); otherwise it signals an error.

Exercise 359. Design `eval-var-lookup`. This function has the same signature as `eval-variable*`:

```
; BSL-var-expr AL -> Number
(define (eval-var-lookup e da) ...)
```

It does not use substitution, however. Instead, the function traverses the expression in the manner that the design recipe for `BSL-var-expr` suggests and “carries along” `da`. When it encounters a symbol `x`, the function looks up the value of `x` in `da`.

24.3 Interpreting Functions

At this point, you understand how to evaluate BSL programs that consist of constant definitions and variable expressions. Naturally you want to add function definitions so that you know—at least in principle—how to deal with a complete programming language such as BSL.

The goal of this section is to refine the evaluator of [Interpreting Variables](#) so that it can cope with function applications, assuming a function definition is given. Put differently, we want to design an evaluator with you that simulates DrRacket when the definitions area contains a number of function definitions and a programmer enters an expression in the interactions area that contains applications of these functions.

For simplicity, let us assume that all functions in the definitions area consume one argument and, for now, let’s assume that there is only one such definition. The domain knowledge you need again dates back to school where you learn that function applications of the shape $f(a)$ are evaluated by substituting a for x in e , assuming the definition for f is $f(x) = e$. As it turns out, the evaluation of function applications in a language such as BSL works like that, too.

Exercise 360. Extend the data representation of [Interpreting Variables](#) to include the application of a programmer-defined function. Recall that a function application consists of two pieces: a name and an expression. The former is the name of the function that is applied; the latter is the argument.

Use your data definition to represent the following expressions:

1. `(k (+ 1 1))`
2. `(* 5 (k (+ 1 1)))`
3. `(* (i 5) (k (+ 1 1)))`

We refer to this newly defined class of data with *BSL-fun-expr*.

Exercise 361. Design `eval-definition1`. The function is given an expression (representation) in the extended data definition of [exercise 360](#) and the one function definition that is assumed to exist in the definitions area. It evaluates the given expression and returns its value.

Specification, `eval-definition1` consumes four arguments:

1. a [BSL-fun-expr](#) `ex`;
2. a symbol `f`, which represents a function name;
3. a symbol `x`, which represents the functions's parameter; and
4. a [BSL-fun-expr](#) `b`, which represents the function's body.

If the terminology poses any difficulties, do re-read [BSL Grammar](#).

To determine the value of `ex`, the function proceeds as before. When it encounters an application of `f` to some argument,

1. `eval-definition1` evaluates the argument,
2. substitutes the value of the argument for `x` in `b`; and
3. finally evaluates the resulting expression with `eval-definition1`.

Here is how to express the steps as code, assuming `arg` is the argument of the function application:

```
| (eval-definition1 (subst b x (eval-definition1 arg f x b)) f x b)
```

Notice that this line uses a form of recursion that you have not encountered so far. The proper design of such functions is discussed in [Generative Recursion](#).

If `eval-definition1` encounters a variable, it signals the same error as `eval-variable` from [exercise 356](#). Also, for function applications that do not refer to `f`, `eval-definition1` signals an error as if it had encountered a variable.

Warning The use of generative recursion introduces a new element into your computations: non-termination. That is, given some argument, a program may not deliver a result or signal an error but run forever. For fun, you may wish to construct an input for `eval-definition1` that causes it to run forever. Use *STOP* to terminate the program.

For an evaluator that mimics the interaction area, we need a representation of the definitions area. Like in [Interpreting Variables](#), we assume that it is a list of definitions.

Exercise 362. Provide a structure type definition and a data definition for function definitions. Recall that a function definition has three essential attributes:

1. the function's name,
2. the function's parameter, which is also a name, and
3. the function's body, which is a variable expression that usually contains the parameter.

We use *BSL-fun-def* to refer to the class of data representations for function definitions.

Use your data definition to represent the following BSL function definitions:

1. `(define (f x) (+ 3 x))`
2. `(define (g y) (f (* 2 y)))`
3. `(define (h v) (+ (f v) (g v)))`

Next, define the class *BSL-fun-def** to represent definitions area that consist of just one-argument function definitions. Translate the definitions area that defines f, g, and h into your data representation and name it da-fgh.

Finally, design the function `lookup-def` with the following header:

```
; BSL-fun-def* Symbol -> BSL-fun-def
; retrieves the definition of f in da
; or signal "undefined function" if da does not contain one
(check-expect (lookup-def da-fgh 'g) g)
(define (lookup-def da f) ...)
```

Looking up a definition is needed for the evaluation of expressions in *BSL-fun-expr*.

Exercise 363. Design `eval-function*`. The function consumes the *BSL-fun-expr* representation of some expression ex and the *BSL-fun-def** representation of a definitions area da. It produces the result that DrRacket shows if you evaluate ex in the interactions area assuming the definitions area contains da.

The function works like `eval-definition1` from [exercise 361](#). For an application of some function f, it

1. evaluates the argument;
2. looks up the definition of f in the *BSL-fun-def* representation of da;
3. substitutes the value of the argument for the function parameter in the function's body; and
4. evaluates the new expression via recursion.

Remember that the representation of a function definition for f comes with a parameter and a body.

Like DrRacket, `eval-function*` signals an error when it encounters a variable or an application whose function is not defined in the definitions area.

Exercise 364. Modify the parser in [figure 91](#) so that it creates *BSL-fun-expr* if it is an appropriate BSL expression. Also see [exercise 357](#).

Exercise 365. [Figure 92](#) presents a BSL definitions parser for S-expressions. Specifically, the `def-parse` function consumes an *S-expr* and produces a *BSL-fun-def*—if and only if the given S-expression is the result of quoting a BSL definition that has a *BSL-fun-def*

representative.

Create test cases for the `def-parse` function until DrRacket tells you that all expressions in the definitions area are covered during the test run.

Note The exercises assumes that you have a solution for [exercise 364](#). That is, you have a function `parse` that turns an S-expr into a BSL-fun-expr representation, if possible.

With `def-parse` you have the essential ingredient for a parser that consumes an S-expression representation of a definitions area and produces a [BSL-fun-def*](#). Now design the function `da-parse`, which parses a [SL](#) as a [BSL-fun-def*](#) assuming the former is a list of quoted BSL definitions.

```
(define WRONG "wrong kind of S-expression")

(define-struct def [name para body])
; see exercise 362

; S-expr -> BSL-fun-def
; creates representation of a BSL definition for s (if possible)
(define (def-parse s)
  (local (; S-expr -> BSL-fun-def
         (define (def-parse s)
           (cond
             [(atom? s) (error WRONG)]
             [else
               (if (and (= (length s) 3) (eq? (first s) 'define))
                   (head-parse (second s) (parse (third s)))
                   (error WRONG))]))
         ; S-expr BSL-expr -> BSL-fun-def
         (define (head-parse s body)
           (cond
             [(atom? s) (error WRONG)]
             [else
               (if (not (= (length s) 2))
                   (error WRONG)
                   (local ((define name (first s))
                           (define para (second s)))
                     (if (and (symbol? name) (symbol? para))
                         (make-def name para body)
                         (error WRONG))))])))
       (def-parse s)))
```

Figure 92: From S-expr to BSL-fun-def

24.4 Interpreting Everything

Take a look at the following BSL program:

```
(define close-to-pi 3.14)

(define (area-of-circle r)
  (* close-to-pi (* r r)))

(define (volume-of-10-cylinder r h)
```

```
(* 10 (area-of-circle r)))
```

Think of these definitions as the definitions area in DrRacket. After you click *RUN*, you can evaluate expressions involving `close-to-pi`, `area-of-circle`, and `volume-of-10-cylinder` at the prompt in the interactions area:

```
> (area-of-circle 1)
#i3.14
> (volume-of-10-cylinder 1)
#i31.40000000000000002
> (* 3 close-to-pi)
#i9.42
```

The goal of this section is to refine your evaluator again so that it can mimic this much of DrRacket.

Exercise 366. Formulate a data definition for the representation of DrRacket's definition area. Concretely, the data representation should work for a sequence that freely mixes constant definitions and one-argument function definitions. Make sure you can represent the definitions area consisting of three definitions at the beginning of this section.—We use *BSL-da-all* for this class of data.

Design the function `lookup-con-def`, It consumes a *BSL-da-all* da and a symbol *x*. It produces the representation of a constant definition whose name is *x*, if such a piece of data exists in da; otherwise the function signals an error saying that no such constant definition can be found.

Design the function `lookup-fun-def`, It consumes a *BSL-da-all* da and a symbol *f*. It produces the representation of a function definition whose name is *f*, if such a piece of data exists in da; otherwise the function signals an error saying that no such function definition can be found.

Exercise 367. Design `eval-all`. Like `eval-function*` from [exercise 363](#), this function consumes the representation of an expression and a definitions area. It produces the same value that DrRacket shows if the expression is entered at the prompt in the interactions area and the definitions area contains the appropriate definitions. **Hint** Your `eval-all` function should process variables in the given expression like `eval-var-lookup` in [exercise 359](#).

Exercise 368. It is cumbersome to enter the structure-based data representation of a BSL expressions and a definitions area. It is much easier to quote an actual expression or a definitions area after surrounding it with parentheses.

Design a function `eval-all-sexpr`. It consumes an *S-expr* and an *sl*. The former is supposed to represent an expression and the latter a list of definitions. The function parses both with the appropriate parsing functions and then uses `eval-all` from [exercise 367](#) to evaluate the expression. **Hint** You must slightly modify `da-parse` from [exercise 365](#) so that it can parse constant definitions, too.

You should know that `eval-all-sexpr` makes it straightforward to check whether it really mimics DrRacket's evaluator.

At this point, you know a lot about interpreting BSL. Here are some of the missing pieces: `Booleans` with `cond` or `if`; `Strings` and such operations `string-length` or `string-append`; and lists with `'()`, `empty?`, `cons`, `cons?`, `first`, `rest`; and so on. Once your evaluator can cope with all these, it is basically complete, because your evaluators already

know how to interpret recursive functions. Now when we say “trust us, you know how to design these refinements,” we mean it.

25 The Commerce of XML

XML is a widely used data language. One use concerns message exchanges between programs running on different computers. For example, when you point your web browser at a web site, you are connecting a program on your computer to a program on another computer, and the latter sends XML data to the former. Once the browser receives the XML data, it renders it as an image on your computer’s monitor.

The following table illustrates this idea with a concrete example:

XML data	rendered in a browser
<pre> hello one two world good bye </pre>	

On the left, you see a piece of XML data that a web site may send to your web browser. On the right, you see how one popular browser renders this snippet graphically.

This chapter explains the basics of processing XML as another design exercise concerning

intertwined data definitions and

iterative refinement. [XML as S-](#)

[expressions](#) starts with an informal comparison of S-expressions and XML data and uses it to formulate a full-fledged data definition. The remaining sections explain with examples how to process an S-expression of XML data.

[Rendering XML Enumerations](#) explains how to render enumerations like the above;

[Domain-Specific Languages](#) illustrates how to use XML files to create a small language for configuring programs, a common mechanism for modern applications.

If you think XML is too old-fashioned for 2015, remember that this chapter is an exercise. Feel free to re-do the exercise for JSON or some other modern data exchange format. The design will remain the same.

25.1 XML as S-expressions

The most basic piece of XML data looks like this:

```
<machine> </machine>
```

It is called an *element* and “machine” is the name of the element. The two parts of the elements are like parentheses that delimit the *content* of an element. When there is no content between the two parts—other than white space—XML allows a shorthand:

```
<machine />
```

But as far as we are concerned here, this shorthand is equivalent to the explicit version.

From an S-expression perspective, an XML element is a **named** pair of parentheses that surround some content. Although we could use the following structure type to represent an XML element

Racket's `xml` library represents XML with structures.

```
(define-struct element [name content])
```

an S-expression representation is even more natural:

```
'(machine)
```

This piece of data has the opening and closing parenthesis, and it comes with space to embed content.

Here is a piece of XML data with content:

```
<machine><action /></machine>
```

Remember that the `<action />` part is a shorthand, meaning we are really looking at this piece of data:

```
<machine><action></action></machine>
```

In general, the content of an XML element is a series of XML elements, e.g.,

```
<machine><action /><action /><action /></machine>
```

Expand the shorthand for `<action />` before you continue.

To represent the first nested XML example in ISL+, we can use the `element` structure type from above:

```
(make-element "machine" (list (make-element "action" '()))))
```

Since we now know that the content of an element is a sequence of XML data, we naturally use lists of (representations of) XML elements for the content field; `'()` signals the lack of content. The S-expression alternative looks much simpler than that:

```
'(machine (action))
```

When you look at the piece of XML data with a sequence of three `<action />` elements as its content, you realize that you may wish to distinguish such elements from each other. To this end, XML elements come with *attributes*. For example,

```
<machine initial="red"></machine>
```

is the “machine” element equipped with one attribute whose *name* is “initial” and whose value is “red” between string quotes. Here is complex XML element with nested elements that have attributes too:

```
<machine initial="red">
  <action state="red" next="green" />
  <action state="green" next="yellow" />
  <action state="yellow" next="red" />
</machine>
```

We use blanks, indentation, and line breaks to make the element readable but this white space has no meaning for our XML data here.

The introduction of attributes requires a modification of the structure representation:

```
| (define-struct element [name attributes content])
```

In addition, a consistently structural representation also calls for an attribute structure type:

```
| (define-struct attribute [name value])
```

Here we represent attributes as instances of this structure type that combine symbols with strings. A sequence of attributes is a list of such instances. Now it is straightforward to get a piece of ISL+ data for our first machine element:

```
| (make-element "machine" (list (make-attribute 'initial "red")) '())
```

The complex machine with nested actions corresponds to a nest of structure instances and lists:

```
(make-element "machine" (list (make-attribute 'initial "red"))
  (list
    (make-element "action"
      (list (make-attribute 'state "red")
        (make-attribute 'next "green")))
      '())
    (make-element "action"
      (list (make-attribute 'state "green")
        (make-attribute 'next "yellow")))
      '())
    (make-element "action"
      (list (make-attribute 'state "yellow")
        (make-attribute 'next "green")))
      '())))
'
```

In contrast, S-expressions for these “machine” elements look much simpler than these two nests of structure instances:

```
| '(machine ((initial "red")))
```

To add attributes to an element, we use a list of lists where each of the latter contains two items: a symbol and a string. The symbol represents the name of the attribute and the string its value. This idea naturally applies to complex forms of XML data, too:

```
'(machine ((initial "red"))
  (action ((state "red") (next "green")))
  (action ((state "green") (next "yellow")))
  (action ((state "yellow") (next "red"))))
```

For now note how the attributes are marked by two opening parentheses and the remaining list of (representations of) XML elements have one opening parenthesis.

You may recall the idea from [Intermezzo: Quote, Unquote](#), which uses S-expressions to represent XHTML, a special dialect of XML. In particular, the intermezzo shows how easily a programmer can write down non-trivial XML data and even templates of XML representations—using backquote and [unquote](#). Of course, the preceding chapter points out that you need a parser to determine whether any given S-expression is a representation of XML data, and a parser is a complex and unusual kind of function.

Nevertheless, we choose to go with a representation of XML based on S-expressions to demonstrate the usefulness of these old ideas in practical terms. Let us proceed gradually to work out a data definition. Here is a first attempt:

```
; An Xexpr.v0 (short for X-expression) is
;   (cons Symbol '())
```

This is the “named parentheses” idea from the beginning of this section. Equipping this element representation with content is easy:

```
; An Xexpr.v1 is
;   (cons Symbol [List-of Xexpr.v1])
```

The symbolic name becomes the first item on a list that otherwise consists of XML element representatives.

The last refinement step is to add attributes. Since the attributes in an XML element are optional, the revised data definition should have two clauses:

```
; An Xexpr.v2 is
; - (cons Symbol [List-of Xexpr.v2])
; - (cons Symbol (cons [List-of Attribute] [List-of Xexpr.v2]))
```

Our sample data representations from above suggest this definition for `Attribute`:

```
; An Attribute is
;   (cons Symbol (cons String '()))
```

The question is whether this data definition is practical for data processing.

Exercise 369. Eliminate the use of `List-of` from the data definition `Xexpr.v2`.

Exercise 370. Represent the following XML data as elements of `Xexpr.v2`:

1. <transition from="seen-e" to="seen-f" />
2. <word /><word /><word />
3. <end></end>

Which one could be represented in `Xexpr.v0` or `Xexpr.v1`?

Exercise 371. Interpret the following elements of `Xexpr.v2` as XML data:

1. '(server ((name "example.org")))
2. '(carcassonne (board (grass)) (player ((name "sam"))))
3. '(start)

Which ones are elements of `Xexpr.v0` or `Xexpr.v1`?

Roughly speaking, X-expressions simulate structures via lists. The simulation is convenient for programmers; it asks for the least amount of keyboard typing. For example, if an X-expression does not come with an attribute list, it is simply omitted. This choice of data representation represents a trade-off between authoring such expressions manually and processing them automatically. The best way to deal with the latter problem is to provide functions that make X-expressions look like structures, especially functions that access the quasi-fields:

- `xexpr-name`, which extracts the tag of the element representation;
- `xexpr-attributes`, which extracts the list of attributes;

- `xexpr-content`, which extracts the list of content elements.

Once we have such functions, we can use lists and act as if they were structures.

These functions parse S-expressions, and parsers are tricky to design. So let us follow the design recipe carefully, starting with some data examples:

```
(define a0 '((initial "red")))

(define e0 '(machine))
(define e1 `(machine ,a0))
(define e2 '(machine (action)))
(define e3 '(machine () (action)))
(define e4 `(machine ,a0 (action) (action)))
```

The first definition introduces a list of attributes, which is reused twice in the construction of X-expressions. The definition of `e0` reminds us that an X-expression may not come with either attributes or content. You should be able to explain why `e2` and `e3` are basically equivalent.

Next we formulate a signature, a purpose statement, and a header:

```
; Xexpr.v2 -> [List-of Attribute]
; retrieves the list of attributes of xe
(define (xexpr-attributes xe) '())
```

Here we focus on `xexpr-attributes`; we leave the other two functions as exercises.

Making up functional examples requires a decision for X-expressions without attributes, e.g., `'(machine)` or `'(machine (action))`. While our chosen representation completely omits missing attributes, we would have to supply `'()` if we represented the equivalent XML data with structures. We therefore decide that `xexpr-attribute` produces `'()` for such X-expressions:

```
(check-expect (xexpr-attributes e0) '())
(check-expect (xexpr-attributes e1) '((initial "red")))
(check-expect (xexpr-attributes e2) '())
(check-expect (xexpr-attributes e3) '())
(check-expect (xexpr-attributes e4) '((initial "red")))
```

Before you read on, make sure you can explain all of the examples.

It is time to develop the template. Since the data definition for `Xexpr.v2` is complex, we proceed slowly, step by step. First, while the data definition distinguishes two kinds of X-expressions, both clauses describe data constructed by `consing` a symbol onto a list. Second, what differentiates the two clauses is the rest of the list and especially the optional presence of a list of attributes. Let us translate these two insights into a template:

```
(define (xexpr-attributes xe)
  (local ((define optional-loa+content (rest xe)))
    (cond
      [(empty? optional-loa+content) ...]
      [else ...])))
```

The local definition chops off the name of the X-expression and leaves the remainder of the list, which may or may not start with a list of attributes. The key is that it is just a list, and the two `cond` clauses indicate so. Third, this list is **not** defined via a self-reference but as

the optional `cons` of some attributes onto a possibly empty list of X-expressions. In other words, we still need to distinguish the two usual cases and extract the usual pieces:

```
(define (xexpr-attributes xe)
  (local ((define optional-loa+content (rest xe)))
    (cond
      [(empty? optional-loa+content) ...]
      [else (... (first optional-loa+content)
                  ... (rest optional-loa+content) ...))]))
```

At this point, we can see that recursion is not needed for the task at hand. So, we switch to the fifth step of the design recipe. Clearly, there are no attributes if the given X-expression comes with nothing but a name. In the second clause, the question is whether the first item on the list is a list of attributes or just an `Xexpr.v2`. Because this sounds complicated, we make a wish:

```
; [List-of Attribute] or Xexpr.v2 -> ???
; determine whether x is an element of [List-of Attribute]; #false otherwise
(define (list-of-attributes? x)
  #false)
```

With this function, it is straightforward to finish `xexpr-attributes`:

```
(define (xexpr-attributes xe)
  (local ((define optional-loa+content (rest xe)))
    (cond
      [(empty? optional-loa+content) '()]
      [else (local ((define loa-or-x (first optional-loa+content)))
              (if (list-of-attributes? loa-or-x)
                  loa-or-x
                  '()))))))
```

If the first item is a list of attributes, the function produces it; otherwise there are no attributes.

For the design of `list-of-attributes?`, we proceed in the same manner and get this definition:

```
; [List-of Attribute] or Xexpr.v2 -> Boolean
; is the given value a list of attributes
(define (list-of-attributes? x)
  (cond
    [(empty? x) #true]
    [else (local ((define possible-attribute (first x)))
            (cons? possible-attribute))]))
```

We skip the details of the design process because they are unremarkable. What is remarkable is the signature of this function. Instead of specifying a single data definition as possible inputs, the signatures combines two data definitions separated with the English word “or.” In ISL+ such an informal signature with a definite meaning is acceptable on occasion; do not use it too often, however.

... and in the currently popular scripting languages

Exercise 372. Design the functions `xexpr-name` and `xexpr-content`.

Exercise 373. The design recipe calls for a self-reference in the template for `xexpr-`

attributes. Add this self-reference to the template and then explain why the finished parsing function does not contain it.

Exercise 374. Formulate a data definition that replaces the informal “or” signature for the definition of the `list-of-attributes?` function.

Exercise 375. Design `lookup-attribute`. The function consumes a list of attributes and a symbol. If the attributes list associates the symbol with a string, the function retrieves this string; otherwise it returns `#false`.—Consider using `assq` to define the function.

For the remainder of this chapter, we assume that `xexpr-name`, `xexpr-attributes`, and `xexpr-content` exist. Additionally, we also use the `lookup-attribute` function from [exercise 375](#). To keep the prose simple, we use `Xexpr` to refer to [Xexpr.v2](#).

25.2 Rendering XML Enumerations

XML is really a family of languages, similar to the teaching languages in DrRacket, and people define dialects for specific channels of communication. For example, XHTML is the language for sending web in XML format. In this section, we illustrate how to design a rendering function for a small snippet of XHTML, specifically the enumerations from the beginning of this section.

The `ul` tag surrounds a so-called unordered HTML list. Each item of this list is tagged with `li`, which tends to contain words but also other elements possibly including enumerations. With “unordered” the authors of HTML express that each item is to be rendered with a leading bullet instead of a number.

Since `Xexpr` does not come with plain strings, it is not immediately obvious how to represent XHTML enumerations in a subset. One option is to refine the data representation one more time, so that an `Xexpr` could be a `String`. Another option is to introduce a representation for text within enumerations:

```
; An XWord is '(word ((text String))).
```

In this section, we use this option; Racket, the language from which the teaching languages are derived, offers libraries that equate `Xexpr` with `String`.

Exercise 376. Make up three examples of `XWords`. Design the functions `word?`, which checks whether any `ISL+` value is in `XWord`, and `word-text`, which extracts the value of the only attribute of an instance of `XWord`.

Exercise 377. Refine the definition of `Xexpr.v2` so that you can represent XML elements that are plain strings. Use this refinement to represent enumerations.

Given the representation of words, representing an XHTML-style enumeration of words is straightforward:

```
; An XEnum.v1 is one of:
; - (cons 'ul [List-of XItem.v1])
; - (cons 'ul (cons [List-of Attribute] [List-of XItem.v1]))
; An XItem.v1 is one of:
; - (cons 'li (cons XWord '()))
; - (cons 'li (cons [List-of Attribute] (cons XWord '())))
```

The data definition includes attribute lists for completeness, but this section does not take attributes into account for rendering enumerations or items.

Stop! Argue that every element of `XEnum.v1` is also in `XExpr`.

Here is a sample element of `XEnum.v1`:

```
(define e0
  '(ul
    (li (word ((text "one"))))
    (li (word ((text "two"))))))
```

It corresponds to the inner enumeration of the example from the beginning of the chapter. Rendering it with help from the `2htdp/image` library should yield an image like this:

The radius of the bullet and the distance between the bullet and the text are matters of aesthetic; here the idea matters.

To create this kind image, you might use this ISL+ program:

We developed these expressions in the interactions area. What would you do?

```
(define item1-rendered
  (beside/align 'center BULLET (text "one" 12 'black)))
(define item2-rendered
  (beside/align 'center BULLET (text "two" 12 'black)))
(define e0-rendered
  (above/align 'left item1-rendered item2-rendered))
```

assuming `BULLET` is a rendering of a bullet.

But let us design the function carefully. Since the data representation requires two data definitions, the design recipe tells you that you must design two functions in parallel. A second look reveals, however, that in this particular case the second data definition is disconnected from the first one, meaning we can deal with it separately.

Furthermore, the definition for `XItem.v1` consists of two clauses, meaning the function itself should consist of a `cond` with two clauses. The point of viewing `XItem.v1` as a sub-language of `Xexpr`, however, is to think of these two clauses in terms of `Xexpr` selector functions, in particular, `xexpr-content`. With this function we can extract the textual part of an item, regardless of whether it comes with attributes or not:

```
; XItem.v1 -> Image
; renders a single item as a "word" prefixed by a bullet
(define (render-item1 i)
  (... (xexpr-content i) ...))
```

In general, `xexpr-content` extracts a list of `Xexpr`; in this specific case, the list contains exactly one `XWord`, and this word contains one text:

```
(define (render-item1 i)
  (local ((define content (xexpr-content i))
          (define element (first content))
          (define word-in-i (word-text element)))
    (... word-in-i ...)))
```

From here, it is straightforward:

```
(define (render-item1 i)
  (local ((define content (xexpr-content i))
          (define element (first content)))
         (define word-in-i (word-text element)))
    (beside/align 'center BULLET (text word-in-i 12 'black))))
```

After extracting the text to be rendered in the item, it is simply a question of rendering it as text and equipping it with a leading bullet; see the examples above for how you might discover this last step.

Exercise 378. Before you read on, equip the definition of `render-item1` with at least one test; make sure that the tests are formulated so that they don't truly depend on the nature of `BULLET`. Then explain **how** the function works; keep in mind that the purpose statement explains only **what** it does.

Now we can focus on the design of a function that renders an enumeration. Using the example from above, we can formulate the result of the first two design steps:

```
; XEnum.v1 -> Image
; renders a simple enumeration as an image

(check-expect (render e0) e0-rendered)

(define (render-enum1 xe)
  empty-image)
```

The key step is the development of a template. According to the data definition, an element of `XEnum.v1` contains one interesting piece of data, namely, the (representation of the) XML elements. The first item is always '`ul`', so there is no need to extract it, and the second, optional item is a list of attributes, which we ignore. With this in mind, the first template draft looks just like the one for `render-item1`:

```
(define (render-enum1 xe)
  (... (xexpr-content xe) ...))
```

The extracted piece of data is an element of [[List-of XItem.v1](#)].

The data-oriented design recipe tells you that you should design a separate function whenever you encounter a complex form of data, such as this list of items. The abstraction-based design recipe from [Abstraction](#) tells you to reuse an existing abstraction, say a list-processing function from [figure 71](#), when possible.

Given that `render-enum1` is supposed to process a list and create a single image from it, the only two list-processing abstractions whose signatures fit the bill are `foldr` and `foldl`. If you also study their purpose statements, you see a pattern that is like the `e0-rendered` example above, especially for `foldr`. Let's try to use it, following the re-use design recipe:

```
(define (render-enum1 xe)
  (local ((define content (xexpr-content xe))
          ; XItem.v1 Image -> Image
          (define (deal-with-one-item fst-item so-far)
            ...))
    (foldr deal-with-one-item empty-image content)))
```

From the type matching, you also know that:

1. the first argument to `foldr` must be a two-argument function, which consumes one

item at a time and the image built up so far;

2. the second argument must be an image;

3. the last argument is the list.

We consider `empty-image` the correct starting point because nothing is known about the image; naturally, the XML content is the list of elements to be processed.

This design-by-reuse focuses our attention on one function definition, the function to be “folded” over the list. It turns one item and the image that `foldr` has created so far into another image. The signature for `deal-with-one-item` articulates this insight. Since the first argument is an instance of `XItem.v1`, `render-item1` is the function that renders it. This yields two images that must be combined: the image of the first item and the image of the rest of the items. Clearly, `deal-with-one-item` must stack them on top of each other, which is precisely what `above` accomplishes:

```
(define (render-enum1 xe)
  (local ((define content (xexpr-content xe))
          ; XItem.v1 Image -> Image
          (define (deal-with-one-item fst-item so-far)
            (above/align 'left (render-item1 fst-item) so-far)))
    (foldr deal-with-one-item empty-image content)))
```

The example suggests the use of `above/align` and `'left`.

Flat enumerations are common but they are also a simple approximation of the full-fledged case. In the real world, web browsers must cope with nested enumerations that arrive over the web, and at least in principle, the nesting can be arbitrarily deep. In XML and its web browser dialect XHTML, nesting is straightforward. Any element may show up as the content of any other element. To represent this relationship in our limited XHTML representation, we say that an item is either a word or another enumeration:

Are you wondering whether arbitrary nesting is the correct way to think about this problem? If so, develop a data definition that allows only three levels of enumeration and use it to design rendering functions.

```
; An XItem.v2 is one of:
; - (cons 'li (cons XWord '()))
; - (cons 'li (cons [List-of Attribute] (cons XWord '())))
; - (cons 'li (cons XEnum.v2 '()))
; - (cons 'li (cons [List-of Attribute] (cons XEnum.v2 '())))
```

We must also revise the data definition for enumerations so that they refer to the correct form of item:

```
; An XEnum.v2 is one of:
; - (cons 'ul [List-of XItem.v2])
; - (cons 'ul (cons [List-of Attribute] [List-of XItem.v2]))
```

The next question is how this change to the data definition affects the rendering functions. Put differently, how do `render-enum1` and `render-item1` have to change so that they can cope with elements of `XEnum.v2` and `XItem.v2`, respectively. Software engineers face these kinds of questions all the time, and it is in this situation where the design recipe shines.

```

(define SIZE 12)
(define COLOR 'black)
(define BULLET
  (beside (circle 1 'solid 'black) (text " " SIZE COLOR)))

; Image -> Image
; marks item with bullet
(define (bulletize item)
  (beside/align 'center BULLET item))

(define e0 ...)
(define e0-rendered ...)

; XEnum.v2 -> Image
; renders an XEnum.v2 as an image

(check-expect (render-enum e0) e0-rendered)

(define (render-enum xe)
  (local ((define content (xexpr-content xe))
          ; XItem.v2 Image -> Image
          (define (deal-with-one-item fst-item so-far)
            (above/align 'left (render-item fst-item) so-far)))
    (foldr deal-with-one-item empty-image content)))

; XItem.v2 -> Image
; renders one XItem.v2 as an image

(check-expect
  (render-item '(li (word ((text "one")))))
  (bulletize (text "one" SIZE COLOR)))

(check-expect (render-item `(li ,e0)) (bulletize e0-rendered))

(define (render-item an-item)
  (local ((define content (first (xexpr-content an-item))))
    (beside/align
      'center BULLET
      (cond
        [(word? content) (text (word-text content) SIZE 'black)]
        [else (render-enum content)]))))
```

Figure 93: Refining functions in response to refinements of data definitions

Figure 93 shows the complete answer. Since the change is confined to the data definitions for `XItem.v2`, it should not come as a surprise that the change to the rendering program shows up in the function for rendering items. While `render-item1` does not need to distinguish between different forms of `XItem.v1`, `render-item` is forced to use a `cond` because `XItem.v2` lists two different kinds of items. Given that this data definition is close to one from the real world, the distinguishing characteristic is not something simple—like `'()` vs `cons`—but a specific piece of the given item. If the item's content is a `Word`, the rendering function proceeds as before. Otherwise, the item contains an enumeration, in which case `render-item` uses `render-enum` to deal with the data, because the data definition for `XItem.v2` refers back to `XEnum.v2` precisely at this point.

Exercise 379. Use the recipe to design the rendering functions for `XEnum.v2` and `XItem.v2` from scratch. You should come up with the same functions.

Exercise 380. The wrapping of `cond` with `(beside/align 'center BULLET ...)` may surprise you. Edit the function definition so that the wrap-around appears once in each clause. Why are you confident that your change works? Which version do you prefer?

Exercise 381. Design a program that counts all occurrences of "hello" in an instance of `XEnum.v2`.

Exercise 382. Design a program that replaces all occurrences of "hello" with "bye" in an enumeration.

25.3 Domain-Specific Languages

Engineers routinely build large software systems that require a configuration for specific contexts before they can be run. This configuration task tends to fall to *systems administrators* who must deal with many different software systems. The word "configuration" refers to the data that the main function needs when the program is launched. In a sense a configuration is just an addition argument, though it is usually so complex that program designers prefer a different mechanism for handing it over.

Since software engineers cannot assume that systems administrators know every programming language, they tend to devise simple, special-purpose configuration languages. These special languages are also known as a *domain-specific languages* (DSL).

Developing these DSLs around a common core, say the well-known XML syntax, simplifies life for systems administrators. They can write small XML "programs" and thus configure the systems they must launch.

Because configurations abstract a program over various pieces of data, Prof. Paul Hudak argued in the 1990s that DSLs are the **ultimate abstractions**, that is, that they generalize the ideas of `Abstraction` to perfection.

While the construction of a DSL is often considered a task for an advanced programmer, you are actually in a position to understand, appreciate, and implement a reasonably complex DSL already. This section explains how it all works. It first re-acquaints you with finite state machines (FSMs). Then it shows how to design, implement, and program a DSL for configuring a system that simulates arbitrary FSMs.

Finite State Machines Remembered The theme of finite state machine is an important one in computing and this book has presented it several times already, starting with [Finite State Worlds](#) and especially [exercise 112](#) through [Finite State Machines](#) and ... [Add Expressive Power](#). Here we reuse the example from the last section as the component for which we wish to design and implement a configuration DSL.

```
; A FSM is a [List-of 1Transition]
; A 1Transition is a list of two items:
;   (cons FSM-State (cons FSM-State '()))
; A FSM-State is a String that specifies color

; data examples
(define fsm-traffic
  '(("red" "green") ("green" "yellow") ("yellow" "red")))

; FSM FSM-State -> FSM-State
; match the keys pressed by a player with the given FSM
(define (simulate state0 transitions)
```

```

; State of the World: FSM-State
(big-bang state0
  [to-draw
    (lambda (current)
      (square 100 "solid" current))]
  [on-key
    (lambda (current key-event)
      (find transitions current)))))

; [List-of [List X Y]] X -> Y
; finds the matching Y for the given X in the association list
(define (find alist x)
  (local ((define fm (assoc x alist)))
    (if (cons? fm) (second fm) (error "next state not found"))))

```

Figure 94: Finite state machines, revisited

For convenience, figure 94 presents the entire code again, though reformulated using just lists and using the full power of ISL+. The program consists of two data definitions, one data example, and two function definitions: `simulate` and `find`. Unlike the related programs in preceding chapters, this one represents a transition as a list of two items: the current state and the next one.

The main function, `simulate`, consumes a transition table and an initial state; it then evaluates a `big-bang` expression, which reacts to each key event with a state transition. The states are displayed as colored squares. The `to-draw` and `on-key` clauses are specified with `lambda` expressions that consume the current state, plus the actual key event, and that produce an image or the next state, respectively.

As its signature shows, the auxiliary `find` function is completely independent of the FSM application. It consumes a list of two-item lists and an item but the actual nature of the items is specified via parameters. In the context of this program, `X` and `Y` represent `FSM-States`, meaning `find` consumes a transition table together with a state and produces a state. The function body uses the built-in `assoc` function to perform most of the work. Look up the documentation for `assoc` so that you understand why the body of `local` uses an `if` expression.

Exercise 383. Modify the rendering function so that it overlays the name of the state onto the colored square.

Exercise 384. Formulate test cases for `find`. Design `find` from scratch.

Exercise 385. Reformulate the data definition for `1Transition` so that it is possible to restrict transitions to certain key strokes. Try to formulate the change so that `find` continues to work without change. What else do you need to change to get the complete program to work? Which part of the design recipe provides the answer(s)? See [exercise 231](#) for the original exercise statement.

Configurations The FSM simulation program requires two arguments, which jointly describe a machine. Rather than teach a potential “customer” how to open a ISL+ program in DrRacket and launch a function of two arguments, the “seller” of `simulate` may wish to supplement this product with a configuration component.

A configuration component consists of two parts. The first one is a widely used simple language that customers use to formulate the initial arguments for a component’s main function(s). The second one is a function that translates what customers say into a

function call for the main function. For the FSM simulator, we must agree on how we represent finite state machines in XML. By judicious planning, [XML as S-expressions](#) presents a series of machine examples that look just right for the task. Recall the final machine example in this section:

```
<machine initial="red">
  <action state="red"    next="green" />
  <action state="green"  next="yellow" />
  <action state="yellow" next="red" />
</machine>
```

Compare it to the transition table `fsm-traffic` from [figure 94](#). Also recall the agreed-upon [Xexpr](#) representation of this example:

```
(define xm0
  '(machine ((initial "red"))
            (action ((state "red") (next "green"))))
            (action ((state "green") (next "yellow"))))
            (action ((state "yellow") (next "red")))))
```

What we are still lacking is a general data definition that describes all possible [Xexpr](#) representations of finite state machines:

```
; An XMachine is:
;   (list 'machine (list (list 'initial FSM-State)) [List-of X1T])
; An X1T is
;   (list 'action (list (list 'state FSM-State) (list 'next FSM-State)))
```

Like [XEnum.v2](#), [XMachine](#) describes a subset of all [Xexpr](#). Thus, when we design functions that process this new form of data, we may continue to use the generic [Xexpr](#) functions to access pieces.

Exercise 386. Formulate an XML configuration for a machine that switches from white to black and back for every key event and then translate it into an [XMachine](#) representation. See [exercise 229](#) for an implementation of the machine as a program.

Before we dive into the translation part of the configuration problem, let's spell it out:

Sample Problem: Design a program that uses an [XMachine](#) configuration to run `simulate`.

While this problem is specific to our case, it is easy to imagine a generalization for similar systems and we encourage you to do so.

The problem statement implies a signature, a purpose statement, and a header:

```
; XMachine -> FSM-State
; simulates an FSM via the given configuration
(define (simulate-xmachine xm)
  (simulate ... ...))
```

Indeed, it almost dictates the complete function definition. Following the problem statement, our function calls `simulate` with two to-be-determined arguments. What we need to complete the definition are two pieces: an initial state and a transition table. These two pieces are part of `xm` and we are best off wishing for appropriate functions:

- `xm-state0` extracts the initial state from the given [XMachine](#):

- (check-expect (xm-state0 xm0) "red")
- xm->transitions translates the embedded [List-of X1T] into [List-of 1Transition]:
- (check-expect (xm->transitions xm0) fsm-traffic)

```

; XMachine -> FSM-State
; interprets the given configuration as a state machine
(define (simulate-xmachine xm)
  (simulate (xm-state0 xm) (xm->transitions xm)))

; XMachine -> FSM-State
; extracts and translates the transition table from a configuration

(check-expect (xm-state0 xm0) "red")

(define (xm-state0 xm0)
  (lookup-attribute (xexpr-attributes xm0) 'initial))

; XMachine -> [List-of 1Transition]
; extracts the transition table from an XML configuration

(check-expect (xm->transitions xm0) fsm-traffic)

(define (xm->transitions xm)
  (local (; X1T -> 1Transition
    (define (xaction->action xa)
      (list (lookup-attribute (xexpr-attributes xa) 'state)
            (lookup-attribute (xexpr-attributes xa) 'next)))
    (map xaction->action (xexpr-content xm))))
```

Figure 95: Interpreting a DSL program

Since **XMachine** is a subset of **Xexpr**, defining **xm-state0** is straightforward. Given that the initial state is specified as an attribute, **xm-state0** extracts the list of attributes using **xexpr-attributes** and then retrieves the value of the '**initial**' attribute.

Let us then turn to **xm->transitions**, which translates the transitions inside of an **XMachine** configuration into a transition table:

```

; XMachine -> [List-of 1Transition]
; extracts and translates the transition table from a configuration
(define (xm->transitions xm)
  '())
```

The name of the function prescribes the signature and suggests a purpose statement. Our purpose statement describes a two-step process: (1) extract the **Xexpr** representation of the transitions and (2) translate them into an instance of [List-of 1Transition].

While the extraction part obviously uses **xexpr-content** to get the list, the translation part calls for some more analysis. If you look back to the data definition of **XMachine**, you see that the content of the **Xexpr** is a list of **X1T**s. The signature tells us that the transition table is a list of **1Transitions**. Indeed, it is quite obvious that each item in the former list is translated into one item of the latter, which suggests we used **map** to define the function:

```

(define (xm->transitions xm)
  (local (; X1T -> 1Transition
```

```
; translates an Xexpr transition into a list
(define (xaction->action xa)
  ...))
(map xaction->action (xexpr-content xm)))
```

As you can see, we follow the design ideas of [Using Abstractions, by Example](#) and formulate the function as a [local](#) whose body uses `map`. Defining `xaction->action` is again just a matter of extracting the appropriate values from an [Xexpr](#).

[Figure 95](#) displays the complete definitions for all three functions: `simulate-xmachine`, `xm-state0`, and `xm->transitions`. In this case, the translation from the DSL to a proper function call is as large as the original component. This is not the case for real-world systems; the DSL component tends to be a small fraction of the overall product, which is why the approach is so popular.

Exercise 387. Run the code in [figure 95](#) with the BW Machine configuration from [exercise 386](#).

25.4 Reading XML

Systems administrators expect that systems configuration systems read configuration programs from a file or possibly from some place on the web. In ISL+ your programs can retrieve (some) XML information from files and the web with the help of the `2htdp/batch-io` library. [Figure 96](#) shows the relevant excerpt from teachpack.

This section requires the `2htdp/batch-io` library, the `2htdp/universe` library, and the `2htdp/image` library.

The figure uses the suffix `.v3` for consistency, including those data definitions for which there is no version 2.

```
; Xexpr.v3 is one of:
; - Symbol
; - String
; - Number
; - (cons Symbol (cons Attribute*.v3 [List-of Xexpr.v3]))
; - (cons Symbol [List-of Xexpr.v3])
;
; Attribute*.v3 is [List-of Attribute.v3]
;
; Attribute.v3 is:
;   (list Symbol String)
; interpretation.: (list 'a "some text") represents a="some text"
;
; Any -> Boolean
; is the given value an Xexpr.v3
; effect display bad piece if x is not an Xexpr.v3
(define (xexpr? x) ...)

; String -> Xexpr.v3
; produces the first XML element in file f as an X-expression
(define (read-xexpr f) ...)
```

```

; String -> Xexpr.v3
; produces the first XML element in file f as an X-expression
; and all whitespace between embedded elements is eliminated
; assume the XML element may not contain any text as elements
(define (read-plain-xexpr f) ...)

; String -> Boolean
; #false, if this url returns a '404'; #true otherwise
(define (url-exists? u) ...)

; String -> [Maybe Xexpr.v3]
; produces the first XML element from URL u as an X-expression
; or #false if (not (url-exists? u))
; reads HTML as XML if possible
; effect signals an error in case of network problems
(define (read-xexpr/web u) ...)

; String -> [Maybe Xexpr.v3]
; produces the first XML element from URL u as an X-expression
; and all whitespace between embedded elements is eliminated
; or #false if (not (url-exists? u))
; reads HTML as XML if possible
; effect signals an error in case of network problems
(define (read-plain-xexpr/web u) ...)

; Xexpr.v3 -> String
; renders the X-expression x as a string
(define (xexpr-as-string x) ...)

```

Figure 96: Reading X-expressions

For illustrative purposes, assume we have a file calle

v.6.3.0.2

Intermezzo: The Nature of Numbers

When it comes to numbers, programming languages mediate the gap between the underlying hardware and true mathematics. The typical computer hardware represents numbers with fixed-size chunks of data; they also come with processors that work on just such chunks. In paper-and-pencil calculations, we don't worry about how many digits we process; in principle, we can deal with numbers

These chunks are called; *bits, bytes, and words*.

that consist of one digit, 10 digits, or 10,000 digits. Thus, if a programming language uses the numbers from the underlying hardware, its calculations are as efficient as possible. If it sticks to the numbers we know from mathematics, it must translate those into chunks of hardware data and back—and these translations cost time. Because of this cost, most creators of programming languages adopt the hardware-based choice.

This intermezzo explains the hardware representation of numbers as an exercise in data representation. Concretely, the first subsection introduces a concrete fixed-size data representation for numbers, discusses how to map numbers into this representation, and hints at how calculations work on such numbers. The second and third section illustrate the two most fundamental problems of this choice: arithmetic overflow and underflow, respectively. The last one sketches how arithmetic in the teaching languages works; its number system **generalizes** what you find in most of today's programming languages. The final exercises show how bad things can get when programs compute with numbers.

Fixed-size Number Arithmetic

Suppose we can use four digits to represent numbers. If we represent natural numbers, one representable range is [0, 10000). For real numbers, we could pick 10,000 fractions between 0 and 1 or 5,000 between 0 and 1 and another 5,000 between 1 and 2, etc. In either case, four digits can represent at most 10,000 numbers for some chosen interval, while the number line for this interval contains an infinite number of numbers.

The common choice for hardware numbers is to use so-called scientific notation, meaning numbers are represented with two parts:

1. a *mantissa*, which is a base number, and
2. an *exponent*, which is used to determine a 10-based factor.

For pure scientific notation, the base is between 0 and 9; we ignore this constraint.

Expressed as a formula, we write numbers as

$$m \cdot 10^e$$

where m is the mantissa and e the exponent. For example, one representation of 1200 with this scheme is

$$120 \cdot 10^1,$$

another one is

$$12 \cdot 10^2.$$

In general, a number has several equivalents in mantissa-exponent representation.

We can also use negative exponents, which adds fractions at the cost of one extra piece of data: the sign of the exponent. For example,

$$1 \cdot 10^{-2}$$

stands for

$$\frac{1}{100}.$$

To use a form of mantissa-exponent notation for our problem, we must decide how many of the four digits we wish to use for the representation of the mantissa and how many for the exponent. Here we use two for each plus a sign for the exponent; other choices are possible. Given this decision, we can still represent 0 as

$$0 \cdot 10^0.$$

The maximal number we can represent is

$$99 \cdot 10^{99},$$

which is 99 followed by 99 0's. Using the negative exponents, we can add fractions all the way down to

$$01 \cdot 10^{-99},$$

which is the smallest representable number. In sum, using scientific notation with four digits (and a sign), we can represent a vast range of numbers and fractions, but this improvement comes with its own problems.

```
; N Number N -> Inex
; make an instance of Inex after checking the arguments
(define (create-inex m s e)
  (cond
    [(and (<= 0 m 99) (<= 0 e 99) (or (= s 1) (= s -1)))
     (make-inex m s e)]
    [else
     (error 'inex "(<= 0 m 99), s in {+1,-1}, (<= 0 e 99) expected")])))

; Inex -> Number
; convert an inex into its numeric equivalent
(define (inex->number an-inex)
  (* (inex-mantissa an-inex)
     (expt 10 (* (inex-sign an-inex) (inex-exponent an-inex)))))
```

Figure 105: Functions for inexact representations

To understand the problems, it is best to make these choices concrete with a data representation in ISL+ and by running some experiments. Let's represent a fixed-size number with a structure that has three fields:

```
(define-struct inex [mantissa sign exponent])
; An Inex is a structure:
;   (make-inex N99 S N99)
; An S is either 1 or -1
; An N99 is an N between 0 and 99 inclusive
```

Because the conditions on the fields of an **Inex** are so stringent, we define the function `create-inex` to instantiate this structure type definition; see [figure 105](#). The figure also defines `inex->number`, which turns **Inexes** into numbers using the above formula.

Let's translate the above example, [1200](#), into our data representation:

```
(create-inex 12 1 2)
```

Representing [1200](#) as $120 \cdot 10^1$ is illegal, however, according to our data definition:

```
> (create-inex 120 1 1)
inex: (<= 0 m 99), s in {+1, -1}, (<= 0 e 99) expected
```

As the error message says, the arguments don't satisfy the stated data contract. For other numbers, though, we can find two **Inex** equivalents. One example is [5e-19](#):

```
> (create-inex 50 -1 20)
(inex 50 -1 20)
> (create-inex 5 -1 19)
(inex 5 -1 19)
```

Confirm the equivalence of these two representations with `inex->number`.

Using `create-inex` it's also easy to delimit the range of representable numbers:

```
(define MAX-POSITIVE (create-inex 99 1 99))
(define MIN-POSITIVE (create-inex 1 -1 99))
```

The question is which of the real numbers in the range between 0 and `MAX-POSITIVE` can be translated into an **Inex**. In particular, any positive number less than

$$10^{-99}$$

has no equivalent **Inex**. Similarly, the representation has gaps in the middle. For example, the immediate successor of

```
(create-inex 12 1 2)
```

is

```
(create-inex 13 1 2)
```

The first **Inex** represents 1200, the second one 1300. Numbers in the middle, say 1240, can be represented as one or the other—no other **Inex** makes sense. The standard choice is to round the number to the closest representable equivalent, and that is what computer scientists mean with *inexact numbers*. That is, the chosen data representation forces us to map mathematical numbers to approximations.

Finally, we must also consider arithmetic operations on **Inex** structures. Adding two **Inex** representations with the same exponent means adding the two mantissas:

```
(inex+ (create-inex 1 1 0) (create-inex 2 1 0))
==
(create-inex 3 1 0)
```

Translated into mathematical notation, we have

$$\begin{array}{r} 1 \cdot 10^0 \\ + 2 \cdot 10^0 \\ \hline 3 \cdot 10^0 \end{array}$$

When the addition of two mantissas yields too many digits, we have to use the closest neighbor in [Inex](#). Consider adding $55 \cdot 10^0$ to itself. Mathematically we get

$$110 \cdot 10^0,$$

but we can't just translate this number naively into our chosen representation because $110 > 99$. The proper corrective action is to represent the result as

$$11 \cdot 10^1.$$

Or, translated into ISL+, we must ensure that `inex+` computes as follows:

```
(inex+ (create-inex 55 1 0) (create-inex 55 1 0))
==
(create-inex 11 1 1)
```

More generally, if the mantissa of the result is too large, we must divide it by 10 and increase the exponent by one.

Sometimes the result contains more mantissa digits than we can represent. In those cases, `inex+` must round to the closest equivalent in the [Inex](#) world. For example:

```
(inex+ (create-inex 56 1 0) (create-inex 56 1 0))
==
(create-inex 11 1 1)
```

For comparison, here is the precise calculation:

$$56 \cdot 10^0 + 56 \cdot 10^0 = (56 + 56) \cdot 10^0 = 112 \cdot 10^0$$

Because the result has too many mantissa digits, the integer division of the result mantissa by 10 produces an approximate result:

$$11 \cdot 10^1.$$

This is an example of the many approximations that make arithmetic on [Inex](#) inexact.

We can also multiply numbers represented as [Inex](#) structures. Recall that

$$\begin{aligned} & (a \cdot 10^n) \cdot (b \cdot 10^m) \\ &= (a \cdot b) \cdot 10^{n+m} \\ &= (a \cdot b) \cdot 10^{(n+m)} \end{aligned}$$

Thus we get:

$$2 \cdot 10^{+4} \cdot 8 \cdot 10^{+10} = 16 \cdot 10^{+14}$$

or, in ISL+ notation:

```
(inex* (create-inex 2 1 4) (create-inex 8 1 10))
==
(create-inex 16 1 14)
```

As with addition, things are not always straightforward. When the result has too many significant digits in the mantissa, `inex*` has to increase the exponent:

```
(inex* (create-inex 20 1 1) (create-inex 5 1 4))
==
(create-inex 10 1 6)
```

And just like `inex+`, `inex*` introduces an approximation if the true mantissa doesn't have an exact equivalent in [Inex](#):

```
(inex* (create-inex 27 -1 1) (create-inex 7 1 4))
==
(create-inex 19 1 4)
```

Stop! Explain this ISL+ equation with a calculation.

Exercise 415. Design `inex+`. The function adds two `Inex` representations of numbers that have the same exponent. The function must be able to deal with inputs that increase the exponent. Furthermore, it must signal its own error if the result is out of range, not rely on `create-inex` for error checking.

Challenge Extend `inex+` so that it can deal with inputs whose exponents differ by 1:

```
(equal? (inex+ (create-inex 1 1 0) (create-inex 1 -1 1))
       (create-inex 11 -1 1))
```

Do not attempt to deal with larger classes of inputs than that without reading the following subsection. ■

Exercise 416. Design `inex*`. The function multiplies two `Inex` representations of numbers, including inputs that force an additional increase of the output's exponent. Like `inex+`, it must signal its own error if the result is out of range, not rely on `create-inex` to perform error checking. ■

Exercise 417. As this section illustrates, gaps in the data representation lead to round-off errors when numbers are mapped to `Inex`s. The problem is, such round-off errors accumulate across computations.

Design `add`, a function that adds up `n` copies of `#i1/185`. Use `0` and `1` for your examples; for the latter, use a tolerance of `0.0001`. What is the result for `(add 185)`? What would you expect? What happens if you multiply the result with a large number?

Design `sub`. The function counts how often `1/185` can be subtracted from the argument until it is `0`. Use `0` and `1/185` for your examples. What are the expected results? What are the results for `(sub 1)` and `(sub #i1.0)`? What happens in the second case? Why? ■

Overflow

While the use of scientific notation expands the range of numbers we can represent with fixed-size chunks of data, it still doesn't cover arbitrarily large numbers. Some numbers are just too big to fit into a fixed-size number representation. For example,

$$99 \cdot 10^{500}$$

can't be represented, because the exponent 500 won't fit into two digits, and the mantissa is as large as legally possible.

Numbers that are too large for `Inex` can arise during a computation. For example, two numbers that we can represent can add up to a number that we cannot represent:

```
(inex+ (create-inex 50 1 99) (create-inex 50 1 99))
==
(create-inex 100 1 99)
```

which violates the data definition. When the result of `Inex` arithmetic produces numbers that are too large to be represented, we say (arithmetic) *overflow* occurred.

When overflow takes place, some language implementations signal an error and stop the computation. Others designate some symbolic value, called *infinity*, to represent such numbers and propagate it through arithmetic operations.

Note If `Inexes` had a sign field for the mantissa, then two negative numbers can add up to one that is so negative that it can't be represented either. This is called *overflow in the negative direction*. **End**

Exercise 418. ISL+ uses `+inf.0` to deal with overflow. Determine the integer n such that `(expt #i10.0 n)` is an inexact number while `(expt #i10. (+ n 1))` is approximated with `+inf.0`. **Hint** Design a function to compute n . ■

Underflow

At the opposite end of the spectrum, there are small numbers that don't have a representation in `Inex`. For example, 10^{-500} is not 0, but it's smaller than the smallest non-zero number we can represent. An (arithmetic) *underflow* arises when we multiply two small numbers and the result is too small for `Inex`:

```
(inex* (create-inex 1 -1 10) (create-inex 1 -1 99))
==
(create-inex 1 -1 109)
```

which signals an error.

When underflow occurs, some language implementations signal an error; others use 0 to approximate the result. Using 0 to approximate underflow is qualitatively different from picking an approximate representation of a number in `Inex`. Concretely, approximating 1250 with `(create-inex 12 1 2)` drops significant digits from the mantissa, but the result is always within 10% of the number to be represented. Approximating on underflow, however, means dropping the entire mantissa, meaning the result is not within a predictable percentage range of the true result.

Exercise 419. ISL+ uses `#i0.0` to approximate underflow. Determine the smallest integer n such that `(expt #i10.0 n)` is still an inexact ISL+ number and `(expt #i10. (- n 1))` is approximated with 0. **Hint** Use a function to compute n . Consider abstracting over this function and the solution of exercise 418. ■

*SL Numbers

Most programming languages support only approximate number representations and arithmetic for numbers. A typical language limits its integers to an interval that is related to the size of the chunks of the hardware on which it runs. Its representation of real numbers is loosely based on the sketch in the preceding sections, though with larger chunks than the four digits `Inex` uses and using digits from the 2-based number system, because that is how computers work.

In the context of such languages, inexact real representations come in several flavors: *float*, *double*, *extflonum*, etc. It is beyond a first course on programming to explain all possibilities.

The teaching languages support both exact and inexact numbers. Their integers and

rationals are arbitrarily large and precise, limited only by the absolute size of the computer's entire memory. For calculations on these numbers, our teaching languages use the underlying hardware as long as the involved rationals fit into the supported chunks of data; it automatically switches to a different representation and to different version of the arithmetic operations for numbers outside of this interval. Their real numbers come in two flavors: exact and inexact. An exact number truly represents a real number; an inexact one approximates a real number in the spirit of the preceding sections. Arithmetic operations preserve exactness when possible; they produce an inexact result when necessary. Thus, `sqr`t returns an inexact number for both the exact and inexact representation of `2`. In contrast, `sqr`t produces an exact `2` when given exact `4` and `#i2.0` for an input of `#i4.0`. Finally, a numeric constant in a teaching program is understood as an exact rational, unless it is prefixed with `#i`.

Plain Racket interprets all decimal numbers as inexact numbers; it also renders all real numbers as decimals, regardless of whether they are exact or inexact. The implication is that all such numbers are dangerous, because they are likely to be inexact approximations of the true number. A programmer can force Racket to interpret numbers with a dot as exact by prefixing numerical constants with `#e`.

At this point, it is natural to ask how much a program's results may differ from the true results if it uses these inexact numbers. This question is one that early computer scientists struggled with a lot, and over the past few decades, these studies have created a separate field, called *numerical analysis*. Every computer scientist, and indeed, every person who uses computers and software, ought to be aware of its existence and some of its basic insights into the workings of numeric programs. As a first taste, the following exercises illustrate how bad things can get, and you should work through them so that you never lose track of the problems of inexact numbers.

For an accessible introduction—in Racket—see [the article on error analysis](#) by Drs. Neil Toronto and Jay McCarthy or watch [Dr. Toronto's lecture](#) on this topic.

Exercise 420. Evaluate

(`expt 1.001 1e-12`)

in Racket and in ISL+. Explain what you see. □

Exercise 421. Design `my-expt` without using `expt`. The function raises the first given number to the power of the second one, a natural number. Using this function, conduct the following experiment. Add

(`define inex (+ 1 #i1e-12))`
 (`define exac (+ 1 1e-12))`

to the definitions area. What is `(my-expt inex 30)`? How about `(my-expt exac 30)`? Which answer is more useful? □

Exercise 422. When you add two inexact numbers of vastly different orders of magnitude, you may get the larger one back as the result. For example, if a number system uses only 15 significant digits, then we run into problems when adding numbers that vary by more than a factor of 10^{16} :

$$1.0 \cdot 10^{16} + 1 = 1.0000000000000001 \cdot 10^{16},$$

but the closest representable answer is 10^{16} .

At first glance, this approximation doesn't look too bad. Being wrong by one part in 10^{16} (ten million billion) is close enough to the truth. Unfortunately, this kind of problem can add up to huge problems. Consider the following list of numbers:

```
(define JANUS
  (list 31.0
        #i2e+34
        #i-1.2345678901235e+80
        2749.0
        -2939234.0
        #i-2e+33
        #i3.2e+270
        17.0
        #i-2.4e+270
        #i4.2344294738446e+170
        1.0
        #i-8e+269
        0.0
        99.0))
```

Now determine the values of the following expressions:

- (`sum JANUS`)
- (`sum (reverse JANUS)`)
- (`sum (sort JANUS <)`)

Assuming `sum` adds the numbers in a list from left to right, explain what these expressions compute. What do you think of the results?

If you search on the world wide web concerning calculations with inexact numbers (floats), you may find advice that says start with the smallest numbers because adding a big number to two small numbers might yield the former, but adding a big number to the `sum` of two small ones might change the outcome:

```
> (expt 2 #i53.0)
#i9007199254740992.0
> (sum (list #i1.0 (expt 2 #i53.0)))
#i9007199254740992.0
> (sum (list #i1.0 #i1.0 (expt 2 #i53.0)))
#i9007199254740994.0
```

Unfortunately, the third of the above `sum` expressions shows that this advice does **not** work. Explain why.

In a language such as ISL+, it does work to convert all the numbers to exact rationals, use exact arithmetic on the resulting list, and convert the sum of the list back to an inexact number:

```
(exact->inexact (sum (map inexact->exact JANUS)))
```

Evaluate this expression and compare the result to the three sums above. What do you think now about advice from the web? ■

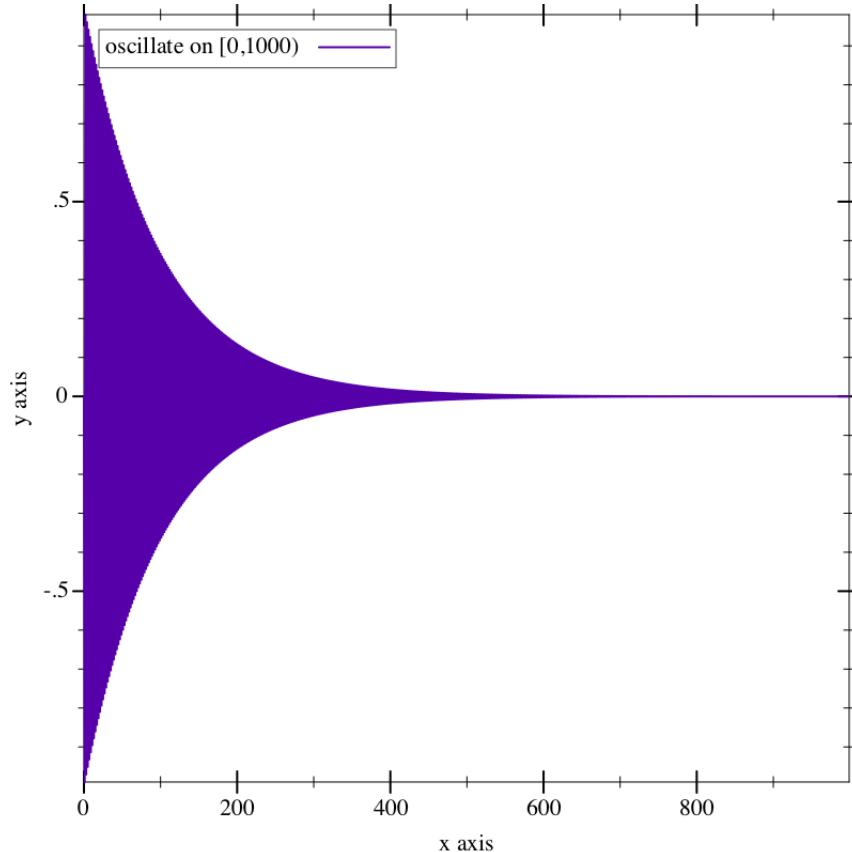
Exercise 423. JANUS looks artificial but take a look at this function definition:

```
(define (oscillate n)
```

```
(local ((define (oscillate i)
  (cond
    [(> i n) '()]
    [else (cons (expt -0.99 i) (oscillate (add1 i))))]))
  (oscillate 1)))
```

Applying oscillate to a natural number n produces the first n elements of a mathematical series. The series rapidly oscillate between -1 and $+1$:

```
> (oscillate 15)
'(#i-0.99
 #i0.9801
 #i-0.970299
 #i0.96059601
 #i-0.9509900498999999
 #i0.941480149401
 #i-0.9320653479069899
 #i0.9227446944279201
 #i-0.9135172474836408
 #i0.9043820750088044
 #i-0.8953382542587164
 #i0.8863848717161292
 #i-0.8775210229989678
 #i0.8687458127689782
 #i-0.8600583546412884)
```



Summing it from left to right computes a different result than from right to left:

```
> (sum (oscillate #i1000.0))
#i-0.49746596003269394
> (sum (reverse (oscillate #i1000.0)))
#i-0.4974659600326953
```

Again, the difference may appear to be small until we see the context:

```
> (- (* 1e+16 (sum (oscillate #i1000.0)))
      (* 1e+16 (sum (reverse (oscillate #i1000.0)))))
#i14.0
```

Explain the difference. Can this difference matter? Can we trust computers? ■

The question is which numbers programmers should use in their programs if they are given a choice. The answer depends on the context of course. In the context of a financial statement, numerical constants should be interpreted as exact numbers, and computational manipulations of financial statements ought to be able to rely on the exactness-preserving nature of mathematical operations. After all, the law cannot accommodate the serious errors that come with inexact numbers and their operations. In the context of scientific computations, however, the extra time needed to produce exact results might impose too much of a burden. Scientists therefore tend to use inexact numbers but carefully analyze their programs to make sure that the numerical errors are

tolerable for their uses of the outputs of programs.

v.6.3.0.2

V Generative Recursion

If you follow the design recipe of the first four parts, you either turn domain knowledge into code or you exploit the structure of the data definition to organize your code. The latter functions typically decompose their arguments into their immediate structural components and then process those components. If one of these immediate components belongs to the same class of data as the input, the function is *structurally recursive*. While structurally designed functions make up the vast majority of code in the real world, some problems cannot be solved with a structural approach to design.

Some functions merely compose such functions; we group those functions also with the “structural” group.

To solve such complicated problems, programmers use *generative recursion*, a form of recursion that is strictly more powerful than structural recursion. The study of generative recursion is as old as mathematics and is often called the study of *algorithms*. The inputs of an algorithm represent a problem. An algorithm tends to re-arrange a problem into a set of several problems, solve those, and combine their solutions into one overall solution. Often some of these newly **generated** problems are the same kind of problem as the given one, in which case the algorithm can be re-used to solve them. In these cases, the algorithm is recursive but its recursion uses newly generated data not immediate parts of the input data.

From the very description of generative recursion, you can tell that designing a generative recursive function is more of an ad hoc activity than designing a structurally recursive function. Still, many elements of the general design recipe apply to the design of algorithms, too, and this part of the book illustrates how and how much the design recipe helps. The key to designing algorithms is the “generation” step, which often means dividing up the problem. And figuring out a novel way of dividing a problem requires insight. Sometimes very little insight is required. For example, it might just require a bit of common-sense knowledge about breaking up sequences of letters. At other times, it may rely on deep mathematical theorems about numbers. In practice, programmers design simple algorithms on their own and rely on domain specialists for their complex brethren. For either kind, programmers must thoroughly understand the underlying ideas so that they can code up algorithms and have the program communicate with future readers. The best way to get acquainted with the idea is to study a wide range of examples and to develop a sense for the kinds of generative recursions that may show up in the real world.

Greeks call it a “eureka.”

29 Non-standard Recursion

At this point you have designed numerous functions that employ structural recursion. When you design a function, you know you need to look at the data definition for its major input. If this input is described by a self-referential data definition, you end up with a function that refers to itself basically where the data definition refers to itself.

This chapter presents two sample programs that use recursion differently. They are

illustrative of the problems that require some eureka, ranging from the obvious idea to the sophisticated insight.

29.1 Recursion without Structure

Imagine you have joined the DrRacket team. The team is working on a sharing service to support collaborations among programmers. Concretely, the next revision of DrRacket is going to enable ISL programmers to share the content of their DrRacket's definition area across several computers. Each time one programmer modifies the buffer, the revised DrRacket broadcasts the content of the definitions area to the instances of DrRacket that participate in the sharing session.

Sample Problem: Your task is to design the function `bundle`, which prepares the content of the definitions area for broadcasting. DrRacket hands over the content as a list of `1Strings`. The function's task is to bundle up sub-sequences of individual "letters" into chunks and to thus produce a list of strings—called **chunks**—of a given length, called **chunk size**.

As you can see, the problem basically spells out the signature and there is no need for any problem-specific data definition:

```
; [List-of 1String] N -> [List-of String]
; bundles sub-sequences of s into strings of length n
(define (bundle s n)
  '())
```

The purpose statement reformulates a sentence fragment from the problem statement and uses the parameters from the dummy function header.

The third step calls for function examples. Here is a list of `1Strings`:

```
(list "a" "b" "c" "d" "e" "f" "g" "h")
```

If we tell `bundle` to bundle this list into pairs—that is, $n = 2$ —then the following list is the expected result:

```
(list "ab" "cd" "ef" "gh")
```

Now if n is 3 instead, there is a left-over "letter". Since the problem statement does not tell us which of the characters is left over, we can imagine at least two valid scenarios:

- The function produces `(list "abc" "def" "g")`, that is, it considers the last letter as the left-over one.
- Or it produces `(list "a" "bcd" "efg")`, which packs the lead character into a string by itself.

Stop! Come up with at least one other choice.

To make things simple, we pick the first choice as the desired result and say so by writing down a corresponding example formulated as a test:

```
(check-expect (bundle (explode "abcdefg") 3) (list "abc" "def" "g"))
```

To keep things concise, this test uses `explode`.

Examples and tests must also describe what happens at the boundary of data definitions. In this context, boundary clearly means `bundle` is given a list that is too short for the given

chunk size:

```
(check-expect (bundle '("a" "b") 3) (list "ab"))
```

It also means we should consider what happens when bundle is given '(). For simplicity, we choose '() as the desired result:

```
(check-expect (bundle '() 3) '())
```

One natural alternative is to ask for '(""). Can you see others?

```
; N as the driving data definition, s considered atomic
; according to Processing Two Lists Simultaneously: Case 1
(define (bundle s n)
  (cond
    [(zero? n) (...)]
    [else (... s ... n ... (bundle s (sub1 n))))])

; [List-of 1String] as the driving data definition, n considered atomic
; according to Processing Two Lists Simultaneously: Case 1
(define (bundle s n)
  ; s as the driving data definition
  (cond
    [(empty? s) (...)]
    [else (... s ... n ... (bundle (rest s) n))]))

; [List-of 1String] and N are on equal footing
; according to Processing Two Lists Simultaneously: Case 2
(define (bundle s n)
  ; lock step
  (cond
    [(and (empty? s) (zero? n)) (...)]
    [else (... s ... n ... (bundle (rest s) (sub1 n))))]))

; the cross-product of possibilities,
; according to Processing Two Lists Simultaneously: Case 3
(define (bundle s n)
  (cond
    [(and (empty? s) (zero? n)) (...)]
    [(and (cons? s) (zero? n)) (...)]
    [(and (empty? s) (positive? n)) (...)]
    [else (... s ... n ...
              (bundle (rest s) (sub1 n)) ...
              (bundle s (sub1 n)) ...
              (bundle (rest s) n) ...)]))
```

Figure 106: A useless template for breaking up strings into chunks

The template step reveals that a structural approach cannot work. Figure 106 shows **four** possible templates given that both arguments to bundle are complex arguments. The first two consider one of the arguments atomic, but that clearly cannot be the case because the function has to understand each argument. The third template is based on the assumption that the two arguments are processed in lock step, which is close—except that bundle clearly has to reset the chunk size to its original value at regular intervals. The final template says that the two arguments are independent and must be processed in this manner, meaning there are four possibilities to proceed at each stage. But this case split decouples the arguments too much because the list and the counting number must be

processed together. In short, we are forced to admit that the structural templates appear to be useless for this design problem.

```

; [List-of 1String] N -> [List-of String]
; bundles sub-sequences of s into strings of length n
; idea take and drop n items at a time

(check-expect (bundle (explode "abcdefg") 3) (list "abc" "def" "g"))
(check-expect (bundle (explode "ab") 3) (list "ab"))
(check-expect (bundle '() 3) '())

(define (bundle s n)
  (cond
    [(empty? s) '()]
    [else (cons (implode (take s n)) (bundle (drop s n) n))]))

; [List-of X] N -> [List-of X]
; retrieves the first n items in l if possible or everything
(define (take l n)
  (cond
    [(zero? n) '()]
    [(empty? l) '()]
    [else (cons (first l) (take (rest l) (sub1 n))))]))

; [List-of X] N -> [List-of X]
; remove the first n items from l if possible or everything
(define (drop l n)
  (cond
    [(zero? n) l]
    [(empty? l) l]
    [else (drop (rest l) (sub1 n))]))

```

Figure 107: Generative recursion

Figure 107 shows a complete definition for `bundle`. The definition uses the `drop` and `take` functions requested in [exercise 398](#); these functions are also available in standard libraries. For completeness, the figure comes with their definitions: `drop` eliminates up to `n` items from the front of the list, `take` returns up to that many items. Using these functions, it is quite straightforward to define `bundle`:

1. if the given list is `'()`, the result is `'()` as decided upon;
2. otherwise `bundle` uses `take` to grab the first `n` `1String`s from `s` and `implodes` them into a plain `String`;
3. it then recurs with a list that is shortened by `n` items, which is accomplished with `drop`;
4. finally, `cons` combines the string from 2 with the list of strings from 3 to create the result for the complete list.

Bullet 3 highlights the key difference between `bundle` and any function in the first four parts of this book. Because the definition of `List-of conses` an item onto a list to create another one, all functions in the first four parts use `first` and `rest` to deconstruct a non-empty list. In contrast, `bundle` uses `drop`, which removes not just one but `n` items at once.

While the definition of `bundle` is unusual, the underlying ideas are intuitive and not too different from the functions seen so far. Indeed, if the chunk size `n` is 1, `bundle` specializes

to a structurally recursive definition. Also, `drop` is guaranteed to produce an integral part of the given list, not some arbitrarily rearranged version. And this idea is precisely what the next section presents.

Exercise 424. Is `(bundle "abc" 0)` a proper use of the `bundle` function? What does it produce? Why? ■

Exercise 425. Define the function `list->chunks`. It consumes a list `l` of arbitrary data and a natural number `n`. The function's result is a list of list chunks of size `n`. Each chunk represents a sub-sequence of items in `l`.

Use `list->chunks` to define `bundle` via function composition. ■

Exercise 426. Define the function `partition`. It consumes a `String s` and a natural number `n`. The function produces a list of string chunks of size `n`.

For non-empty strings `s` and positive natural numbers `n`,

■ `(equal? (partition s n) (bundle (explode s) n))`

is `#true`. But don't use this equality as the definition for `partition`; use `substring` instead.

Hint Have `partition` produce its “natural” result for the empty string. For the case where `n` is `0`, see [exercise 424](#).

Note The `partition` function is somewhat closer to what a cooperative DrRacket environment would need than `bundle`. ■

29.2 Recursion that Ignores Structure

Recall that the `sort>` function from section [Design by Composition](#) consumes a list of, say, numbers and re-arranges the same list of numbers in some order, typically ascending or descending. It proceeds by inserting the first number into the appropriate position of the sorted rest of the list. Put differently, it is a structurally recursive function that re-processes the result of the natural recursions.

Hoare's quick-sort algorithm goes about sorting lists in a radically different manner and is the classic example of generative recursion. The underlying generative step uses the time-honored strategy of divide-and-conquer. That is, it divides the non-trivial instances of the problem into two smaller, related problems, solves those smaller problems, and combines their solutions into a solution for the original problem. In the case of the quick-sort algorithm, the intermediate goal is to divide the list of numbers into two lists:

- one that contains all the numbers that are strictly smaller than the first item
- and another one with all those items that are strictly larger than the first.

Then the two smaller lists are sorted via the quick-sort algorithm. Once the two lists are sorted, it composes the two pieces with the first item in the middle. Owing to its special role, the first item on the list is called the *pivot item*.

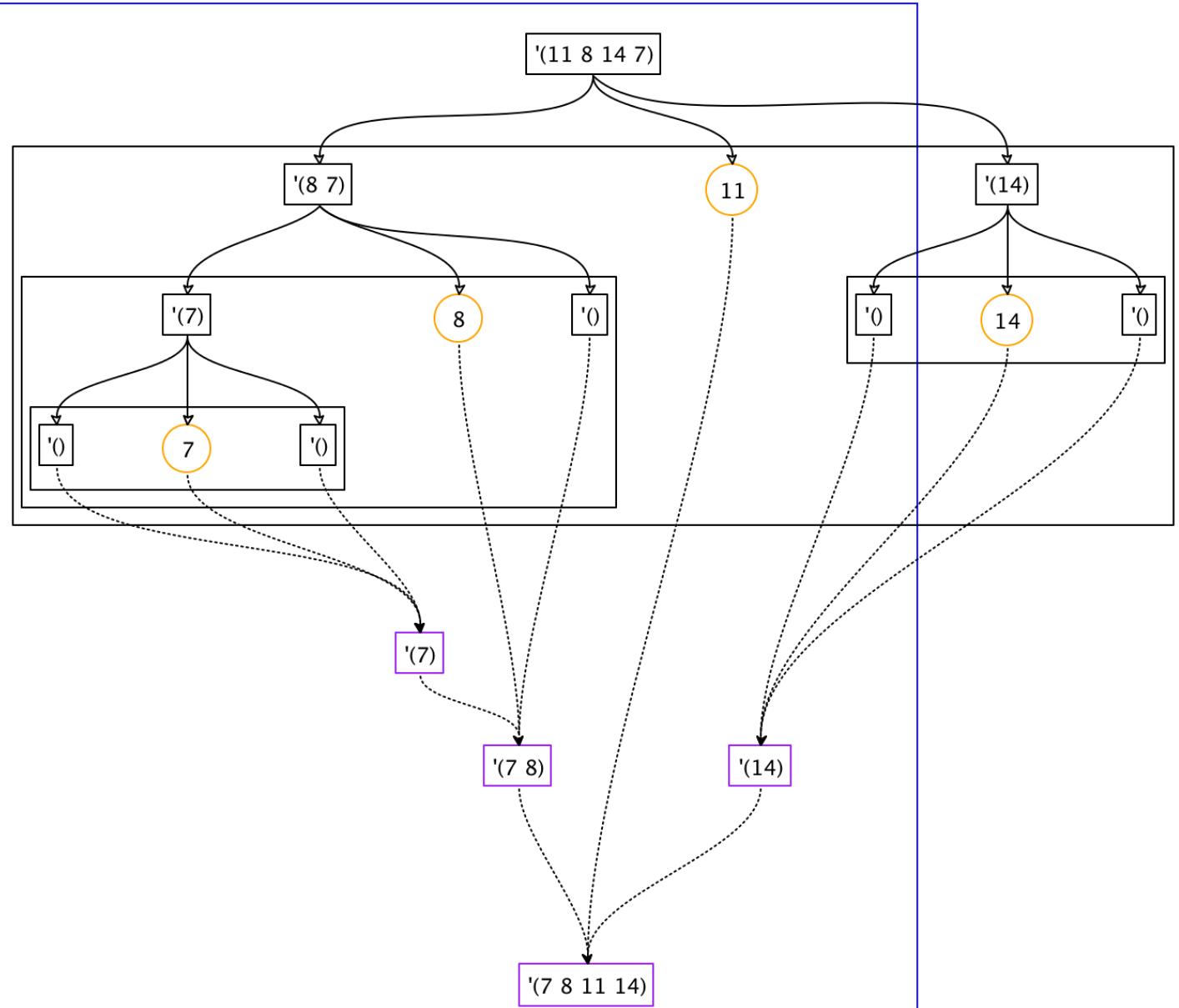


Figure 108: A graphical illustration of the quick-sort algorithm

To develop an understanding of how the quick-sort algorithm works, let's walk through an example, quick-sorting `(list 11 8 14 7)`. Figure 108 illustrates the process in a graphical way. The figure consists of a top half, the divide phase, and the bottom half, the conquer phase.

The partition phase is represented with boxes and solid arrows. Three arrows emerge from each boxed list and go to a box with three pieces: the circled pivot element in the middle, to its left the boxed list of numbers smaller than the pivot, and to its right the boxed list of those numbers that are larger than the pivot. Each of these steps isolates at least one number as the pivot, meaning the two neighboring lists are shorter than the given list. Consequently, the overall process terminates too.

Consider the first step where the input is `(list 11 8 14 7)`. The pivot item is `11`. Partitioning the list into items larger and smaller than `11` produces `(list 8 7)` and `(list 14)`. The remaining steps of the partitioning phase work in an analogous way. Partitioning ends when all numbers have been isolated as pivot elements. At this point, you can already read off the final result by reading the pivots from left to right.

The conquering phase is represented with dashed arrows and boxed lists. Three arrows enter each result box: the middle one from a pivot, the left one from the boxed result of sorting the smaller numbers, and the right one from the boxed result of sorting the larger

ones. Each step adds at least one number to the result list, the pivot, meaning the lists grow toward the bottom of the diagram. The box at the bottom is a sorted variant of the given list at the top.

Take a look at the left-most, upper-most conquer step. It combines the pivot 7 with two empty lists, resulting in '(7). The next one down corresponds to the partitioning step that isolated 8 and thus yields '(7 8). Each level in the conquering phase mirrors a corresponding level from the partitioning phase. After all, the overall process is recursive.

Exercise 427. Draw a quick-sort diagram for (list 11 9 2 18 12 14 4 1). ■

Now that we have a good understanding of the quick-sort idea, we can translate it into ISL+. Clearly, quick-sort distinguishes two cases. If the input is '(), it produces '() because this list is sorted already; otherwise, it performs a generative recursion. This case split suggests the following cond expression:

```
; [List-of Number] -> [List-of Number]
; creates a list of numbers with the same numbers as
; alon, sorted in ascending order
(define (quick-sort alon)
  (cond
    [(empty? alon) '()]
    [else ...]))
```

The answer for the first case is given. For the second case, when quick-sort's input is a non-empty list, the algorithm uses the first item to partition the rest of the list into two sublists: a list with all items smaller than the pivot item and another one with those larger than the pivot item.

Since the rest of the list is of unknown size, we leave the task of partitioning the list to two auxiliary functions: smaller-items and larger-items. They process the list and filter out those items that are smaller and larger, respectively, than the pivot. Hence each auxiliary function accepts two arguments, namely, a list of numbers and a number. Designing these two functions is an exercise in structural recursion. Try on your own or read the definitions shown in figure 109.

```
; [List-of Number] -> [List-of Number]
; creates a list of numbers with the same numbers as
; alon, sorted in ascending order
; assume the numbers are all distinct
(define (quick-sort alon)
  (cond
    [(empty? alon) '()]
    [else (local ((define pivot (first alon)))
            (append (quick-sort (smaller-items alon pivot))
                    (list pivot)
                    (quick-sort (larger-items alon pivot))))])))

; [List-of Number] Number -> [List-of Number]
; creates a list with all those numbers on alon
; that are larger than n
(define (larger-items alon n)
  (cond
    [(empty? alon) '()]
    [else (if (> (first alon) n)
              (cons (first alon) (larger-items (rest alon) n))
              '())]))
```

```

        (larger-items (rest alon) n)))))

; [List-of Number] Number -> [List-of Number]
; creates a list with all those numbers on alon
; that are smaller than n
(define (smaller-items alon n)
  (cond
    [(empty? alon) '()]
    [else (if (< (first alon) n)
               (cons (first alon) (smaller-items (rest alon) n))
               (smaller-items (rest alon) n)))]))

```

Figure 109: The quick-sort algorithm

Each of these lists is sorted separately, using `quick-sort`, which implies the use of recursion, specifically the following two expressions:

1. `(quick-sort (smaller-items alon pivot))`, which sorts the list of items smaller than the pivot; and
2. `(quick-sort (larger-items alon pivot))`, which sorts the list of items larger than the pivot.

Once `quick-sort` has the sorted versions of the two lists, it must combine the two lists and the pivot in the proper order: first all those items smaller than pivot, then pivot, and finally all those that are larger. Since the first and last list are already sorted, `quick-sort` can simply use `append`:

```

(append (quick-sort (smaller-items alon pivot))
        (list (first alon))
        (quick-sort (larger-items alon pivot)))

```

[Figure 109](#) contains the full program; read it before proceeding.

Now that we have an actual function definition, we can evaluate the example from above by hand:

```

(quick-sort (list 11 8 14 7))
==
(append (quick-sort (list 8 7))
        (list 11)
        (quick-sort (list 14)))
==
(append (append (quick-sort (list 7))
                 (list 8)
                 (quick-sort '()))
        (list 11)
        (quick-sort (list 14)))
==
(append (append (append (quick-sort '())
                         (list 7)
                         (quick-sort '()))
                     (list 8)
                     (quick-sort '()))
                    (list 11)
                    (quick-sort (list 14)))
==

```

```

  (append (append (append '()
                           (list 7)
                           '())
                           (list 8)
                           '())
                           (list 11)
                           (quick-sort (list 14)))
== 
  (append (append (list 7)
                  (list 8)
                  '())
                  (list 11)
                  (quick-sort (list 14)))
...

```

The calculation shows the essential steps of the sorting process, that is, the partitioning steps, the recursive sorting steps, and the concatenation of the three parts. From this calculation, it is easy to see how `quick-sort` implements the process illustrated in figure 108.

Both figure 108 and the calculation also show how `quick-sort` completely ignores the structure of the given list. The first recursion works on two distant numbers from the originally given list and the second one on the list's third item. These recursions aren't random but they are certainly not relying on the structure of the data definition.

Contrast `quick-sort`'s organization with that of the `sort>` function from [Design by Composition](#). The design of the latter follows the structural design recipe, yielding a program that processes a list item by item. By splitting the list, `quick-sort` can speed up the process of sorting the list, though at the cost of not using plain `first` and `rest`.

Exercise 428. Complete the hand evaluation.

The hand evaluation of `(quick-sort (list 11 8 14 7))` suggests an additional trivial case for `quick-sort`. Every time `quick-sort` consumes a list of one item, it produces the very same list. After all, the sorted version of a list of one item is the list itself.

Modify the definition of `quick-sort` to take advantage of this observation.

Hand evaluate the example again. How many steps does the extended algorithm save? ▀

Exercise 429. While `quick-sort` quickly reduces the size of the problem in many cases, it is inappropriately slow for small problems. Hence people use `quick-sort` to reduce the size of the problem and switch to a different sort function when the list is small enough.

Develop a version of `quick-sort` that uses `sort>` from [Recursive Auxiliary Functions](#) if the length of the input is below some threshold. ▀

Exercise 430. If the input to `quick-sort` contains the same number several times, the algorithm returns a list that is strictly shorter than the input. Why? Fix the problem so that the output is as long as the input. ▀

Exercise 431. Use `filter` to define `smaller-items` and `larger-items` as one-liners. ▀

Exercise 432. Develop a variant of `quick-sort` that uses only one comparison function, say, `<`. Its partitioning step divides the given list `alon` into a list that contains the items of `alon` smaller than (`first alon`) and another one with those that are not smaller.

Use `local` to package up the program as a single function: Abstract this function so that it consumes a list and a comparison function. ■

30 Designing Algorithms

The overview for this part already explains that the design of generative recursion functions is more ad hoc than structural design. As the first chapter shows, two generative recursions can radically differ in how they process functions. Both `bundle` and `quick-sort` process lists, but while the former at least respects the sequencing in the given list, the latter rearranges its given list at will. The question is whether a single design recipe can help with the creation of such widely differing functions.

The first section of this chapter shows how to adapt the process dimension of the design recipe to generative recursion. The second section hones in on another new phenomenon: an algorithm may fail to produce an answer for some of its inputs. Programmers must therefore analyze their programs and supplement the design information with a comment on termination. The remaining sections in this chapter compare and contrast structural and generative recursion.

30.1 Adapting the Design Recipe

Let's examine the six general steps of our structural design recipe in light of the examples in the preceding chapter:

- As before, we must represent the problem information as data in our chosen programming language. The choice of a **data representation** for a problem affects our thinking about the computational process, so some planning ahead is necessary. Alternatively, be prepared to backtrack and to explore different data representations. Regardless, we must analyze the problem information and define data collections.
- We also need a signature, a function header, and a purpose statement. Since the generative step has no connection to the structure of the data definition, the purpose statement must go beyond **what** the function is to compute and also explain **how** the function computes its result.
- It is useful to explain the “how” with function examples, the way we explained `bundle` and `quick-sort` in the previous chapter. That is, while function examples in the structural world merely specify which output the function is to produce for which input, the purpose of examples in the world of generative recursion is to explain the underlying idea behind the computational process.

For `bundle`, the examples specify how the function acts in general and in certain boundary cases. For `quick-sort`, the example in [figure 108](#) illustrates how the function partitions the given list with respect to the pivot item. By adding such worked examples to the purpose statement, we—the designers—gain an improved understanding of the desired process, and we communicate this understanding to future readers of this code.

- Our discussion suggests a general template for algorithms. Roughly speaking, the design of an algorithm distinguishes two kinds of problems: those that are *trivially solvable* and those that are not. If a given problem is trivially solvable, an algorithm produces the matching solution. For example, the problems of sorting an

For this part of the book, “trivial” is a technical term.

empty list or a one-item list are trivially solvable. A list with many items is a non-trivial problem. For these non-trivial problems, algorithms commonly generate new problems of the same kind as the given one, solve those recursively, and combine the solutions into an overall solution.

Based on this rough sketch, all algorithms have roughly this organization:

```
(define (generative-recursive-fun problem)
  (cond
    [(trivially-solvable? problem)
     (determine-solution problem)]
    [else
     (combine-solutions
      ...
      problem ...
      (generative-recursive-fun (generate-problem-1 problem))
      ...
      (generative-recursive-fun (generate-problem-n problem)))])))
```

The original problem is occasionally needed to combine the solutions for the newly generated problems, which is why it is handed over to `combine-solutions`.

- This template is only a suggestive blueprint, not a definitive shape. Each piece of the template is to remind us to think about the following four questions:
 - What is a trivially solvable problem?
 - How are trivial solutions solved?
 - How does the algorithm generate new problems that are more easily solvable than the original one? Is there one new problem that we generate or are there several?
 - Is the solution of the given problem the same as the solution of (one of) the new problems? Or, do we need to combine the solutions to create a solution for the original problem? And, if so, do we need anything from the original problem data?

To define the algorithm as a function, we must express the answers to these four questions as functions and expressions in terms of the chosen data representation.

For this step, the table-driven attempt from [Designing with Self-Referential Data Definitions](#) might help again. Reconsider the `quick-sort` example from [Recursion that Ignores Structure](#). The central idea behind `quick-sort` is to divide a given list into a list of smaller items and larger items and to sort those separately. [Figure 110](#) spells out how some simple numeric examples work out for the non-trivial cases. From these examples it is straightforward to guess that the answer to the fourth question is

`; append sorted-smaller, pivot, and sorted-larger into one list`

which can easily be translated into code.

- Once the function is complete, it is time to test it. As before, the goal of testing is to discover and eliminate bugs. Remember that testing cannot validate that the function works correctly for all possible inputs.

alon	pivot	sorted, smaller	sorted, larger	expected
'(2 3 1 4)	2	'(1)	'(3 4)	'(1 2 3 4)
'(2 0 1 4)	2	'(0 1)	'(3)	'(0 1 2 4)
'(3 0 1 4)	3	'(0 1)	'(4)	'(0 1 3 4)

Figure 110: The table-based guessing approach for combining solutions

Exercise 433. Formulate informal answers to the four key questions for `bundle`. ▀

Exercise 434. Formulate informal answers to the first three key questions for the `quicksort` problem. How many instances of `generate-problem` are there? ▀

Exercise 435. [Exercise 221](#) defines the function `food-create`, which consumes a `Posn` and produces a randomly chosen, guaranteed distinct `Posn`. The exercise defers to this part of the book for an explanation of the design of the function. Justify the design of `food-create`. Re-formulate the two functions as a single definition; use `local..`. ▀

30.2 Termination

Generative recursion adds an entirely new aspect to computations: non-termination. A function such as `bundle` may never produce a value nor signal an error for certain inputs.

Exercise 424 asks what the result of `(bundle '("a" "b" "c") 0)` is, and here is an explanation of why it does not have a result:

```
(bundle '("a" "b" "c") 0)
== (cons (implode (take '("a" "b" "c") 0)) (bundle (drop '("a" "b" "c") 0)))
== (cons (implode '()) (bundle (drop '("a" "b" "c") 0)))
== (cons "" (bundle (drop '("a" "b" "c") 0)))
== (cons "" (bundle '("a" "b" "c") 0))
```

The calculation shows that determining the result of `(bundle '("a" "b" "c") 0)` requires having a result for the very same expression. In the context of ISL+ this means the evaluation does not stop. Computer scientists say that `bundle` does not *terminate* when the second argument is `0`; they also say that the function *loops* or that the computation is stuck in an *infinite loop*.

Contrast this insight with the designs presented in the first four parts. Every function designed according to the recipe either produces an answer or raises an error signal for every input. After all, the recipe dictates that each natural recursion consumes an immediate piece of the input, not the input itself. Because data is constructed in a hierarchical manner, input shrinks at every stage. Eventually the function is applied to an atomic piece of data, and the recursion stops.

This reminder also explains why generative recursive functions may diverge. According to the design recipe for the latter, an algorithm may generate new problems without any limitations. If the design recipe required the designer to guarantee that the new problem were “smaller” than the given one, it would terminate. But, imposing such a restriction would needlessly complicate the design of functions such as `bundle`.

The theory of computation actually shows that we must lift these restrictions eventually.

In this book, we therefore keep the first six steps of the design recipe intact and supplement them with a seventh step: the *termination argument*. [Figure 111](#) summarizes this decision in a table. It shows the new design recipe in the conventional tabular form. The unmodified steps come with – in the **activity** column. Others come with comments on how the design recipe for generative recursion differs from the one for structural recursion. The last row is completely new.

steps	outcome	activity
-------	---------	----------

problem analysis	data representation and definition	—
header	a purpose statement concerning the ``how'' of the function	supplement the explanation of what the function computes with a one-liner on how it computes the result
examples	examples and tests	explain the ``how'' with several examples
template	fixed template	—
definition	full-fledged definition	formulate conditions for trivially solvable problems; formulate answers for these trivial cases; determine how to generate new problems for non-trivial problems, possibly using auxiliary functions; determine how to combine the solutions of the generated problems into a solution for the given problem
tests	discover mistakes	—
termination	(1) a size argument for each recursive call or (2) examples of exceptions to termination	investigate whether the problem data for each recursive data is smaller than the given data; find examples that cause the function to loop

Figure 111: Designing algorithms

A termination argument comes in one of two forms. The first one argues why each recursive call works on a problem that is smaller than the given one. Often this argument is straightforward; on rare occasions, you will need to work with a mathematician to prove a new theorem for such arguments. The second kind of termination argument illustrates with an example that the function may not terminate. Ideally it should also describe the class of data for which the function may loop.

In some extremely rare cases, you may not be able to make either argument because computer science does not know enough yet.

You cannot define a predicate for this class; otherwise you could modify the function and ensure that it always terminates.

Let's illustrate the two kinds of termination arguments with two examples. For the `bundle` function, it suffices to warn readers about chunk size 0:

```
; [List-of 1String] N -> [List-of String]
; bundles sub-sequences of s into strings of length n
; termination (bundle s 0) loops unless s is '()
(define (bundle s n)
  ...)
```

In this case, it is possible to define a predicate that precisely describes when `bundle` terminates. Do so before you read on. For `quick-sort`, the key observation is that each recursive use of `quick-sort` receives a shorter list:

```
; [List-of Number] -> [List-of Number]
; creates a sorted variant of alon
; termination both recursive calls to quick-sort
; processes list that are guaranteed to miss the pivot
(define (quick-sort alon)
  ...)
```

In one case, the list consists of the numbers that are strictly smaller than the pivot; the other one is for numbers strictly larger.

Exercise 436. Develop a checked version of `bundle` that is guaranteed to terminate for all inputs. It may signal an error for those cases where the original version loops. ■

Exercise 437. Consider the following definition of `smaller-items`, one of the two “problem generators” for `quick-sort`:

```
; [List-of Number] Number -> [List-of Number]
; creates a list with all those numbers on alon
; that are smaller than or equal to threshold
(define (smaller-items alon threshold)
  (cond
    [(empty? alon) '()]
    [else (if (<= (first alon) threshold)
              (cons (first alon) (smaller-items (rest alon) threshold))
              (smaller-items (rest alon) threshold))]))
```

What can go wrong when this version is used with the `quick-sort` definition from [Recursion that Ignores Structure?](#) ■

Exercise 438. When you worked on [exercise 432](#) or [exercise 430](#), you may have produced looping solutions. Similarly, [exercise 437](#) actually reveals how brittle the termination argument is for `quick-sort`. In all cases, the argument relies on the idea that `smaller-items` and `larger-items` produce lists that are maximally as long as the given list, and our understanding that neither includes the given pivot in the result.

Based on this explanation, modify the definition of `quick-sort` so that both functions already receive lists that are shorter than the given one. ■

Exercise 439. Formulate a termination argument for `food-create` from [exercise 435](#). ■

30.3 Structural versus Generative Recursion

The template for algorithms is so general that it includes structurally recursive functions. Consider the left side of [figure 112](#). This template is specialized to deal with one trivial clause and one generative step. If we replace `trivial?` with `empty?` and generate with `rest`, we get a template for list-processing functions; see the right side of [figure 112](#).

<pre>(define (general P) (cond [(trivial? P) (solve P)] [else (combine-solutions P (general (generate P))))]))</pre>	<pre>(define (special P) (cond [(empty? P) (solve P)] [else (combine-solutions P (special (rest P))))]))</pre>
--	--

Figure 112: From generative to structural recursion

Exercise 440. Define `solve` and `combine-solutions` so that

- `special` computes the length of its input,
- `special` negates each number on the given list of numbers, and
- `special` uppercases the given list of strings.

What do you conclude from these exercises? ■

Is there a real difference between structural recursive design and the one for generative recursion?

Our answer is “it depends.” Of course, we could say that all functions using structural recursion are just special cases of generative recursion. This “everything is equal” attitude, however, is of no help if we wish to understand the process of designing functions. It confuses two kinds of design that require different forms of knowledge and that have different consequences. One relies on a systematic data analysis and not much more; the other requires a deep, often mathematical, insight into the problem-solving process itself. One leads programmers to naturally terminating functions; the other requires a termination argument. Conflating these two approaches is simply unhelpful.

30.4 Making Choices

When you interact with a function `f` that consumes lists of numbers and produces sorted variants, it is impossible for you to know whether `f` is the `sort>` function or `quick-sort`. The two functions behave in an observably equivalent way. This raises the question of which of the two a programming language should provide in its library. More generally, when we can design a function using structural recursion and generative recursion, we need to figure out which one to pick.

Observable equivalence plays a central role in the study of programming languages.

To understand this choice, let’s discuss another classical example of generative recursion from mathematics: the problem of finding the greatest common divisor of two positive natural numbers. All such numbers have at least one divisor in common: 1. On occasion, this is also the only common divisor. For example, 2 and 3 have only 1 as common divisor because 2 and 3, respectively, are the only other divisors. Then again, 6 and 25 are both numbers with several divisors:

Dr. John Stone suggested the material on the greatest common divisor.

- 6 is evenly divisible by 1, 2, 3, and 6;
- 25 is evenly divisible by 1, 5, and 25.

Still, their greatest common divisor is 1. In contrast, 18 and 24 have many common divisors and their greatest common divisor is 6:

- 18 is evenly divisible by 1, 2, 3, 6, 9, and 18;
- 24 is evenly divisible by 1, 2, 3, 4, 6, 8, 12, and 24.

Completing the first three steps of the design recipe is straightforward:

```
; N[>= 1] N[>= 1] -> N
; finds the greatest common divisor of n and m

(check-expect (gcd 6 25) 1)
(check-expect (gcd 18 24) 6)

(define (gcd n m)
  1)
```

The signature specifies the precise inputs: natural numbers greater or equal to 1.

From here we design both a structural and a generative recursive solution. Since this part of the book is about generative recursion, we merely present a structural solution in figure 113 and leave the design ideas to exercises. Just note that `(= (remainder n i) (remainder m i) 0)` encodes the idea that both n and m are “evenly divisible” by i.

```
(define (gcd-structural n m)
  (local (; N -> N
          ; determines the greatest divisor of n and m less than i
          (define (greatest-divisor-<= i)
            (cond
              [(= i 1) 1]
              [else (if (= (remainder n i) (remainder m i) 0)
                        i
                        (greatest-divisor-<= (- i 1))))])
          (greatest-divisor-<= (min n m))))
```

Figure 113: Finding the greatest common divisor via structural recursion

Exercise 441. Explain in your words how `greatest-divisor-<=` works. Use the design recipe to find the right words. Why is `greatest-divisor-<=` applied to `(min n m)` in the body of the `local` definition? ■

Although the design of `gcd-structural` is rather straightforward, it is also naive. It simply tests for every number between the smaller of n and m and 1 whether it divides both n and m evenly and returns the first such number. For small n and m, this works just fine. Consider the following example, however:

```
(gcd-structural 101135853 45014640)
```

The result is 177 and to get there, `gcd-structural` had to check the “evenly divisible” condition for 101135676, that is, 101135853 - 177, numbers. Checking that many `remainders`—actually twice as many—is a large effort, and even reasonably fast computers spend many seconds on this task.

Exercise 442. Copy `gcd-structural` into the definitions area of DrRacket and evaluate

```
(time (gcd-structural 101135853 45014640))
```

in the interactions area. ■

Since mathematicians recognized the inefficiency of this structural function a long time ago, they studied the problem of finding divisors in depth. The essential insight is that

for two natural numbers—large and small—the greatest common divisor is

equal to the greatest common divisor of `small` and the remainder of `large` divided by `small`.

Here is how we can articulate this insight as an equation:

```
(gcd large small) == (gcd small (remainder large small))
```

Since `(remainder large small)` is smaller than both `large` and `small`, the right-hand side use of `gcd` consumes `smaller` first.

Here is how this insight applies to our small example:

- The given numbers are `18` and `24`.
- According to the insight, they have the same greatest common divisor as `18` and `6`.
- And these two have the same greatest common divisor as `6` and `0`.

And here we seem stuck because `0` is unexpected. But, `0` can be evenly divided by every number, meaning we have found our answer: `6`.

Working through the example not only validates the basic insight but also suggests how to turn the insight into an algorithm:

- when the smaller of the two given numbers is `0`, we are faced with a trivial case;
- the larger of the two numbers is the solution in the trivial case;
- generating a new problem requires a single `remainder` operation; and
- the above equation tells us that the answer to the newly generated problem is also the answer to the originally given problem.

In other words, the answers for all four questions from the design recipe just fall out.

```
(define (gcd-generative n m)
  (local (; N[>= 1] N[>=1] -> N
          ; generative recursion
          ; (gcd large small) == (gcd small (remainder large small)))
    (define (clever-gcd large small)
      (cond
        [(= small 0) large]
        [else (clever-gcd small (remainder large small))])))
  (clever-gcd (max m n) (min m n))))
```

Figure 114: Finding the greatest common divisor via generative recursion

Figure 114 presents the definition of the algorithm. The `local` definition introduces the workhorse of the function: `clever-gcd`. Its first `cond` line discovers the trivial case by comparing `smaller` to `0` and produces the matching solution. The generative step uses `smaller` as the new first argument and `(remainder large small)` as the new second argument to `clever-gcd`.

If we now use `gcd-generative` with our complex example from above:

```
(gcd-generative 101135853 45014640)
```

we see that the response is nearly instantaneous. A hand-evaluation shows that `clever-gcd` recurs only nine times before it produces the solution:

```

...
== (clever-gcd 101135853 45014640)
== (clever-gcd 45014640 11106573)
== (clever-gcd 11106573 588348)
== (clever-gcd 588348 516309)
== (clever-gcd 516309 72039)
== (clever-gcd 72039 12036)
== (clever-gcd 12036 11859)
== (clever-gcd 11859 177)
== (clever-gcd 177 0)

```

This also means that it checks only nine `remainder` conditions, clearly a much smaller effort than `gcd-structural` expends.

Exercise 443. Copy `gcd-generative` into the definitions area of DrRacket and evaluate

```
| (time (gcd-generative 101135853 45014640))
```

in the interactions area. |

You may now think that generative recursion design has discovered a much faster solution to the `gcd` problem, and you may conclude that generative recursion is always the right way to go. This judgment is too rash for three reasons. First, even a well-designed algorithm isn't always faster than an equivalent structurally recursive function. For example, `quick-sort` wins only for large lists; for small ones, the standard `sort>` function is faster. Worse, a badly designed algorithm can wreak havoc on the performance of a program. Second, it is typically easier to design a function using the recipe for structural recursion. Conversely, designing an algorithm requires an idea of how to generate new problems, a step that often requires some deep insight. Finally, people who read functions can easily understand structurally recursive functions, even without much documentation. To understand an algorithm, the generative step must be explained really well, but generating a really good explanation can be a lot of hard work.

Experience shows that most functions in a program employ structural design; only a few exploit generative recursion. When we encounter a situation where a design could use the recipe for either structural or generative recursion, the best approach is to start with a structural version. If it turns out to be too slow for the task at hand, explore the use of generative recursion.

Exercise 444. Evaluate

```
| (quick-sort (list 10 6 8 9 14 12 3 11 14 16 2))
```

by hand. Show only those lines that introduce a new recursive call to `quick-sort`. How many recursive applications of `quick-sort` are required? How many recursive applications of the `append` function? Suggest a general rule for a list of length n .

Evaluate

```
| (quick-sort (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14))
```

by hand. How many recursive applications of `quick-sort` are required? How many recursive applications of `append`? Does this contradict the first part of the exercise? |

Exercise 445. Add `sort>` and `quick-sort` to the definitions area. Run tests on the functions to ensure that they work on basic examples. Also develop `create-tests`, a function that creates large test cases randomly. Then explore how fast each works on various lists.

Does the experiment confirm the claim that the plain `sort>` function wins in many comparisons over `quick-sort` for short lists and vice versa?

Determine the cross-over point. Use this information to build a `clever-sort` function that behaves like `quick-sort` for large lists and switches over to the plain `sort>` function for lists below this cross-over point. See [exercise 429](#). ■

Exercise 446. Given the header material for `gcd-structural`, a naive use of the design recipe might use the following template or some variant:

```
(define (gcd-structural n m)
  (cond
    [(and (= n 1) (= m 1)) ...]
    [(and (> n 1) (= m 1)) ...]
    [(and (= n 1) (> m 1)) ...]
    [else (... (gcd-structural (sub1 n) (sub1 m)) ...
               ... (gcd-structural (sub1 n) m) ...
               ... (gcd-structural n (sub1 m)) ...))]))
```

Why is it impossible to find a divisor with this strategy? ■

Exercise 447. [Exercise 446](#) means that the design for `gcd-structural` calls for some planning and a design by composition approach.

The very explanation of “greatest common denominator” suggests a two-stage approach. First design a function that can compute the list of divisors of a natural number. Second, design a function that picks the largest common number in the list of divisors of `n` and the list of divisors of `m`. The overall function would look like this:

Ideally, you should use sets not lists.

```
(define (gcd-structural small large)
  (largest-common (divisors smaller smaller) (divisors small large)))

; N[>= 1] N[>= 1] -> [List-of N]
; computes the list of divisors of l smaller or equal to k
(define (divisors k l)
  '())

; [List-of N] [List-of N] -> N
; finds the largest number common to both k and l
(define (largest-common k l)
  1)
```

Before you design `largest-common` explain why `divisors` consumes two numbers and why it consumes `smaller` in both cases. ■

31 Variations on the Theme

The design of an algorithm starts with an informal description of a process of how to create a problem that is more easily solvable than the given one and whose solution contributes to the solution of the given problem. Coming up with this kind of idea requires inspiration, penetration of an area, and experience with many different kinds of examples.

This chapter presents several illustrative examples of algorithms. Some are directly drawn

from mathematics, which is the source of many ideas; others come from computational settings. The first example is a graphical illustration of our principle: the Sierpinski triangle. The second one explains the divide-and-conquer principle with the simple mathematical example of finding the root of a function. It then shows how to turn this idea into a fast algorithm for searching sequences, a widely used application. The third section concerns “parsing” of sequences of `1Strings`, also a common problem in real-world programming.

31.1 Fractals, a First Taste

Fractals play an important role in computational geometry. Flake writes in *The Computational Beauty of Nature* (The MIT Press, 1998) that “geometry can be extended to account for objects with a fractional dimension. Such objects, known as *fractals*, come very close to capturing the richness and variety of forms found in nature. Fractals possess structural self-similarity on multiple ... scales, meaning that a piece of a fractal will often look like the whole.”

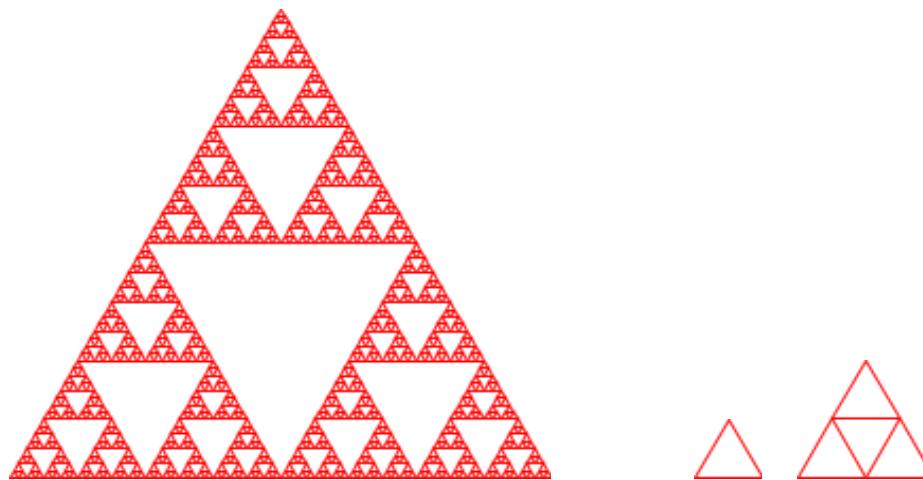


Figure 115: The Sierpinski triangle

[Figure 115](#) displays an example of a fractal shape, known as the Sierpinski triangle. The basic shape is an (equilateral) triangle, like the one in the center. When this triangle is composed sufficiently many times in a triangular fashion, we get the left-most shape.

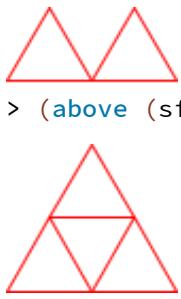
The right-most image in [figure 115](#) explains the generative step. When taken by itself, it says that given a triangle, find the midpoint of each side and connect them to each other. This step yields four triangles; repeat the process for each of the outer of these three triangles unless these triangles are too small.

An alternative explanation, well suited for the shape composition functions in the `2htdp/image` library, is based on the transition from the image in the center to the image on the right. By juxtaposing two of the center triangles and then placing one copy above these two, we also get the shape on the right:

We owe this solution to Dr. Marc Smith.

```
> (sf SMALL)

> (beside (sf SMALL) (sf SMALL))
```



```
> (above (sf SMALL) (beside (sf SMALL) (sf SMALL)))
```

This section uses the alternative description to design the Sierpinski algorithm; [Accumulators as Results](#) deals with the first description. Given that the goal is to generate the image of an equilateral triangle, we encode the problem with a (positive) number, the length of the triangle's side. This decision, yields a signature, purpose statement, and header:

```
; Number -> Image
; creates Sierpinski triangle of size side

(define (sierpinski side)
  (triangle side 'outline 'red))
```

Now it is time to address the four questions of generative recursion:

- When the given number is so small that drawing triangles inside of it is pointless, the problem is trivial.
- In that case, it suffices to generate a triangle.
- Otherwise, the algorithm must generate a Sierpinski triangle of size $side / 2$ because juxtaposing two such triangles in either direction yields one of size $side$.
- If `half-sized` is the Sierpinski triangle of size $side / 2$, then

```
(above half-sized
      (beside half-sized half-sized))
```

is a Sierpinski triangle of size $side$.

With these answers, it is straightforward to define the function.

```
(define SMALL 4)

(define small-triangle (triangle SMALL 'outline 'red))

; Number -> Image
; generative creates Sierpinski triangle of size side by generating
; one of size side/2 and placing one copy above two composed copies

(check-expect (sierpinski SMALL) small-triangle)
(check-expect (sierpinski (* 2 SMALL)))
  (above small-triangle
        (beside small-triangle small-triangle)))

(define (sierpinski side)
  (cond
    [(<= side SMALL) (triangle side 'outline 'red)]
    [else (local ((define half-sized (sierpinski (/ side 2))))
```

```
(above half-sized
  (beside half-sized half-sized))))])
```

Figure 116: The Sierpinski algorithm

Figure 116 spells out the details. The “triviality condition” translates to (`<= side SMALL`) for some constant `SMALL`. For the trivial answer, the function returns a (red) triangle of the given size. In the recursive case, a `local` expression introduces the name `half-sized` for the Sierpinski triangle that is half as big as the specified size. Once the recursive call has generated the small Sierpinski triangle, the `above-beside` composition of three copies yields the desired triangle.

The figure highlights two other points. First, the purpose statement is articulated as an explanation of **what** the function accomplishes

```
; create Sierpinski triangle of size side by ...
```

and **how** it accomplishes this goal:

```
; generating one of size side/2 and
; placing one copy above two composed copies
```

Second, the examples illustrate the two possible cases: one if the given size is small enough, and one for a size that is too large still. In the latter case, the expression that computes the expected value explains exactly the meaning of the purpose statement.

Since `sierpinski` is based on generative recursion, defining the function and testing it is not the last step. We must also consider why the algorithm terminates for any given legal input. The input of `sierpinski` is a single positive number. If the number is smaller than `SMALL`, the algorithm terminates. Otherwise, the recursive call uses a number that is half as large as the given one. Hence, the algorithm must terminate for all positive `sides`, assuming `SMALL` is positive, too.

One view of the Sierpinski process is that it divides its problem in half until it is immediately solvable. With a little imagination, you can see that the process can be used to search for numbers with certain properties. The next section explains this idea in detail.

31.2 Binary Search

Applied mathematicians model the real-world with non-linear equations and then try to solve them. Specifically, they translate problems into a function f from numbers to numbers and look for some number r such that

$$f(r) = 0.$$

The value r is called the *root* of f .

Here is a problem from the physical domain:

Sample Problem: A rocket is flying at the constant speed of v miles per hour on a straight line towards some target, d_0 miles away. It then accelerates at the rate of a miles per hour squared for t hours. When will it hit its target?

Physics tells us that the distance covered is the following function of time:

$$d(t) = (v * t + 1/2 * a * t^2)$$

The question of when it hits the target asks us to find the time t_0 such that the object reaches the desired goal:

$$d_0 = (v * t_0 + 1/2 * a * t_0^2)$$

From algebra we know that this is a quadratic equation and that it is possible to solve such equations given d_0 , a , and v satisfy certain conditions.

Generally such problems call for more complex equations than quadratic ones. In response, mathematicians have spent the last few centuries developing root-finding methods for different types of functions. In this section, we study a solution that is based on the **Intermediate Value Theorem**, an early result of mathematical analysis. The resulting algorithm is a primary example of generative recursion based on a mathematical theorem. It has been adapted to other uses and has become known as the *binary search* algorithm in computer science.

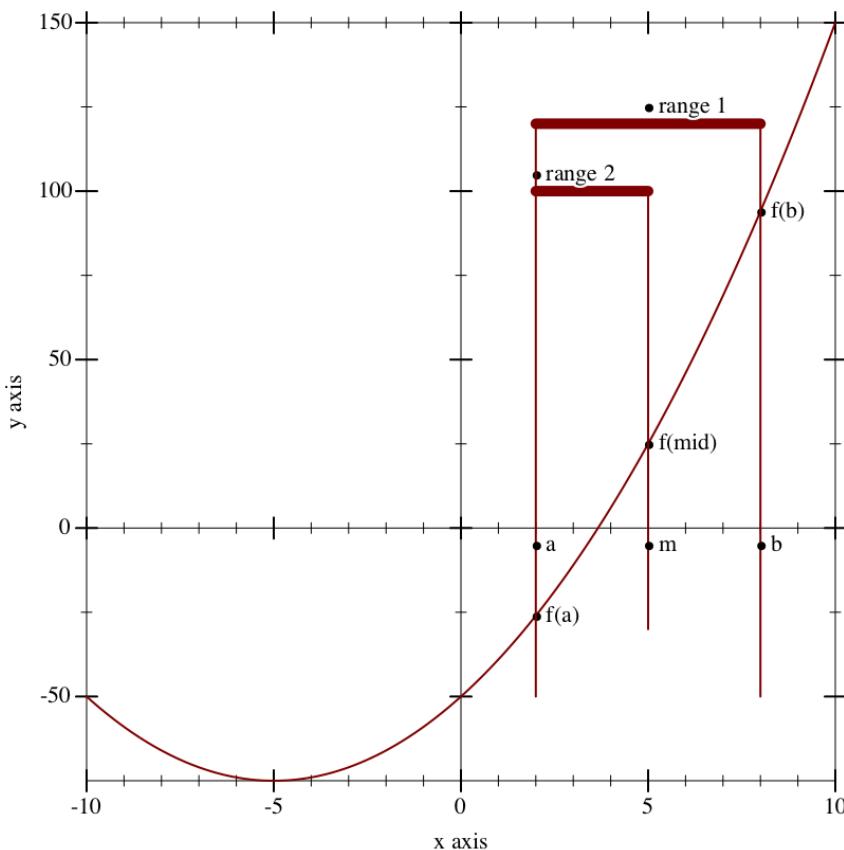


Figure 117: A numeric function f with root in interval $[a,b]$ (stage 1)

The Intermediate Value Theorem says that a continuous function f has a root in an interval $[a,b]$ if $f(a)$ and $f(b)$ are on the opposite side of the x-axis. By *continuous* we mean a function that doesn't "jump," that doesn't have gaps, and that proceeds on a "smooth" path.

We thank Dr. Neil Toronto for [plot](#).

[Figure 117](#) illustrates the Intermediate Value Theorem in a graphical manner. The function f is below the x-axis at a and above the x-axis at b . It is a continuous function, as suggested by the uninterrupted, smooth graph. And indeed, f intersects the x-axis somewhere between a and b , labeled "range 1" in the figure.

Now take a look at the midpoint between a and b :

$$m = (a+b) / 2 .$$

It partitions the interval $[a,b]$ into two smaller, equally large intervals. We can now compute the value of f at m and see whether it is below or above 0. Here $f(m) > 0$, so according to the Intermediate Value Theorem, the root is in the left interval: $[a,m]$. Our picture confirms this because the root is in the left half of the interval, labeled “range 2” in figure 117.

We now have a description of the key step in the root-finding process. The next step is to translate this description into a ISL+ algorithm. Our first task is to agree on the exact task of `find-root`. Clearly the algorithm consumes a function and the boundaries of the interval in which we expect to find a root:

```
; [Number -> Number] Number Number -> ...
(define (find-root f left right) ...)
```

The three parameters can't be just any function and numbers. For `find-root` to work, we must assume that

```
(or (<= (f left) 0 (f right))
    (<= (f right) 0 (f left)))
```

holds, that is, f 's values for `left` and `right` must be on opposite side's of the x-axis.

Next we need to fix the function's result and formulate a purpose statement. Simply put, `find-root` finds an interval that contains a root. The search divides the interval until its size, $(- right left)$, is tolerably small, say, smaller than some constant `TOLERANCE`. At that point, the function could produce one of three results: the left boundary, the right one, or a representation of the interval. Any one of them completely identifies the interval, and since it is simpler to return numbers, we pick the left boundary. Here is the complete header material:

```
; [Number -> Number] Number Number -> Number
; determine R such that f has a root in [R, (+ R TOLERANCE)]
; assume f is continuous
; assume (or (<= (f left) 0 (f right)) (<= (f right) 0 (f left)))
; generative divide interval in half, the root is in one of the two
; halves, pick according to assumption
(define (find-root f left right)
  0)
```

Exercise 448. Consider the following function definition:

```
; Number -> Number
(define (poly x)
  (* (- x 2) (- x 4)))
```

It defines a binomial for which we can determine its roots by hand:

```
> (poly 2)
0
> (poly 4)
0
```

Use `poly` to formulate a `check-satisfied` test for `find-root`.

Also use `poly` to illustrate the root-finding process. Start with the interval $[3,6]$ and tabulate the information as follows:

step	left	f(left)	right	f(right)	mid	f(mid)
n=1	3	-1	6.00	8.00	4.50	1.25
n=2	3	-1	4.50	1.25	?	?

Assume TOLERANCE is 0.5 for the construction of this table. ▀

Our next task is to address the four fundamental questions of algorithm design:

1. We need a condition that describes when the problem is solved and a matching answer.
Given our discussion so far, this is straightforward:

```
| (≤ (- right left) TOLERANCE)
```

2. The matching result in the trivial case is left.

3. For the generative case, we need an expression that generates new problems for find-root. According to our informal description, this step first requires determining the midpoint and its function value:

```
| (local ((define mid (/ (+ left right) 2))
         (define f@mid (f mid)))
  ...)
```

The midpoint is then used to pick the next interval. Following Intermediate Value Theorem, the interval [left,mid] is the next candidate if

```
| (or (≤ (f left) 0 f@mid) (≤ f@mid 0 (f left)))
```

while [mid,right] is used for the recursive call if

```
| (or (≤ f@mid 0 (f right)) (≤ (f right) 0 f@mid))
```

Translated into code, the body of the local expression must be a conditional:

```
| (local ((define mid (/ (+ left right) 2))
         (define f@mid (f mid)))
  (cond
    [(or (≤ (f left) 0 f@mid) (≤ f@mid 0 (f left)))
     (... (find-root f left mid) ...)]
    [(or (≤ f@mid 0 (f right)) (≤ (f right) 0 f@mid))
     (... (find-root f mid right) ...)]))
```

In both clauses, we use find-root to continue the search.

4. The answer to the final question is obvious. Since the recursive call to find-root finds the root of f, there is no need to process its solution any further.

The completed function is displayed in figure 118. The following exercises suggest some tests and a termination argument.

```
; [Number -> Number] Number Number -> Number
; determines R such that f has a root in [R, (+ R TOLERANCE)]
; assume f is continuous
; assume (or (≤ (f left) 0 (f right)) (≤ (f right) 0 (f left)))
; generative divide interval in half, the root is in one of the two
; halves, pick according to assumption
(define (find-root f left right)
  (cond
    [(≤ (- right left) TOLERANCE) left]
```

```
[else
  (local ((define mid (/ (+ left right) 2))
          (define f@mid (f mid)))
    (cond
      [(or (<= (f left) 0 f@mid) (<= f@mid 0 (f left)))
       (find-root f left mid)]
      [(or (<= f@mid 0 (f right)) (<= (f right) 0 f@mid))
       (find-root f mid right)])))]
```

Figure 118: The *find-root* algorithm

Exercise 449. Add the test from [exercise 448](#) to the program in [figure 118](#). Experiment with different values for TOLERANCE. ▀

Exercise 450. The `poly` function has two roots. Use `find-root` with `poly` and an interval that contains both roots. ▀

Exercise 451. The `find-root` algorithm terminates for all (continuous) `f`, `left`, and `right` for which the assumption holds. Why? Formulate a termination argument.

Hint Suppose the arguments of `find-root` describe an interval of size `S1`. How large is the distance between `left` and `right` for the first, second, third recursive call to `find-root`? After how many steps is `(- right left)` smaller than or equal to `TOLERANCE`? ▀

Exercise 452. As presented in [figure 118](#), `find-root` computes the value of `f` for each boundary value twice to generate the next interval. Use `local` to avoid this re-computation.

In addition, `find-root` recomputes the value of a boundary across recursive calls. For example, `(find-root f left right)` computes `(f left)` and, if $[left, mid]$ is chosen as the next interval, `find-root` computes `(f left)` again. Introduce a helper function that is like `find-root` but consumes not only `left` and `right` but also `(f left)` and `(f right)` at each recursive stage.

How many re-computations of `(f left)` does this design maximally avoid?

Note The two additional arguments to this helper function change at each recursive stage but the change is related to the change in the numeric arguments. These arguments are so-called *accumulators*, which are the topic of [Accumulators](#). ▀

Exercise 453. A function `f` is *monotonically increasing* if `(<= (f a) (f b))` holds whenever `(< a b)` yields `#true`. Simplify `find-root` assuming the given function is not only continuous but also monotonically increasing. ▀

Exercise 454. A table is a structure of two fields: the natural number `VL` and a function `array`, which consumes natural numbers and, for those between `0` and `VL` (exclusive), produces answers:

Many programming languages, including Racket, support arrays and vectors, which are similar to tables.

```
(define-struct table [length array])
; A Table is a
; (make-table N [N -> Number])
```

Since this data structure is somewhat unusual, it is critical to illustrate it with examples:

```
(define table1 (make-table 3 (lambda (i) i)))
```

```
; N -> Number
(define (a2 i)
  (if (= i 0) pi (error "table2 is not defined for i != 0")))

(define table2 (make-table 1 a2))
```

Here `table1`'s array function is defined for more inputs than its length field allows; `table2` is defined for just one input, namely `0`. Finally, we also define a useful function for looking up values in tables:

```
; Table N -> Number
; looks up the ith value in array of t
(define (table-ref t i)
  ((table-array t) i))
```

The root of a table `t` is the number in `(table-array t)` that is closest to `0`. A *root index* is a natural number `i` such that `(table-ref t i)` is a root of table `t`.

A table `t` is monotonically increasing if `(table-ref t 0)` is less than `(table-ref t 1)`, `(table-ref t 1)` is less than `(table-ref t 2)`, and so on.

Design `find-linear`. The function consumes a monotonically increasing table and finds the smallest index for a root of the table. Use the structural recipe for `N`, proceeding from `0` through `1`, `2`, and so on to the `array-length` of the given table. This kind of root-finding process is often called a *linear search*.

Design `find-binary`, which also finds the smallest index for root of a monotonically increasing table but uses generative recursion to do so. Like ordinary binary search, the algorithm narrows an interval down to the smallest possible size and then chooses the index. Don't forget to formulate a termination argument.

Hint The key problem is that a table index is a **natural** number, not a plain number. Hence the interval boundary arguments for `find` must be natural numbers. Consider how this observation changes (1) the nature of trivially solvable problem instances, (2) the midpoint computation, (3) and the decision which interval to generate next?

Consider `(make-table 1024 a)` and assume `(= (a 1023) 0)`. How many recursive calls to `find` are needed in `find-linear` and `find-binary` respectively? ■

31.3 A Glimpse at Parsing

As mentioned in [Incremental Refinement](#), computers come with files, which provide a form of permanent memory. From our perspective a *file* is just a list of `1String`s, though interrupted by a special string:

The exact convention differs from one operating system to another, but for our purposes this is irrelevant.

```
; A File is one of:
; - '()
; - (cons "\n" File)
; - (cons 1String File)
; interpretation "\n" represents the newline character
```

The idea is that *Files* are broken into lines, where "`\n`" represents the so-called newline

character, which indicates the end of a line. Let's also introduce lines before we move on:

; A Line is [List-of 1String]

Many programs need to process files as list of lines. For example, the file

```
(list "h" "o" "w" " " "a" "r" "e" " " "y" "o" "u" "\n"
      "d" "o" "i" "n" "g" "?" "\n"
      "a" "n" "y" " " "p" "r" "o" "g" "r" "e" "s" "s" "?")
```

might have to be processed as a list of three lines:

```
(list (list "h" "o" "w" " " "a" "r" "e" " " "y" "o" "u")
      (list "d" "o" "i" "n" "g" "?")
      (list "a" "n" "y" " " "p" "r" "o" "g" "r" "e" "s" "s" "?"))
```

Similarly, the file

```
(list "a" "b" "c" "\n"
      "d" "e" "\n"
      "f" "g" "h" "\n")
```

also corresponds to a list of three lines:

```
(list (list "a" "b" "c")
      (list "d" "e")
      (list "f" "g" "h"))
```

Stop! What are the list-of-lines representation for these three cases: '(), (list "\n"), and (list "\n" "\n")?. Why are these examples important test cases?

The problem of turning a sequence of 1Strings into a list of lines is called the *parsing* problem. Many programming languages provide functions that retrieve lines, words, numbers and other kinds of so-called tokens from files. But even if they do, it is common that programs need to parse these tokens even further. This section provides a glimpse at a parsing technique. Parsing is so complex and so central to the creation of full-fledged software applications, however, that most undergraduate curricula come with at least one course on parsing. So do not think you can tackle real parsing problems properly even after mastering this section.

We start by stating the obvious—a signature, a purpose statement, one of the above examples, and a header—for a function that turns a File into a list of Lines:

```
; File -> [List-of Line]
; converts a file into a list of lines

(check-expect (file->list-of-lines
                (list "a" "b" "c" "\n" "d" "e" "\n" "f" "g" "h" "\n"))
               (list (list "a" "b" "c")
                     (list "d" "e")
                     (list "f" "g" "h")))

(define (file->list-of-lines afile)
  '())
```

It is also easy to describe the parsing process, given our experience with [Recursion without Structure](#):

1. The problem is trivially solvable if the file is '()'.
2. In that case, the file doesn't contain a line.
3. Otherwise, the file contains at least "\n" or one 1String. These items—up to and including the first "\n", if any—must be separated from the rest of the File. The remainder is a new problem of the same kind that file->list-of-lines can solve.
4. It then suffices to cons the initial segment as a single line to the list of Lines that result from the rest of the File.

The four questions suggest a straightforward instantiation of the template for generative recursive functions. Because the separation of the initial segment from the rest of the file requires a scan of an arbitrarily long list of 1Strings, we put two auxiliary functions on our wish list: first-line, which collects all 1Strings up to, but excluding, the first occurrence of "\n" or the end of the list; and remove-first-line, which removes the very same items that first-line collects.

```

; File -> [List-of Line]
; converts a file into a list of lines

(check-expect (file->list-of-lines '("\n" "\n"))
               '(() ()))

(check-expect (file->list-of-lines
              (list "a" "b" "c" "\n" "d" "e" "\n" "f" "g" "h" "\n"))
               (list (list "a" "b" "c")
                     (list "d" "e")
                     (list "f" "g" "h")))

(define (file->list-of-lines afile)
  (cond
    [(empty? afile) '()]
    [else
      (cons (first-line afile)
            (file->list-of-lines (remove-first-line afile))))]

; File -> Line
; retrieves the prefix of afile up to the first occurrence of NEWLINE
(define (first-line afile)
  (cond
    [(empty? afile) '()]
    [(string=? (first afile) NEWLINE) '()]
    [else (cons (first afile) (first-line (rest afile)))]))

; File -> Line
; drops the suffix of afile behind the first occurrence of NEWLINE
(define (remove-first-line afile)
  (cond
    [(empty? afile) '()]
    [(string=? (first afile) NEWLINE) (rest afile)]
    [else (remove-first-line (rest afile))]))

(define NEWLINE "\n")

```

Figure 119: Translating a file into a list of lines

From here, it is easy to create the rest of the program. In `file->list-of-lines`, the answer in the first clause must be `'()` because an empty file does not contain any lines. The answer in the second clause must `cons` the value of `(first-line afile)` onto the value `(file->list-of-lines (remove-first-line afile))`, because the first expression computes the first line and the second one computes the rest of the lines. Finally, the auxiliary functions process their inputs in a structurally recursive manner; their development is a straightforward exercise. [Figure 119](#) collects the three function definitions and the definition for `NEWLINE`.

Here is how `file->list-of-lines` processes the second test:

```
(file->list-of-lines (list "a" "b" "c" "\n" "d" "e" "\n" "f" "g" "h" "\n"))
==  

(cons (list "a" "b" "c")
      (file->list-of-lines (list "d" "e" "\n" "f" "g" "h" "\n")))
==  

(cons (list "a" "b" "c")
      (cons (list "d" "e")
            (file->list-of-lines (list "f" "g" "h" "\n"))))
==  

(cons (list "a" "b" "c")
      (cons (list "d" "e")
            (cons (list "f" "g" "h")
                  (file->list-of-lines '()))))
==  

(cons (list "a" "b" "c")
      (cons (list "d" "e")
            (cons (list "f" "g" "h")
                  '())))
==  

(list (list "a" "b" "c")
      (list "d" "e")
      (list "f" "g" "h"))
```

This evaluation is another reminder that the argument of the recursive application of `file->list-of-lines` is almost never the rest of the given file. It also shows why this generative recursion is guaranteed to terminate for every given `File`. Every recursive application consumes a list that is shorter than the given one, meaning the recursive process stops when the process reaches `'()`.

Exercise 455. Design the function `tokenize`. It turns a `Line` into a list of tokens. Here a token is either a `1String` or a `String` that consists of lower-case letters and nothing else. That is, all white-space `1Strings` are dropped; all other non-letters remain as is; and all consecutive letters are bundled into “words.” **Hint** Read up on the `string-whitespace?` function. ■

Exercise 456. Design `create-matrix`. The function consumes a number n and a list of n^2 numbers. It produces a list of n lists of n numbers, for example:

```
(check-expect
  (create-matrix 2 (list 1 2 3 4))
  (list (list 1 2)
        (list 3 4)))
```

Make up a second example. ■

32 Mathematical Examples

Many solutions to mathematical problems employ generative recursion. A future programmer must get to know such solutions for two reasons. On one hand, a fair number of programming tasks are essentially about turning these kinds of mathematical ideas into programs. On the other hand, practicing with such mathematical problems often proves inspirational for the design of algorithms. This chapter deals with three such problems.

32.1 Newton's Method

[Binary Search](#) introduces one method for finding the root of a mathematical function. As the exercises in the same section sketch, the method naturally generalizes to computational problems, such as finding certain values in tables, vectors, and arrays. In mathematical applications, programmers tend to employ methods that originate from analytical mathematics. A prominent one is due to Newton. Like binary search, the so-called *Newton method* repeatedly improves an approximation to the root until it is “close enough.” Starting from a guess, say, r_1 , the essence of the process is to construct the tangent of f at r_1 and to determine its root. While the tangent approximates the function, it is also straightforward to determine its root. By repeating this process sufficiently often, an algorithm can find a root r for which $(f r)$ is close enough to 0.

Newton proved this fact.

Clearly this process relies on two pieces of domain knowledge

concerning tangents: their slopes and their roots. Informally, a tangent of f at some point r_1 is the line that goes through the point $(r_1, f(r_1))$ and has the same slope as f . One mathematical way to obtain the tangent’s slope is to pick two close points on the x-axis that are equidistant from r_1 and to use the slope of the line determined by f at those two points. The convention is to choose a small number ϵ and to work with $r_1 + \epsilon$ and $r_1 - \epsilon$. That is, the points are $(r_1 - \epsilon, f(r_1 - \epsilon))$ and $(r_1 + \epsilon, f(r_1 + \epsilon))$, which determines a line and a slope:

$$\text{slope} = \frac{f(r_1 + \epsilon) - f(r_1 - \epsilon)}{(r_1 + \epsilon) - (r_1 - \epsilon)} = \frac{f(r_1 + \epsilon) - f(r_1 - \epsilon)}{2 \cdot \epsilon}$$

Stop! Solve the next exercise.

Exercise 457. Translate this mathematical formula into the ISL+ function `slope`, which maps function `f` and a number `x` to the slope of `f` at `x`. Assume that `EPSILON` is a global constant. For your examples, use functions whose exact slope you can figure out, say, horizontal lines, linear functions, and perhaps polynomials if you know some calculus. ■

The second piece of domain knowledge concerns the root of a tangent, which is just a line or a linear function. The tangent goes through $(r_1, f(r_1))$ and has the above `slope`.

Mathematically, it is defined as

$$\text{tangent}(x) = \text{slope} \cdot x + f(r_1)$$

Finding the root of this tangent also means finding a value `root-of-tangent` so that the function value equals 0:

$$0 = \text{slope} \cdot \text{root-of-tangent} + f(r_1)$$

In contrast to arbitrary functions, though, we can solve this equation in a straightforward manner:

$$\text{root-of-tangent} = r_1 - \frac{f(r_1)}{\text{slope}}$$

Stop! Solve the next exercise.

Exercise 458. Translate this mathematical formula into the function `root-of-tangent`, which maps function `f` and a number `x` to the root of the tangent through $(x, (f \ x))$. You need to solve [exercise 457](#) first and reuse the solution here. ▀

Now we can use the design recipe to translate the description of Newton's process into an ISL+ program. The function—let's call it `newton` in honor of its inventor—consumes a function `f` and a number `r1`, the current guess:

```
; [Number -> Number] Number -> Number
; finds a number r such that (f r) is small
; generative repeatedly generate improved guesses using f and r

(define (newton f r1)
  1.0)
```

For the template of `newton`, we turn to the central four questions of the design recipe for generative recursion:

1. If $(f \ r1)$ is close enough to `0`, the problem is solved. Close to `0` could be mean $(f \ r1)$ is a small positive number or a small negative number. Hence we translate this idea into

```
| (<= (abs (f r1)) EPSILON)
```

That is, we determine whether the absolute value is small.

2. The solution is `r1`.
3. The generative step of the algorithm consists of finding the root of the tangent of `f` at `r1`, which generates the next guess. By applying `newton` to `f` and this new guess, we resume the process.
4. The answer of the recursion is also the answer of the original problem.

With these in place, putting together the function is a matter of formulating formulas as ISL+ code.

```
; [Number -> Number] Number -> Number
; finds a number r such that (<= (abs (f r)) EPSILON)

(check-within (newton poly 1) 2 EPSILON)
(check-within (newton poly 3.5) 4 EPSILON)

(define (newton f r1)
  (cond
    [(<= (abs (f r1)) EPSILON) r1]
    [else (newton f (root-of-tangent f r1))]))

; see exercise 457
(define (slope f r)
  1.0)

; see exercise 458
(define (root-of-tangent f r)
  1.0)
```

Figure 120: The Newton process

Figure 120 displays the definition of newton. It includes two tests that are derived from the tests in [Binary Search](#) for the find-root function. After all, both functions search for the root of a function and poly has two known roots.

We are not finished with the design of newton. The new, seventh step of the design recipe calls for an investigation into the termination behavior of the function. For newton, the problem shows up with poly already. Recall its definition:

```
; Number -> Number
(define (poly x)
  (* (- x 2) (- x 4)))
```

As mentioned and as figure 121 shows, its roots are 2 and 4. The figure also shows that between the two roots the function flattens out. For a mathematically inclined person, this shape almost immediately suggests the question what newton computes for an initial guess of 3:

```
> (poly 3)
-1
> (newton poly 3)
/: division by zero
```

The explanation is that slope produces a “bad” value and root-of-tangent reacts to it in its own bad way:

```
> (slope poly 3)
0
> (root-of-tangent poly 3)
/: division by zero
```

The newton function just propagates this error.

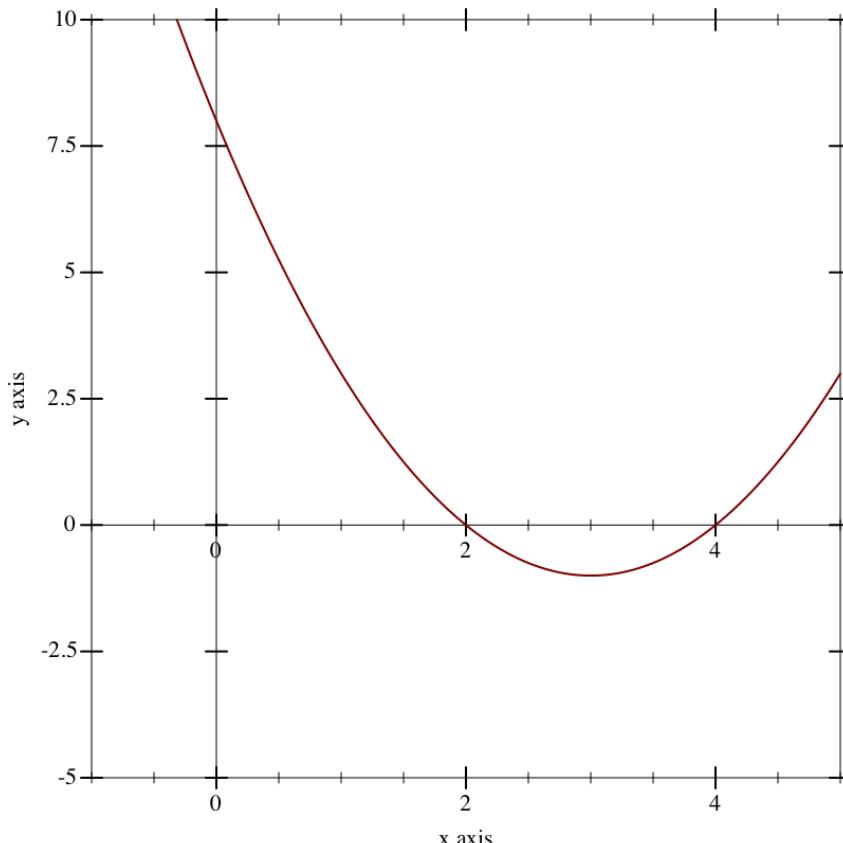


Figure 121: A plot of poly on the interval [-1,5]

In addition to this run-time error, `newton` exhibits two other problems with respect to termination. Fortunately, we can demonstrate both with `poly`. The first one concerns the nature of numbers, which we briefly touched on in [The Arithmetic of Numbers](#). It is safe to ignore the distinction between exact and inexact numbers for many beginner exercises in programming but when it comes to translating mathematics into programs, you need to proceed with extreme caution. Consider the following:

```
|> (newton poly 2.9999)
```

An ISL+ program treats `2.9999` as an exact number and the computations in `newton` process it as such, though because the numbers aren't integers, the computation uses exact rational fractions. Since the arithmetic for fractions can get much slower than the arithmetic for inexact numbers, the above function call takes a significant amount of time in DrRacket. Depending on your computer, it may take between a few second and a minute or more. If you happen to choose other numbers that trigger this form of computation, it may seem as if the call to `newton` does not terminate at all.

The second problem concerns non-termination. Here is the example:

```
|> (newton poly #i3.0)
```

It uses the inexact number `#i3.0` as the initial guess, which unlike `3` causes a different kind of problem. Specifically, the `slope` function now produces an inexact `0` for `poly` while `root-of-tangent` jumps to infinity:

```
|> (slope poly #i3.0)
#i0.0
|> (root-of-tangent poly #i3.0)
#i+inf.0
```

As a result, the evaluation immediately falls into an infinite loop.

In short, `newton` exhibits the full range of complex termination behavior. For some inputs, the function produces a correct result. For some others, it signals errors. And for yet others, it goes into infinite loop or appears to go into one. The header for `newton`—or some other piece of writing—must warn others who wish to use the function and future readers of these complexities, and good math libraries in common programming languages do so.

The calculation in `newton` turns `#i+inf.0` into `+nan.0`, a piece of hardware data that says “not a number.” It then continues to use this guess.

Exercise 459. Design the function `double-amount`, which computes how many months it takes to double a given amount of money when a savings account pays interest at a fixed rate on a monthly basis.

Hint The function must know the current amount and the initially given one; see ... [Add Expressive Power](#).

This exercise is due to Adrian German.

Domain Knowledge With a minor algebraic manipulation, you can show that the given amount is irrelevant. Only the interest rate matters. Also domain experts know that doubling occurs after roughly $72/r$ as long as the interest rate r is not large. ▀

32.2 Numeric Integration

Many problems in physics boil down to determining the area under a curve:

Sample Problem: A car drives at constant speed of v meters per second. How far does it travel in 5, 10, 15 seconds?

A rocket lifts off at the constant rate of acceleration of $12m/s^2$. What height does it reach after 5, 10, 15 seconds?

Physics tells us that a vehicle travels $d = v \cdot t$ meters if it moves at a constant speed v . For vehicles that accelerate, the distance traveled depends on the square of the time t passed:

$$\frac{1}{2} \cdot a \cdot t^2$$

In general, the law tells us that the distance corresponds to the area under the graph of speed $v(t)$ over time t .

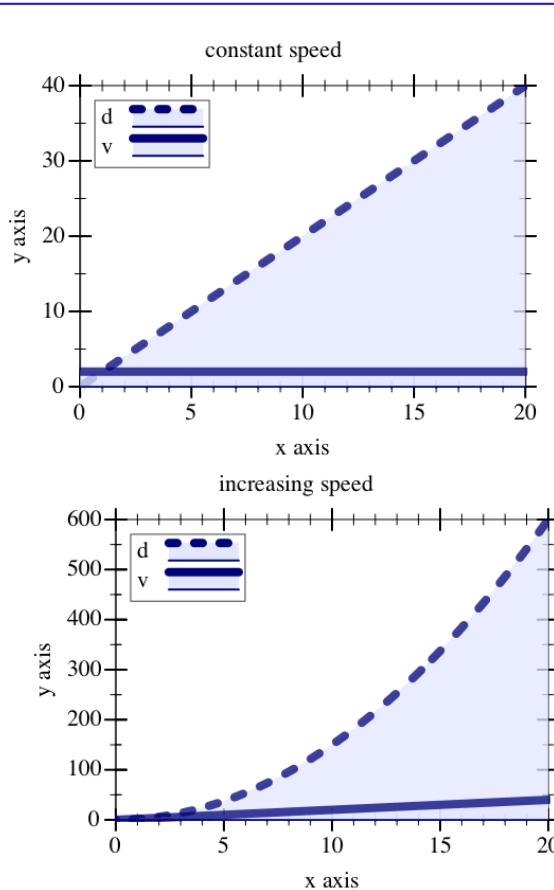


Figure 122: Distance traveled with constant vs accelerating speed

Figure 122 illustrates the idea in a graphical manner. On the left, we see an overlay of two graphs: the solid flat line is the speed of the vehicle and the rising dashed line is the distance traveled. A quick check shows that the latter is indeed the area determined by the former and the x-axis at **every point in time**. Similarly, the graphs on the right show the relationship between a rocket moving at constantly increasing speed and the height it reaches. Determining this area under the graph of a function for some specific interval is called (function) *integration*.

While mathematicians know formulas for the two sample problems that give precise answers, the general problem calls for computational solutions. The problem is that curves often come with complex shapes, more like those in figure 123, which suggests that someone needs to know the area between the x-axis, the vertical lines labeled a and b , and the graph of f . Applied mathematicians determine such areas in an approximate manner, summing the areas of many small geometric shapes. It is therefore natural to develop

algorithms that deal with these calculations.

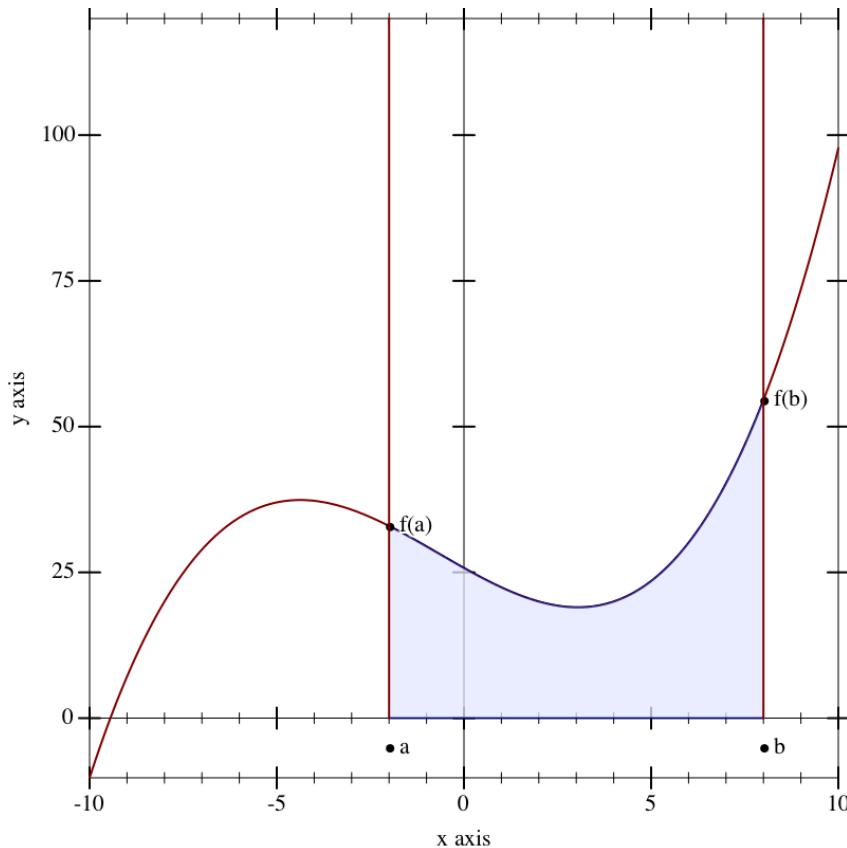


Figure 123: Integrating a function f between a and b

An integration algorithm consumes three inputs: the function f and two borders, a and b . The fourth part, the x -axis, is implied. This suggests the following signature:

```
| ; [Number -> Number] Number Number -> Number
```

In order to understand the idea behind integration, it is best to study simple examples such as a constant function or a linear one. Thus, consider

```
| (define (constant x) 20)
```

Passing `constant` to `integrate`, together with `12` and `22`, describes a rectangle of width `10` and height `20`. The area of this rectangle is `200`, meaning we get this test:

```
| (check-expect (integrate constant 12 22) 200)
```

Similarly, let's use `linear` to create a second test:

```
| (define (linear x) (* 2 x))
```

If we use `linear`, `0`, and `10` with `integrate`, the area is a triangle with a base width of `10` and a height of `20`. Here is the example reformulated as a test:

```
| (check-expect (integrate linear 0 10) 100)
```

After all, a triangle's area is half of the product of its base width and height.

For a third example, we exploit some domain-specific knowledge. As mentioned, mathematicians know how to determine the area under some functions in a precise manner. For example, the area under the function

$$\text{square}(x) = 3 \cdot x^2$$

on the interval $[a,b]$ can be calculated with the following formula

$$b^3 - a^3.$$

Here is how to turn this idea into a concrete test:

```
(define (square x) (* 3 (sqr x)))

(check-expect (integrate square 0 10) (- (expt 10 3) (expt 0 3)))

(define EPSILON 0.1)

; [Number -> Number] Number Number -> Number
; computes the area under the graph of f between a and b
; assume (< a b) holds

(check-within (integrate (lambda (x) 20) 12 22) 200 EPSILON)
(check-within (integrate (lambda (x) (* 2 x)) 0 10) 100 EPSILON)
(check-within (integrate (lambda (x) (* 3 (sqr x))) 0 10) 1000 EPSILON)

(define (integrate f a b)
#i0.0)
```

Figure 124: A generic integration function

Figure 124 collects the result of the first three steps of the design recipe. The figure adds a purpose statement and an obvious assumption concerning the two interval boundaries. Instead of `check-expect` it uses `check-within`, which anticipates the numerical inaccuracies that comes with computational approximations in such calculations. Analogously, the header of `integrate` specifies `#i0.0` as the return result, signaling that the function is expected to return an inexact number.

The following two exercises show how to turn domain knowledge into integration functions. Both functions compute rather crude approximations. While the design of the first uses only mathematical formulas, the second also exploits a bit of structural design ideas. Solving these exercises creates the necessary appreciation for the core of this section, which presents a generative-recursive integration algorithm.

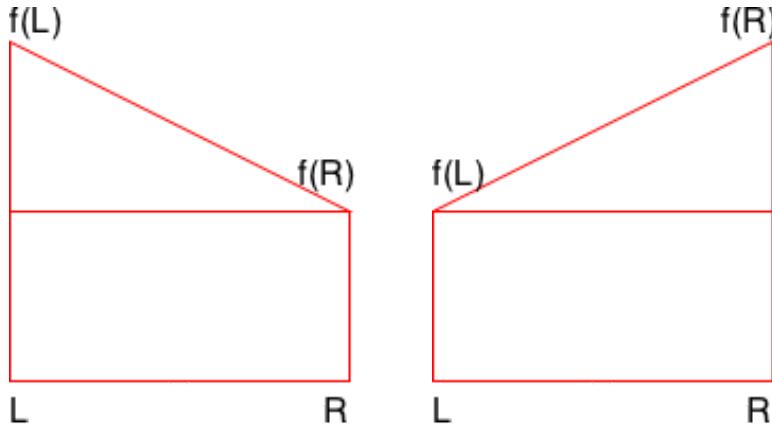
Exercise 460. Kepler suggested a simple integration method. To compute a rough estimate of the area under f between a and b , proceed as follows:

1. divide the interval into half at $mid = (a + b) / 2$;
2. compute the areas of the two *trapezoids*, each determined by four points:
 - o $[(a, 0), (a, f(a)), (mid, 0), (mid, f(mid))]$
 - o $[(mid, 0), (mid, f(mid)), (b, 0), (b, f(b))]$;
3. then add the two areas.

The method is known as *Kepler's rule*.

Design the function `integrate-kepler`. That is, turn the mathematical knowledge into a ISL+ function. Make sure to adapt the test cases from figure 124 to this use. Which of the three tests fails and by how much?

Domain knowledge Let us take a close look at the kind of trapezoids whose area you need to compute. Here are the two basic shapes, without a coordinate system to reduce clutter:



The left assumes $f(L) > f(R)$ while the right one shows the case where $f(L) < f(R)$.

Surprisingly, it is possible to calculate the area of these trapezoids with a single formula:

$$[(R - L) \cdot f(R)] + [\frac{1}{2} \cdot (R - L) \cdot (f(L) - f(R))]$$

Stop! Convince yourself that this formula **adds** the area of the triangle to the area of the lower rectangle for the left trapezoid above, while it **subtracts** the area of the triangle from the area of the large rectangle for the right one.

Also show that the above formula is equal to

$$\frac{(R - L) \cdot (f(L) + f(R))}{2}$$

This is a mathematical way to convince yourself of the asymmetry of the formula. ▀

Exercise 461. Another simple integration method divides the area into many small rectangles. Each rectangle has a fixed width and is as tall as the function graph in the middle of the rectangle. Adding up the areas of the rectangles produces an estimate of the area under the function's graph.

Let's use

$$R = 10$$

to stand for the number of rectangles to be considered. Hence the width of each rectangle is

$$W = \frac{(b-a)}{R} .$$

For the height of one of these rectangles, we determine the value of f at its midpoint. The first midpoint is clearly at a plus half of the width of the rectangle,

$$S = \frac{\text{width}}{2} ,$$

which means its area is

$$W \cdot f(a + S) .$$

To compute the area of the second rectangle, we must add the width of one rectangle to the first midpoint:

$$W \cdot f(a + W + S) ,$$

For the third one, we get

$$W \cdot f(a + 2 \cdot W + S) .$$

In general, we can use the following formula for the i th rectangle:

$$W \cdot f(a + i \cdot W + S) .$$

The first rectangle has index 0, the last one $R - 1$.

Using this sequence of rectangles, we can now determine the area under the graph as a series:

$$\begin{aligned} \sum_{i=0}^{i=R-1} W \cdot f(a + i \cdot W + S) &= W \cdot f(a + 0 \cdot W + S) \\ &\quad + \\ &\quad \dots \\ &\quad + \\ &\quad \dots \\ &\quad W \cdot f(a + (R - 1) \cdot W + S). \end{aligned}$$

Design the function `integrate-rectangles`. That is, turn the description of the rectangle process an ISL+ function. Make sure to adapt the test cases from [figure 124](#) to this use.

The more rectangles the algorithm uses, the closer its estimate is to the actual area. Make `R` a top-level constant and increase it by factors of `10` until the algorithm's accuracy eliminates problems with `EPSILON` value of `0.1`.

Decrease `EPSILON` to `0.01` and increase `R` enough to eliminate any failing test cases again. Compare the result to [exercise 460](#). ■

The Kepler method of [exercise 460](#) immediately suggests a divide-and-conquer strategy like binary search introduced in [Binary Search](#). Roughly speaking, the algorithm would split the interval into two pieces, recursively compute the area of each piece, and add the two results.

Exercise 462. Develop the algorithm `integrate-dc`, which integrates a function `f` between the boundaries `a` and `b` using a divide-and-conquer strategy. Use Kepler's method when the interval is sufficiently small. ■

The plain divide-and-conquer approach of [exercise 462](#) is wasteful. Consider a function whose graph is level in one part and rapidly changes in another; see [figure 125](#) for a concrete example. For the level part on the graph, it is pointless to keep splitting the interval. It is just as easy to compute the trapezoid for the complete interval as for the two halves. For the wavy part, however, the algorithm must continue dividing the interval until the irregularities of the graph are reasonably small.

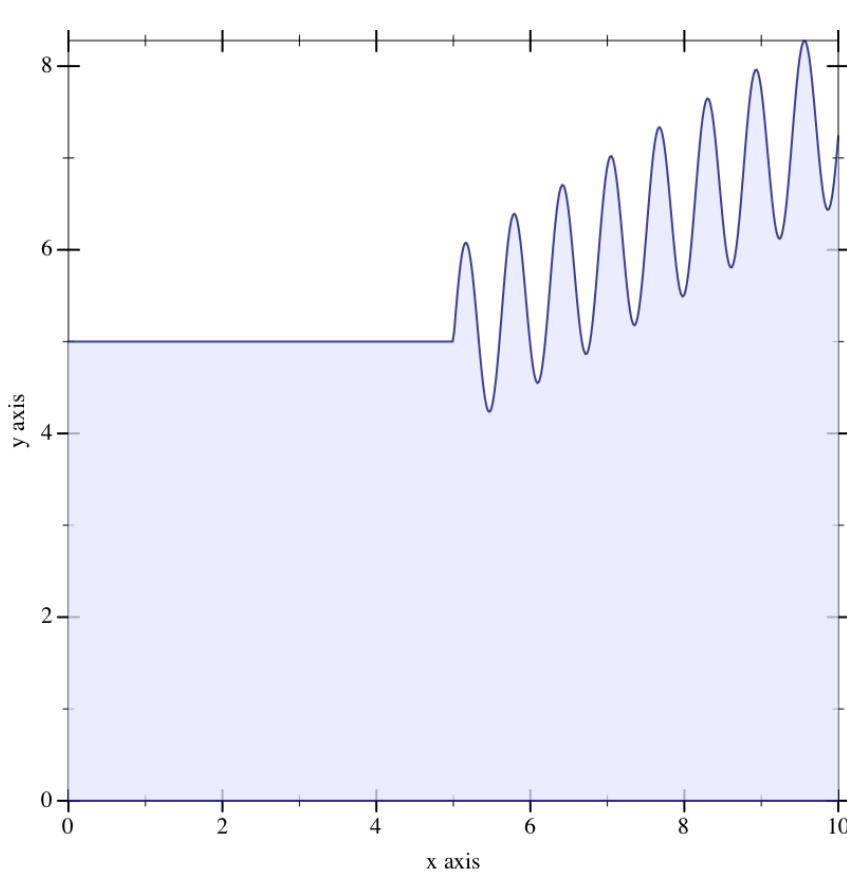


Figure 125: A candidate for adaptive integration

To discover when f is level, we can change the algorithm as follows. Instead of just testing how large the interval is, the new algorithm computes the area of three trapezoids: the given one, and the two halves. If the difference between the two is less than the area of a small rectangle of height ε and width $b - a$,

$$\cdot (b - a)$$

it is safe to assume that the overall area is a good approximation. In other words, the algorithm determines whether f changes so much that it affects the error margin. If so, it continues with the divide-and-conquer approach; otherwise it stops and uses the Kepler approximation.

Exercise 463. Design `integrate-adaptive`. That is, turn the recursive process description into an ISL+ algorithm. Make sure to adapt the test cases from [figure 124](#) to this use.

Do not discuss the termination of `integrate-adaptive`

Does `integrate-adaptive` necessarily compute a better answer than either `integrate-kepler` or `integrate-rectangles`? Which aspect is `integrate-adaptive` guaranteed to improve? ■

Terminology The algorithm is called *adaptive integration* because it automatically allocates time to those parts of the graph that needs it and spends little time on the others. Specifically, for those parts of f that are level, it performs just a few calculations; for the other parts, it inspects small intervals to decrease the error margin. Computer science knows many adaptive algorithms, and `integrate-adaptive` is just one of them.

32.3 Project: Gaussian Elimination

Mathematicians not only search for solutions of equations in one variable; they also study whole systems of linear equations:

Sample Problem: In a bartering world, the values of coal (x), oil (y), and natural gas (z) are determined by the following exchange equations:

$$\begin{aligned} 2 \cdot x + 2 \cdot y + 3 \cdot z &= 10 \\ 2 \cdot x + 5 \cdot y + 12 \cdot z &= 31 \quad (\dagger) \\ 4 \cdot x + 1 \cdot y - 2 \cdot z &= 1 \end{aligned}$$

A solution to such a system of equations consists of a collection of numbers, one per variable, such that if we replace the variable with its corresponding number, the two sides of each equation evaluate to the same number. In our running example, the solution is

$$x = 1, y = 1, \text{ and } z = 2.$$

We can easily check this claim:

$$\begin{aligned} 2 \cdot 1 + 2 \cdot 1 + 3 \cdot 2 &= 10 \\ 2 \cdot 1 + 5 \cdot 1 + 12 \cdot 2 &= 31 \\ 4 \cdot 1 + 1 \cdot 1 - 2 \cdot 2 &= 1 \end{aligned}$$

The three equations reduce to

$$10 = 10, 31 = 31, \text{ and } 1 = 1.$$

```
; An SOE is a non-empty Matrix
; constraint if its length is n (in N), each item has length (+ n 1)
; interpretation an SOE represents a system of linear equations

; An Equation is [List-of Number]
; constraint an Equation contains at least two numbers.
; interpretation if (list a1 ... an b) is an Equation, a1, ..., an are
; the left-hand side variable coefficients and b is the right-hand side

; A Solution is [List-of Number]

; examples:
(define M
  (list (list 2 2 3 10)
        (list 2 5 12 31)
        (list 4 1 -2 1)))

(define S '(1 1 2))
```

Figure 126: A data representation for systems of equations

Figure 126 introduces a data representation for our problem domain. It also illustrates how to represent the sample system of equations and its solution with this data representation. This representation captures the essence of a system of equations, namely, the numeric coefficients of the variables on the left-hand side and the right-hand side values. The names of the variables don't play any role because they are like parameters of functions; meaning, as long as they are consistently renamed the equations have the same solutions.

For the rest of this section, and especially the exercises, it is convenient to use the following functions:

```
; Equation -> [List-of Number]
```

```

; extracts the left-hand side from a row in a matrix
(define (lhs M)
  (reverse (rest (reverse M)))))

; Equation -> Number
; extracts the right-hand side from a row in a matrix
(define (rhs M)
  (first (reverse M)))

```

Exercise 464. Design the function `check-solution`. It consumes an `SOE` and a `Solution`.

Its result is `#true` if plugging in the numbers from the `Solution` for the variables in the `Equations` of the `SOE` produces equal left-hand side values and right-hand side values; otherwise the function produces `#false`. Use `check-solution` to formulate tests with `check-satisfied`.

Hint Design the function `plug-in` first. It consumes the left-hand side of an `Equation` and a `Solution` and calculates out the value of the left-hand side when the numbers from the solution are plugged in for the variables. ■

Gaussian elimination is a standard method for finding solutions to systems of linear equations. It consists of two steps. The first step is to transform the system of equations into a system of different shape but with the same solution. The second step is to find solutions to one equation at a time. Here we focus on the first step because it is another interesting instance of generative recursion.

The first step of the Gaussian elimination algorithm is called “triangulation” because the result is a system of equations in the shape of a triangle. In contrast, the original system is a rectangle. To understand this terminology, take a look at this list, which represents the original system:

```
(list (list 2 2 3 10)
      (list 2 5 12 31)
      (list 4 1 -2 1))
```

Triangulation transforms this matrix into the following:

```
(list (list 2 2 3 10)
      (list 3 9 21)
      (list 1 2))
```

As promised, the shape of this system of equations is (roughly) a triangle. Stop! Solve the next exercise.

Exercise 465. Check that the following system of equations

$$\begin{array}{rcl} 2 \cdot x + 2 \cdot y + 3 \cdot z & = & 10 \\ 3 \cdot y + 9 \cdot z & = & 21 \\ 1 \cdot z & = & 2 \end{array} \quad (*)$$

has the same solution as the one labeled with (\dagger) . Do so by hand and with `check-solution` from [exercise 464](#). ■

The key idea of triangulation is to subtract the first `Equation` from the remaining ones. To subtract one `Equation` from another means to subtract the corresponding coefficients in the two `Equations`. With our running example, subtracting the first from the second equation yields the following matrix:

```
(list (list 2 2 3 10)
      (list 0 3 9 21)
      (list 4 1 -2 1))
```

The goal of these subtractions is to put a 0 into the first column of all but the first equation. For the third equation, getting a 0 into the first position means subtracting the first equation **twice** from the third one:

```
(list (list 2 2 3 10)
      (list 0 3 9 21)
      (list 0 -3 -8 -19))
```

Following convention, we drop the leading 0's from the last two equations:

```
(list (list 2 2 3 10)
      (list 3 9 21)
      (list -3 -8 -19))
```

Generally speaking, we first multiply each item in the first row with 2 and then subtract the result from the last row. As mentioned, these subtractions do not change the solution; that is, the solution of the original system is also the solution of the transformed one.

Exercise 466. Check that the following system of equations

Mathematics teaches how to prove such facts.
We use them.

$$\begin{array}{rcl} 2 \cdot x + 2 \cdot y + 3 \cdot z & = & 10 \\ 3 \cdot y + 9 \cdot z & = & 21 \\ -3 \cdot y - 8 \cdot z & = & -19 \end{array} \quad (\dagger)$$

has the same solution as the one labeled with (\dagger) . Again do so by hand and with `check-solution` from [exercise 464](#). ■

Exercise 467. Design `subtract`. The function consumes two [Equations](#) of equal length. It subtracts the second from the first, item by item, as many times as necessary to obtain an [Equation](#) with a 0 in the first position. Since the leading coefficient is known to be 0, `subtract` returns the rest of the list that results from the subtractions. ■

Now consider the rest of the [SOE](#):

```
(list (list 3 9 21)
      (list -3 -8 -19))
```

It is also an [SOE](#) and we can thus apply the same algorithm again. For our running example, this next subtraction step calls for subtracting the first [Equation -1](#) times from the second one—that is equivalent to adding the two equations. Doing so yields

```
(list (list 3 9 21)
      (list 1 2))
```

And the remainder of this list is a single equation and obviously cannot be simplified any further.

Exercise 468. Here is the data definition for triangular systems of equations:

```
; A TM is [List-of Equation]
; such that the Equations are of decreasing length:
;   n + 1, n, n - 1, ..., 2.
; interpretation represents a triangular matrix
```

Design the `triangulate` algorithm:

```
; SOE -> TM
; triangulates the given system of equations
(define (triangulate M)
  '(1 2))
```

Turn the above example into a test and spell out explicit answers for the four questions based on our loose description.

Do not yet deal with the termination step of the design recipe. ▀

Unfortunately, the solution to [exercise 468](#) occasionally fails to produce the desired triangular system. Consider the following representation of a system of equations:

```
(list (list 2 3 3 8)
      (list 2 3 -2 3)
      (list 4 -2 2 4))
```

Its solution is $x = 1$, $y = 1$, and $z = 1$.

The first step is to subtract the first row from the second and to subtract it twice from the last one, which yields the following matrix:

```
(list (list 2 3 3 8)
      (list 0 -5 -5)
      (list -8 -4 -12))
```

Next triangulation would focus on the rest of the matrix:

```
(list (list 0 -5 -5)
      (list -8 -4 -12))
```

but the first item of this matrix is `0`. Since it is impossible to divide by `0`, the algorithm signals an error via `subtract`.

To overcome this problem, we need to use another piece of knowledge from our problem domain. Mathematics tells us that switching equations in a system of equations does not affect the solution. Of course, as we switch equations, we must eventually find an equation whose leading coefficient is not `0`. Here we can simply swap the first two:

```
(list (list -8 -4 -12)
      (list 0 -5 -5))
```

From here we may continue as before, subtracting the first equation from the remaining one `0` times. The final triangular matrix is:

```
(list (list 2 3 3 8)
      (list -8 -4 -12)
      (list -5 -5))
```

Stop! Show that $x = 1$, $y = 1$, and $z = 1$ is still a solution to this system of equations.

Exercise 469. Revise the algorithm `triangulate` from [exercise 468](#) so that it rotates the equations first to find one with a leading coefficient that is not `0` before it subtracts the first equation from the remaining ones.

Does this algorithm terminate for all possible system of equations?

Hint The following expression rotates a non-empty list L:

```
 (append (rest L) (list (first L)))
```

Explain why.

Some systems of equations don't have a solution. Consider the following example:

$$\begin{aligned} 2 \cdot x + 2 \cdot y + 2 \cdot z &= 6 \\ 2 \cdot x + 2 \cdot y + 4 \cdot z &= 8 \\ 2 \cdot x + 2 \cdot y + 1 \cdot z &= 2 \end{aligned}$$

If you try to triangulate this system—by hand or with your solution from [exercise 469](#)—you discover that it yields an intermediate matrix all of whose equations start with 0:

$$\begin{aligned} 2 \cdot x + 2 \cdot y + 2 \cdot z &= 6 \\ 0 \cdot x + 0 \cdot y + 2 \cdot z &= 2 \\ 0 \cdot x + 0 \cdot y - 1 \cdot z &= -4 \end{aligned}$$

Exercise 470. Modify the `triangulate` function from [exercise 469](#) so that it signals an error if it encounters an `SOE` whose leading coefficients are all 0.

After we obtain a triangular system of equations such as (*) in [exercise 465](#), we can solve the equations, one at a time. In our specific example, the last equation says that z is 2. Equipped with this knowledge, we can eliminate z from the second equation through a substitution:

$$3 \cdot y + 9 \cdot 2 = 21 .$$

Doing so, in turn, determines the value for y :

$$y = \frac{21 - 9 \cdot 3}{3}$$

In other words, we now know that $z = 2$ and $y = 1$, knowledge that we can plug into in the first equation:

$$2 \cdot x + 2 \cdot 1 + 3 \cdot 2 = 10$$

And again, this yields an equation in a single variable, which we know how to solve:

$$x = \frac{10 - (2 \cdot 1 + 3 \cdot 2)}{2}$$

This finally yields a value for x and thus the complete solution for the entire system of equations.

Exercise 471. Design the `solve` function. It consumes triangular systems of equations and produces a solution.

Hint Use structural recursion for the design. Start with the design of a function that solves a single linear equation in $n+1$ variables, given a solution for the last n variables. In general, this function plugs in the values for the rest of the left-hand side, subtracts the result from the right-hand side, and divides by the first coefficient. Experiment with this suggestion and the above examples.

Challenge Use an existing abstraction and `Lambda` to define `solve` as a one-liner.

Exercise 472. Define the function `gauss`. It combines the `triangulate` function from [exercise 470](#) and the `solve` function from [exercise 471](#).

33 Algorithms that Backtrack

Problem solving isn't always straightforward. Sometimes we make progress by pursuing one approach only to discover that we are stuck because we took a wrong turn. One obvious option is to backtrack to the place where we took the wrong turn and to take a different turn. Some algorithms work just like that. This chapter presents two instances. The first section deals with an algorithm for traversing graphs. The second one is an extended exercise that uses backtracking in the context of a chess puzzle.

33.1 Traversing Graphs

Graphs are ubiquitous in our world and the world of computing. Imagine a group of people, say, the students in your school. Write down all the names and connect the names of those people that know each other. You have just created your first undirected graph.

Now take a look at [figure 127](#), which displays a small directed graph. It consists of seven nodes—the circled letters—and nine edges—the arrows. The graph may represent a small version of a so-called email network. Imagine a company and all the emails that go back and forth. Write down the email addresses of all employees. Then, address by address, draw an arrow from the address to all those addresses to whom the owner sends emails during a week. This is how you would create the directed graph in [figure 127](#), though it might end up looking much more complex, almost impenetrable.

In general, a *graph* consists of a collection of *nodes* and a collection of *edges*, which connect nodes. In a *directed graph*, the edges represent one-way connections between the nodes; in an *undirected graph*, the edges represent two way connections between the nodes. In this context, the following is a common type of problem:

Sample Problem: Design an algorithm that proposes a way to introduce one person to another in a directed email graph for a large company. The program consumes a directed graph representing established email connections and two email addresses. It returns a sequence of email addresses that connect the first email with the second.

Social scientists use such algorithms to figure out the power structure in a company from such email graphs. Similarly they can use such graphs to predict the probable activities of participants, even without knowledge of the content of the emails.

Mathematical scientists call the desired sequence a *path*.

[Figure 127](#) provides a basis for making the sample problem concrete. For example, you may wish to test whether the program can find a path from *C* to *D*. This particular path consists of the origination node *C* and the destination node *D*. In contrast, if you wish to connect *E* with *D*, there are two paths:

- send email from *E* to *F* and then to *D*.
- send it from *E* to *C* and then to *D*.

Sometimes it is impossible to connect two nodes with a path. In the graph of [figure 127](#), you cannot move from *C* to *G* by following the arrows.

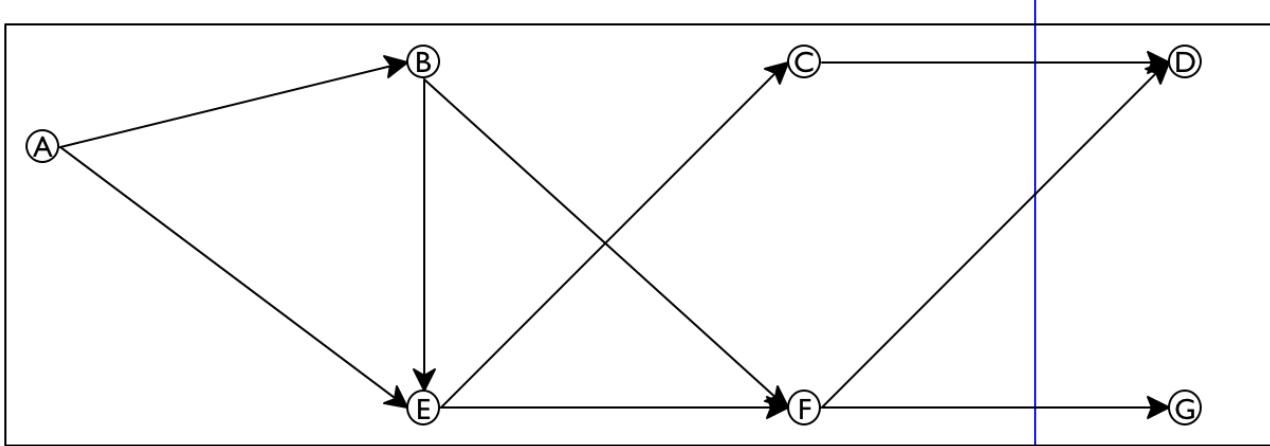


Figure 127: A directed graph

Looking at [figure 127](#) you can easily figure out how to get from one node to another without thinking much about how you did it. So imagine for a moment that the graph in [figure 127](#) is a large park. Also imagine someone says you are located at E and you need to get to G . You can clearly see two paths, one leading to C and another one leading to F . Follow the first one and make sure to remember that it is also possible to get from E to F . Now you have a new problem, namely, how to get from C to G . The key insight is that this new problem is just like the original problem; it asks you to find a path from one node to another. Furthermore, if you can solve the problem, you know how to get from E to G —just add the step from E to C . But there is no path from C to G . Fortunately, you remember that it is also possible to go from E to F , meaning you can *backtrack* to some point where you have a choice to make and re-start the search from there.

Now let's design this algorithm in a systematic manner. Following the general design recipe, we start with a data analysis. Here are two compact list-based representations of the graph in [figure 127](#):

```
(define sample-graph
  '((A (B E))
    (B (E F))
    (C (D))
    (D ())
    (E (C F))
    (F (D G))
    (G ())))
```

```
(define sample-graph
  '((A B E)
    (B E F)
    (C D)
    (D)
    (E C F)
    (F D G)
    (G)))
```

Both contain one list per node. Each of these lists starts with the name of a node followed by its (immediate) *neighbors*, that is, nodes reachable by following a single arrow. The two differ in how they connect the (name of the) node and its neighbors: the left one uses [list](#) while the right one uses [cons](#). For example, the second list represents node B with its two outgoing edges to E and F in [figure 127](#). On the left ' B ' is the first name on a two-element list; on the right it is the first name on a three-element list.

Exercise 473. Translate one of the above definitions into proper list form using [list](#) and proper symbols.

The data representation for nodes is straightforward:

; A Node is a Symbol

Formulate a data definition to describe the class of all *Graph* representations, allowing an arbitrary number of nodes and edges. Only one of the above representations has to belong to [Graph](#).

Design the function `neighbors`. It consumes a `Node` `n` and a `Graph` `g` and produces the list of immediate neighbors of `n` in `g`.

Using your data definitions for `Node` and `Graph`—regardless of which one you chose, as long as you also designed `neighbors`—we can now formulate a signature and a purpose statement for `find-path`, the function that searches a path in a graph:

```
; Node Node Graph -> [List-of Node]
; finds a path from origination to destination in G
(define (find-path origination destination G)
  '())
```

What this header leaves open is the exact shape of the result. It implies that the result is a list of nodes, but it does not say exactly which nodes the list contains.

To appreciate this ambiguity and why it matters, let's study the examples from above. In ISL+, we can now formulate them like this:

```
(find-path 'C 'D sample-graph)
(find-path 'E 'D sample-graph)
(find-path 'C 'G sample-graph)
```

The first call to `find-path` must return a unique path, the second one must choose one from two, and the third one must signal that there is no path from '`C`' to '`G`' in `sample-graph`. Here are two possibilities then on how to construct the return value:

- The result of the function consists of all nodes leading from the origination node to the destination node, including those two. In this case, an empty path could be used to express the lack of a path between two nodes.

It is easy to imagine others, such as skipping either of the two given nodes.
- Alternatively, since the call itself already lists two of the nodes, the output could mention only the “interior” nodes of the path. Now the answer for the first call would be '`()`' because '`D`' is an immediate neighbor of '`C`'. Of course, this also means that '`()`' no longer signals failure.

Concerning the lack-of-a-path issue, we must choose a distinct value for signaling this notion. Because `#false` is distinct, meaningful, and works in either case, we opt for it. As for the multiple-paths issue, we postpone making a choice for now and list both possibilities in the example section:

```
; A Path is [List-of Node]
; interpretation The list of nodes specifies a sequence of
; immediate neighbors that leads from the first Node on the
; list to the last one.

; Node Node Graph -> [Maybe Path]
; finds a path from origination to destination in G
; if there is no path, the function produces #false

(check-expect (find-path 'C 'D sample-graph) '(C D))
(check-member-of (find-path 'E 'D sample-graph) '(E F D) '(E C D))
(check-expect (find-path 'C 'G sample-graph) #false)

(define (find-path origination destination G)
```

```
| #false)
```

Our next design step is to understand the four essential pieces of the function: the “trivial problem” condition, a matching solution, the generation of a new problem, and the combination step. The above discussion of the search process and the analysis of the three examples suggest answers:

1. If the two given nodes are directly connected with an arrow in the given graph, the path consists of just these two nodes. But there is an even simpler case, namely, when the `origination` argument of `find-path` is equal to its `destination`.
2. In that second case, the problem is truly trivial and the matching answer is (`list destination`).
3. If the arguments are different, the algorithm must inspect all immediate neighbors of `origination` and determine whether there is a path from any one of those to `destination`. In other words, picking one of those neighbors generates a new instance of the “find a path” problem.
4. Finally, once the algorithm has a path from a neighbor of `origination` to `destination`, it is easy to construct a complete path from the former to the latter—just add the `origination` node to the list.

From a programming perspective, the third point is critical. Since a node can have an arbitrary number of neighbors, the “inspect all neighbors” task is too complex for a single primitive. We need an auxiliary function whose task it is to consume a list of nodes and to generate a new path problem for each of them. Put differently, the function is a list-oriented version of `find-path`.

Let’s call this auxiliary function `find-path/list` and let’s formulate a wish:

```
| ; [List-of Node] Node Graph -> [Maybe Path]
| ; finds a path from some node on lo-originations to destination
| ; if there is no path, the function produces #false
| (define (find-path/list lo-originations destination G)
|   #false)
```

Using this wish, we can fill in the generic template for generative-recursive functions to get a first draft of `find-path`:

```
| (define (find-path origination destination G)
|   (cond
|     [(symbol=? origination destination) (list destination)]
|     [else (... origination ...
|               (find-path/list (neighbors origination G) destination G)
|               ...))])
```

It uses the `neighbors` from [exercise 473](#) and the wish list function `find-path/list` and otherwise uses the answers to the four questions about generative recursive functions.

The rest of the design process is about details of composing these functions properly. Consider the signature of `find-path/list`. Like `find-path`, it produces [\[Maybe Path\]](#). That is, if it finds a path from any of the neighbors, it produces this path; otherwise, if none of the neighbors is connected to `destination`, the function produces `#false`. Hence the answer of `find-path` depends on the kind of result `find-path/list` produces, meaning the code must distinguish the two possible answers with a `cond` expression:

```
(define (find-path origination destination G)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define next (neighbors origination G))
                  (define candidate (find-path/list next destination G)))
            (cond
              [(boolean? candidate) ...]
              [else ; candidate satisfies cons?
               ...]))]))
```

The two cases reflect the two kinds of answers we might receive: a boolean or a list. In the first case, `find-path/list` cannot find a path from any neighbor to `destination`, meaning `find-path` itself cannot construct such a path either. In the second case, the auxiliary function found a path, but `find-path` must still add `origination` to the front of this path because `candidate` starts with one of `origination`'s neighbors not `origination` itself as agreed upon above.

```
; Node Node Graph -> [Maybe Path]
; finds a path from origination to destination in G
; if there is no path, the function produces #false
(define (find-path origination destination G)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define next (neighbors origination G))
                  (define candidate (find-path/list next destination G)))
            (cond
              [(boolean? candidate) #false]
              [else (cons origination candidate)])))))

; [List-of Node] Node Graph -> [Maybe Path]
; finds a path from some node on lo-0s to D
; if there is no path, the function produces #false
(define (find-path/list lo-0s D G)
  (cond
    [(empty? lo-0s) #false]
    [else (local ((define candidate (find-path (first lo-0s) D G)))
            (cond
              [(boolean? candidate) (find-path/list (rest lo-0s) D G)]
              [else candidate]))]))
```

Figure 128: Finding a path in a graph

Figure 128 contains the complete definition of `find-path`. It also contains a definition of `find-path/list`, which processes its first argument via structural recursion. For each node in the list, `find-path/list` uses `find-path` to check for a path. If `find-path` indeed produces a path, that path is the answer. Otherwise, `find-path` produces `#false` and the function backtracks.

Note [Trees](#) discusses backtracking in the structural world. A particularly good example is the function that searches blue-eyed ancestors in a family tree. When the function encounters node, it first searches one branch of the family tree, say the father's, and if this search produces `#false`, it searches the other half. Since graphs generalize trees, comparing this function with `find-path` is an instructive exercise. **End**

Lastly, we need to check whether `find-path` produces an answer for all possible inputs. It is relatively easy to check that when given the graph in [figure 127](#) and any two nodes in this

graph, `find-path` always produces some answer. Stop! Solve the next exercise before you read on.

Exercise 474. Test `find-path`. Use the function to find a path from '`A`' to '`G`' in `sample-graph`. Which one does it find? Why?

Design the function `test-on-all-nodes`, which consumes a graph `g` and tries to find a path `find-path` between all pairs of nodes in `g`. If it succeeds, it produces `#true`. Test the function on `sample-graph`. ■

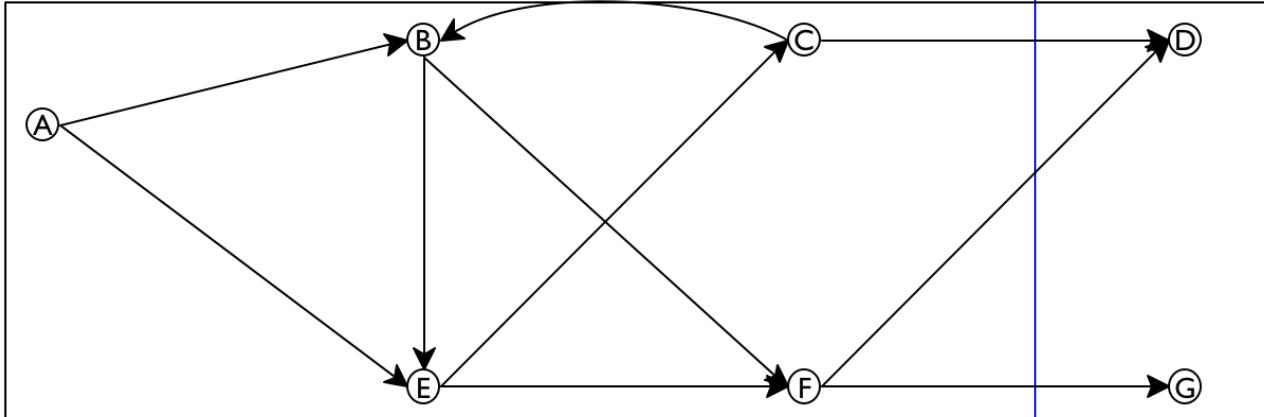


Figure 129: A directed graph with cycle

For other graphs, however, `find-path` may not terminate for certain pairs of nodes.

Consider the graph in [figure 129](#).

Stop! Define `cyclic-graph` to represent the graph in this figure.

Compared to [figure 127](#), this new graph contains only one extra edge, a connection from `C` to `B` though this seemingly small addition allows to start a search in a node and to return to the same node by just following the arrows. Specifically, it is possible to move from `B` to `E` to `C` and back to `B`. Indeed, when `find-path` is applied to '`B`', '`D`', and `cyclic-graph`, it fails to stop as the following hand-evaluation confirms:

```

(find-path 'B 'D cyclic-graph)
== ... (find-path 'B 'D cyclic-graph) ...
== ... (find-path/list (list 'E 'F) 'D cyclic-graph) ...
== ... (find-path 'E 'D cyclic-graph) ...
== ... (find-path/list (list 'C 'F) 'D cyclic-graph) ...
== ... (find-path 'C 'D cyclic-graph) ...
== ... (find-path/list (list 'B 'D) 'D cyclic-graph) ...
== ... (find-path 'B 'D cyclic-graph) ...
== ...
  
```

The hand-evaluation shows that after seven applications of `find-path` and `find-path/list`, ISL+ must evaluate the exact same expression from where it started. Since the same input triggers the same evaluation for any defined function, we now know that `find-path` does not terminate for these inputs.

You know only one exception to this rule:
[random](#).

In summary, the **termination** argument goes like this. If some given graph is free of cycles, `find-path` produces some output for any given inputs. After all, every path can only contain a finite number of nodes and the number of paths is finite, too.

The function therefore either exhaustively inspects all solutions starting from some given node or finds a path from the origination to the destination node. If, however, a graph contains a cycle, that is, a path from some node back to itself, `find-path` may not produce a result for some inputs.

The next part presents a program design technique that addresses just this kind of problem. In particular, it presents a variant of `find-path` that can deal with cycles in a graph.

Exercise 475. Test `find-path` on '`B`', '`C`', and the graph in figure 129. Also use `test-on-all-nodes` from exercise 474 on this graph. ▀

Exercise 476. Re-design the `find-path` program as a single function definition. Remove parameters from the locally defined functions. ▀

Exercise 477. Re-design `find-path/list` so that it uses an existing list abstraction from figure 71 instead of explicit structural recursion.

Once you have this version tested, merge the two functions.

Note Read the documentation for Racket's `ormap`. How does it differ from ISL+'s `ormap` function? Would the former be helpful here? ▀

Note on Data Abstraction You may have noticed that the `find-path` function does not need to know how `Graph` is defined. As long as you provide a correct `neighbors` function for `Graph`, `find-path` works perfectly fine. In short, the `find-path` program uses **data abstraction**.

In a way, data abstraction works just like function abstraction, as discussed in [Abstraction](#). Here you could create a function `abstract-find-path`, which would consume one more parameter than `find-path`: `neighbors`. As long as you always handed `abstract-find-path` a graph `G` from `Graph` and the matching `neighbors` function, it would process the graph properly. While the extra parameter suggests abstraction in the conventional sense, the required relationship between two of the parameters—`G` and `neighbors`—really means that `abstract-find-path` is also abstracted over the definition of `Graph`. Since the latter is a data definition, the idea is dubbed data abstraction.

When programs grow large, data abstraction becomes a critical tool for the construction of a program's components. The next volume in the "How to Design" series addresses this idea in depth; the next section illustrates the idea with another example. **End**

Exercise 478. [Finite State Machines](#) poses a problem concerning finite state machines and strings but immediately defers to this chapter because the solution calls for generative recursion. You have now acquired the design knowledge needed to tackle the problem.

Design the function `fsm-match`. It consumes the data representation of a finite state machine and a string. It produces `#true` if the sequence of characters in the string causes the finite state machine to transition from an initial state to a final state.

Since this problem is about the design of generative recursive functions, we provide the essential data definition and a data example:

```
(define-struct transition [current key next])
(define-struct fsm [initial transitions final])

; A FSM is (make-fsm FSM-State [List-of 1Transition] FSM-State)
; A 1Transition is
;   (make-transition FSM-State 1String FSM-State)
```

```
; A FSM-State is String

; data example: see exercise 112
(define fsm-a-bc*-d
  (make-fsm
    "AA"
    (list (make-transition "AA" "a" "BC")
          (make-transition "BC" "b" "BC")
          (make-transition "BC" "c" "BC")
          (make-transition "BC" "d" "DD")))
    "DD"))
```

The data example corresponds to the regular expression $a (b|c)^* d$. As mentioned in [exercise 112](#), "acbd" is one example of an acceptable string; "ad" and "abcd" are two others. Of course, "da", "aa", or "d" do not match.

In this context, you are designing the following function:

```
; FSM String -> Boolean
; does the given string match the regular expression expressed as fsm
(define (fsm-match? a-fsm a-string)
  #false)
```

Hint Design the necessary auxiliary function locally to `fsm-match?`. In this context, represent the problem as a pair of parameters: the current state of the finite state machine and the remaining list of `1String`s. ■

```
; [List-of X] -> [List-of [List-of X]]
; creates a list of all rearrangements of the items in w
(define (arrangements w)
  (cond
    [(empty? w) '(())]
    [else
      (foldr (lambda (item others)
                (local ((define without-item
                               (arrangements (remove item w)))
                        (define add-item-to-front
                          (map (lambda (a) (cons item a)) without-item)))
                  (append add-item-to-front others)))
              '()
              w)]))

; test
(define (all-words-from-rat? w)
  (and (member (explode "rat") w)
       (member (explode "art") w)
       (member (explode "tar") w)))

(check-satisfied (arrangements '("r" "a" "t")) all-words-from-rat?))
```

Figure 130: A compact definition of arrangements using generative recursion

Exercise 479. Inspect the function definition of `arrangements` in [figure 130](#). The figure displays a generative-recursive solution of the extended design problem covered by [Word Games, the Heart of](#)

We thank Mr. Mark Engelberg for suggesting this exercise. Also see [figure 80](#).

the Problem, namely

given a word, create all possible re-arrangements of the letters in a list.

The extended exercise is a direct guide to the structurally recursive design of the main function and two auxiliaries, where the design of the latter requires the creation of two more helper functions. In contrast, [figure 130](#) uses the power of generative recursion—plus `foldr` and `map`—to define the same program as a single function definition.

Explain the design of the generative-recursive version of arrangements. Answer all questions that the design recipe for generative recursion poses, including the question of termination.

Does arrangements in [figure 130](#) create the same lists as the solution of [Word Games, the Heart of the Problem?](#) |

33.2 Project: Backtracking

We thank Mr. Mark Engelberg for his suggestions on how to reformulate this section for young students.

The n queens puzzle is a famous problem from the world of chess that also illustrates the applicability of backtracking in a natural way. For our purposes, a chessboard is a grid of n by n squares. The queen is a game piece that can move in a horizontal, vertical, or diagonal direction arbitrarily far without “jumping” over another piece. We say that a queen *threatens* a square if it is on the square or can move to it. [Figure 131](#) illustrates the notion in a graphical manner. The queen is in the second column and sixth row. The solid lines radiating out from the queen go through all those squares that are threatened by the queen.

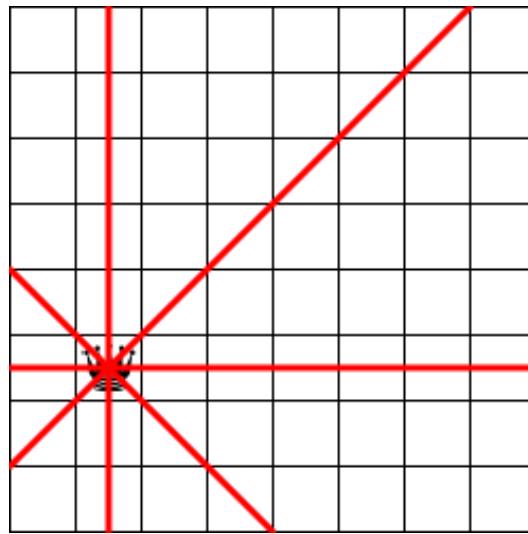


Figure 131: A chessboard with a single queen and the positions it threatens

The classical queen-placement problem is to place 8 queens on a 8 by 8 chessboard such that the queens on the board don't threaten each other. Computer scientists generalize the problem and ask whether it is possible to place n queens on a k by k , for $k \geq n$, chessboard such that the queens don't pose a threat to each other. When $k = n$, a solution solves the complete puzzle.

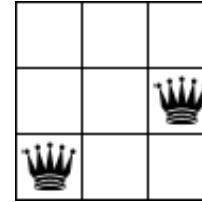
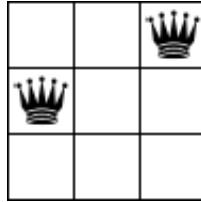
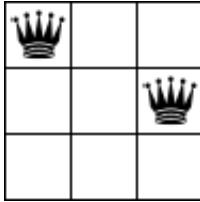


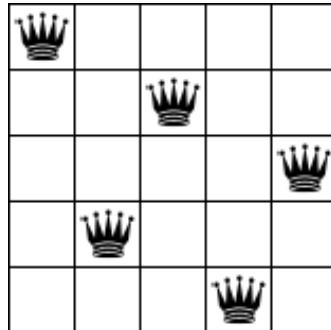
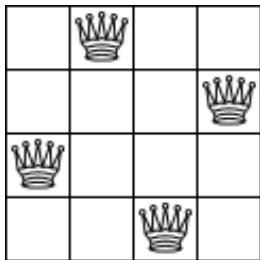
Figure 132: Three queen configurations for a 3 by 3 chess board

For $n = 2$, the complete puzzle obviously has no solution. A queen placed on any of the four squares threatens all remaining squares.

It turns out that there is also no complete solution for $n = 3$. [Figure 132](#) presents three different placements of two queens, that is, solutions for $k = 3$ and $n = 2$. In each case, the left-most queen occupies a square in the left-most column while a second queen is placed in one of two squares that the first one does not threaten. The placement of a second queen ensures that all seven unoccupied squares are threatened by some queen, meaning it is impossible to place a third queen. Together, the three placements explore all possibilities of placing the first queen in a square of the first column.

Exercise 480. It is also possible to place the first queen in all squares of the top-most row, the right-most column, and the bottom-most row. Explain why all of these solutions are just like the three scenarios depicted in [figure 132](#)?

This leaves the central square. Is it possible to place even a second queen after you place one on the central square of a 3 by 3 board? ■

Figure 133: Solutions for the n queens puzzle for 4 by 4 and 5 by 5 boards

[Figure 133](#) displays two solutions for the n queens puzzle: the left one is for $n = 4$, while the right one solves the $n = 5$ version. The figure shows how, in each case, a solution places one queen in each row and column of the board, which makes sense because a queen threatens the entire row and column that radiate out from its square.

Now that we have conducted a sufficiently detailed analysis of the problem, we can proceed to the solution phase. The analysis suggests several ideas:

1. The problem is about placing one queen at a time. When we place a queen on a board, we can mark the corresponding rows, columns, and diagonals as unusable for other

queens.

2. When we place another queen on a board, we consider only those spots that are not threatened.
3. Just in case this first choice of a spot leads to problems later, we remember what other squares are feasible for placing this queen.
4. If we are supposed to place a queen on a board but no safe squares are left, we backtrack to a previous point in the process where we chose one square over another and try one of the remaining squares.

In short, the solution process for the n queens puzzle has the same flavor as the “find a path” algorithm from [Traversing Graphs](#).

```
; N -> [Maybe [List-of QP]]
; find a solution to the n queens problem

(define (check-member-of
          (n-queens 4)
          (list (make-posn 0 1) (make-posn 1 3) (make-posn 2 0) (make-posn 3 2))
          (list (make-posn 0 1) (make-posn 1 3) (make-posn 3 2) (make-posn 2 0))
          (list (make-posn 0 1) (make-posn 2 0) (make-posn 1 3) (make-posn 3 2))
          (list (make-posn 0 1) (make-posn 2 0) (make-posn 3 2) (make-posn 1 3))
          (list (make-posn 0 1) (make-posn 3 2) (make-posn 1 3) (make-posn 2 0))
          (list (make-posn 0 1) (make-posn 3 2) (make-posn 2 0) (make-posn 1 3))
          (list (make-posn 0 2) (make-posn 1 0) (make-posn 2 3) (make-posn 3 1)))
          ...
          (list (make-posn 3 2) (make-posn 2 0) (make-posn 1 3) (make-posn 0 1)))

(define (n-queens n)
  (place-queens (board0 n) n))
```

Figure 134: Solutions for the 4 queens puzzle

Moving from the process description to a designed algorithm clearly calls for two data representations: one for the chess boards and one for positions on the board. Let’s start with the latter because the chess board basically dictates the choice:

```
(define QUEENS 8)

; QP is (make-posn CI CI)
; CI is a natural number in [0,QUEENS)
; interpretation a CI denotes a row or column index for a chess board,
; (make-posn r c) specifies the square in the r-th row and the c-th column
```

The definition for **CI** could use `[1,QUEENS]` instead of `[0, QUEENS)`, but the two definitions are basically equivalent and counting up from `0` is what programmers do. Similarly, the so-called algebraic notation for chess positions uses the letters `'a` through `'h` for one of the board’s dimension, meaning **QP** could have used **CIs** and such letters. Again, the two are roughly equivalent and with natural numbers it is easier in ISL+ to create many positions than with letters.

Exercise 481. Design the `threatening?` function. It consumes two **QPs** and determines whether queens placed on the two respective squares would threaten each other.

Domain Knowledge (1) Study [figure 131](#). The queen in this figure threatens all squares on

the horizontal, the vertical, and the diagonal lines. Conversely, a queen on any square on these lines threatens the queen.

(2) Translate your insights into mathematical conditions that relate the squares' coordinates to each other. For example, all squares on a horizontal have the same y-coordinate. Similarly, all squares on one diagonal have coordinates whose sums are the same. Which diagonal is that? For the other diagonal, the differences between the two coordinates remains the same. Which diagonal does this idea describe?

Hint Once you have figured out the domain knowledge, formulate a test suite that covers horizontals, verticals, and both diagonals. Don't forget to include a pair of arguments for which threatening? must produce `#false`. |

Exercise 482. Design `render-queens`. The function consumes a natural number n , a list l of QPs, and an `Image` i representing a queen. It produces an image of an n by n chess board with images i placed according to l .

You may wish to look for an image for a chess queen on-line or create a simplistic one with the available image functions. |

As for a data representation for *Boards*, we postpone this step until we know how the algorithm implements the process. Doing so in another exercise in data abstraction—see the note at end of preceding chapter. Indeed, a data definition for `Board` isn't even necessary to state the signature for the algorithm proper:

```
; N -> [Maybe [List-of QP]]
; find a solution to the n queens problem

; data example:
(define 4QUEEN-SOLUTION-2
  (list (make-posn 0 2) (make-posn 1 0) (make-posn 2 3) (make-posn 3 1)))

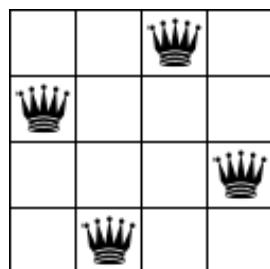
(define (n-queens n)
  #false)
```

The complete puzzle is about finding a placement for n queens on an n by n chess board. So clearly, the algorithm consumes nothing else but a natural number, and it produces a representation for the n queen placements—if a solution exists. The latter can be represented with a list of QPs, which is why we choose

```
; [List-of QP] or #false
```

as the result. Naturally, `#false` represents the failure to find a solution.

The next step is to develop examples and to formulate them as tests. We know that n -queens must fail when given 2 or 3. For 4, there are two solutions with real boards and four identical queens. [Figure 133](#) shows one of them, on the left, and the other one is this:



In terms of data representations, however, there are many different ways to describe these

two images via lists of QPs. [Figure 134](#) sketches some. Can you fill in the rest?

Exercise 483. The tests in [figure 134](#) are awful. No real-world programmer ever spells out all these possible outcomes.

One solution is to use property testing again. Design the `n-queens-solution?` function, which consumes a natural number n and produces a predicate on queen placements that determines whether a given placement is a solution to an n queens puzzle:

- A solution for an n queens puzzle must have length n .
- A [QP](#) on such a list may not threaten any other, distinct [QP](#).

Once you have tested this predicate, use it and `check-satisfied` to formulate the tests for `n-queens`.

An alternative solution is to understand the lists of QPs as sets. If two lists contains the same QPs in different order, they are equivalent as the figure suggests. Hence you could formulate the test for `n-queens` as

```
; [List-of QP] -> Boolean
; is the result equal [as a set] to either of two lists
(define (is-queens-result? x)
  (or (set=? 4QUEEN-SOLUTION-1 x) (set=? 4QUEEN-SOLUTION-2 x)))
```

Design the function `set=?`. It consumes two lists and determines whether they contain the same items—regardless of order. ■

Exercise 484. As bullet 2 above says, you really want to design a function that places n queens on a k by k chessboard in a non-threatening manner:

```
; Board N -> [Maybe [List-of QP]]
; places n queens on board. if possible; otherwise, returns #false

(define (place-queens a-board n)
  #false)
```

[Figure 134](#) already refers to this function in the definition of `n-queens`.

Design the `place-queens` algorithm. Assume you have the following functions to deal with [Boards](#):

```
; N -> Board
; creates the initial n by n board
(define (board0 n)
  ...)

; Board QP -> Board
; places a queen at qp on a-board
(define (add-queen a-board qp)
  a-board)

; Board -> [List-of QP]
; finds spots where it is still safe to place a queen
(define (find-open-spots a-board)
  '())
```

The first function is used in [figure 134](#) to create the initial board representation for `place-`

queens. You will need the other two to describe the generative steps for the algorithm. ▀

You cannot confirm yet that your solution to the preceding exercise works because it relies on an extensive wish list. Technically, it calls for a data representation of [Boards](#) that supports three specific functions. The remaining problem is then to formulate a definition for [Board](#) and to design the functions on the wish list.

Exercise 485. Develop a data definition for [Board](#) and design the three functions specified in [exercise 484](#). Consider the following ideas:

- a [Board](#) collects those positions where a queen can still be placed;
- a [Board](#) contains the list of positions where a queen has been placed;
- a [Board](#) is a grid of n by n squares, each possibly occupied by a queen. **Hint** For this representation, consider using a structure to represent a square with one field for the x index, another one for y , and a third one saying whether the square is threatened.

Use one of the above ideas to solve this exercise.

Challenge Use all three ideas to come up with three different data representations of [Board](#). Abstract your solution to [exercise 484](#) and confirm that it works with any of your data representations of [Board](#). ▀

34 Summary

This fifth part of the book introduces the idea of *eureka!* into program design. Unlike the structural design of the first four parts, *eureka!* design starts from an idea of how the program should solve a problem or process data that represents a problem. Designing here means coming up with a clever way to call a recursive function on a new kind of problem that is like the given one but simpler.

Keep in mind that while we have dubbed it **generative recursion**, most computer scientists refer to these functions as **algorithms**.

Once you have completed this part of the book, you understand the following about the design of generative recursion:

1. The standard outline of the design recipe remains valid.
2. The major change concerns the coding step. It introduces four new questions on going from the completely generic template for generative recursion to a complete function. With two of these questions, you work out the “trivial” parts of the solution process; and with the other two you work out the generative solution step.
3. The minor change is about the termination behavior of generative recursive functions. Unlike structurally designed functions, algorithms may not terminate for some inputs. This problem might be due to inherent limitations in the idea or the translation of the idea into code. Regardless, the future reader of your program deserves a warning about potentially “bad” inputs.

You will encounter some simple or well-known algorithms in your real-world programming tasks. And you will be expected to cope. For truly clever algorithms, software companies employ highly paid computer scientists, domain experts, and mathematicians to work out the conceptual details before they ask programmers to turn the concepts into programs. You must also be prepared for this kind of task, and the best preparation is practice.

v.6.3.0.2

Intermezzo: The Cost of Computation

What do you know about program `f` once the following tests succeed:

```
(check-expect (f 0) 0)
(check-expect (f 1) 1)
(check-expect (f 2) 8)
```

If this question showed up on a standard test, you might guess at this definition:

```
(define (f x) (expt x 3))
```

But nothing speaks against the following:

```
(define (f x) (if (= x 2) 8 (* x x)))
```

Tests tell you only that a program works as expected on a small number of inputs.

You may also wish to re-read [Local Function Definitions](#) and the discussion of integrity checks in [Project: Database](#).

In the same spirit, timing the evaluation of a program application for specific inputs tells you how long it takes to compute the answers for those inputs—and nothing else. You may have two programs—`prog-linear` and `prog-square`—that compute the same answers when given the same inputs and you may find that for all chosen inputs, `prog-linear` always computes the answer faster than `prog-square`. [Making Choices](#) presents just such a pair of programs: `gcd`, a structurally-recursive program, and `gcd-generative`, an equivalent but generative-recursive program. The timing comparison suggests that the latter is much faster than the former.

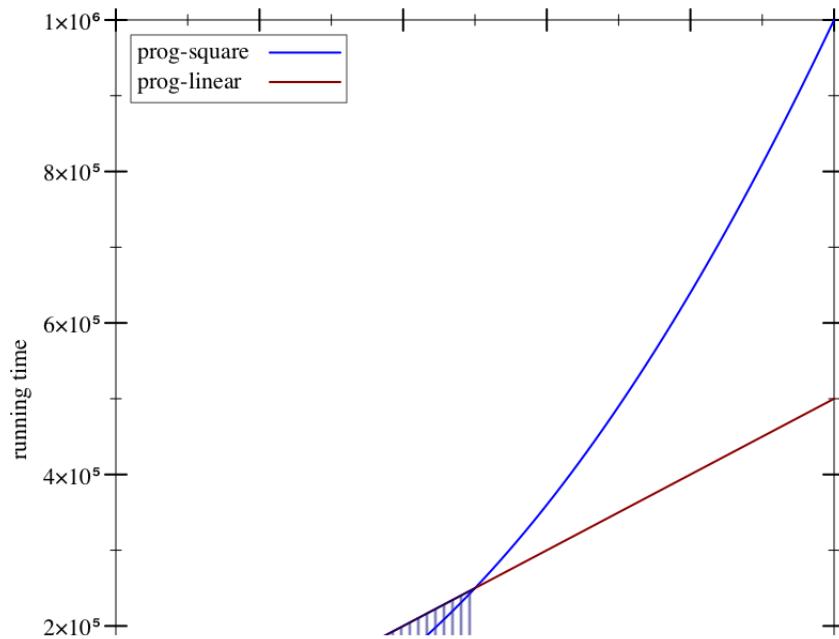


Figure 135: A comparison of two running time expressions

How confident are you that you wish to use `prog-linear` instead of `prog-square`?

Consider the graph in [figure 135](#). In this graph, the x-axis records the size of the input—say the length of a list—and the y-axis records the time it takes to compute the answer for an input of a specific size. Assume that the straight line represents the running time of `prog-linear` and the curved graph represents `prog-square`. In the shaded region, `prog-linear` takes less time than `prog-square`, but at the edge of this region, the two graphs cross, and to its right, the performance of `prog-square` is better than that of `prog-linear`. If, for whatever reasons, you had evaluated the performance of `prog-linear` and `prog-square` only for input sizes in the blue region and if your clients were to run your program mostly on inputs that fall in the non-shaded region, you would be delivering the wrong program.

This intermezzo introduces the basic idea of *algorithmic analysis*, which allows programmers to make general statements about a program’s performance. Any serious programmer must be thoroughly familiar with this notion. It is the basis for analyzing performance attributes of programs, and it is a generally useful concept for describing the growth of functions in many other disciplines. This intermezzo provides a first glimpse at the idea; to understand the details and how to use it properly, you will need to study a text on algorithm analysis.

Concrete Time, Abstract Time

[Making Choices](#) compares the running time of `gcd` and `gcd-generative`. In addition, it argues that the latter is better because it always uses fewer recursive steps than the former to compute an answer. We use this idea as the starting point to analyze the performance of `how-many`, a simple program from [Designing with Self-Referential Data Definitions](#):



```
(define (how-many a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (how-many (rest a-list)) 1)]))
```

Suppose we want to know how long it takes to compute the length of some unknown, non-empty list. Using the rules of computation from [Intermezzo: BSL](#), we can look at this computation as a series of algebraic manipulations:

```
(how-many some-non-empty-list)
==
(cond
  [(empty? some-non-empty-list) 0]
  [else (+ (how-many (rest some-non-empty-list)) 1)])
==
(cond
  [#false 0]
  [else (+ (how-many (rest some-non-empty-list)) 1)])
==
(cond
  [else (+ (how-many (rest some-non-empty-list)) 1)])
==
(+ (how-many (rest some-non-empty-list)) 1)
```

The first step is to replace `a-list` in the definition of `how-many` with the actual argument, `some-non-empty-list`, which yields the first `cond` expression. Next we must evaluate

```
(empty? some-non-empty-list)
```

By assumption the result is `#false`. The question is how long it takes to determine this result. While we don't know the precise amount of time, it is safe to say that checking on the constructor of a list takes a small and fixed amount of time. Indeed, this assumption also holds for the next step, when `cond` checks what the value of the first condition is. Since it is `#false`, the first `cond` line is dropped. Checking whether a `cond` line starts with `else` is equally fast, which means we are left with

```
(+ (how-many (rest some-non-empty-list)) 1)
```

Finally we may safely assume that `rest` extracts the remainder of the list in a fixed amount of time, but otherwise it looks like we are stuck. To compute how long `how-many` takes to determine the length of some list, we need to know how long `how-many` takes to count the number of items in the rest of that list.

Alternatively, if we assume that predicates and selectors take some fixed amount of time, the time it takes `how-many` to determine the length of a list depends on the number of recursive steps it takes. Somewhat more precisely, evaluating `(how-many some-list)` takes roughly n times some fixed amount times where n is the length of the list or, equivalently, the number of times the program recurs.

Generalizing from this example suggests that the running time depends on the size of the input and that the number of recursive steps is a good estimate for the length of an evaluation sequence. For this reason, computer scientists discuss the *abstract running time* of a program as a relationship between the size of the input and the number of recursive steps in an

Abstract because the measure ignores the details of how much time primitive steps take.

evaluation. In our first example, the size of the input is the number of items on the list. Thus, a list of one item requires one recursive step, a list of two needs two steps, and for a list of n items, it's n steps.

Computer scientists use the phrase a program f takes “on the order of n steps” to formulate a claim about abstract running time of f . To use the phrase correctly, it must come with an explanation of n , for example, “it counts the number of items on the given list” or “it is the number of digits in the given number.” Without such an explanation, the original phrase is actually meaningless.

Not all programs have the kind of simple abstract running time as how-many. Take a look at the first recursive program in this book:

```
(define (contains-flatt? a-list-of-names)
  (cond
    [(empty? a-list-of-names) #false]
    [(cons? a-list-of-names)
     (or (string=? (first a-list-of-names) 'flatt)
         (contains-flatt? (rest a-list-of-names)))])))
```

For a list that starts with '`flatt`

```
(contains-flatt? (list 'flatt 'robot 'ball 'game-boy 'pokemon))
```

the program requires no recursive steps. In contrast, if '`flatt` occurs at the end of the list,

```
(contains-flatt? (list 'robot 'ball 'game-boy 'pokemon 'flatt))
```

the evaluation needs as many recursive steps as there are items in the list.

This second analysis brings us to the second important idea of program analysis, namely, the kind of analysis that is performed:

- A *best-case analysis* focuses on the class of inputs for which the program can easily find the answer. In our running example, a list that starts with '`flatt`' is the best kind of input.
- In turn, a *worst-case analysis* determines how badly a program performs for those inputs that stress it most. The `contains-flatt?` function exhibits its worst performance when '`flatt`' is at the end of the input list.
- Finally, a *average analysis* starts from the idea that programmers cannot assume that inputs are always of the best possible shape and that they must hope that the inputs are not of the worst possible shape. In many cases, they must estimate the **average** time a program takes. For example, `contains-flatt?` may—on the average—find '`flatt`' somewhere in the middle of the list. Thus, we could say that if the input contains n items, the abstract running time of `contains-flatt?` is $\frac{n}{2}$, that is, it recurs half as often as the number of items on the input.

Computer scientists therefore usually employ the “on the order of” phrase in conjunction with “on the average” or “in the worst case.”

Returning to the idea that `contains-flatt?` uses, on the average, an “order of a $\frac{n}{2}$ steps” brings us to one more characteristic of abstract running time. Because abstract running time ignores the exact time it takes to evaluate primitive computation steps—checking predicates, selecting values, picking `cond` clauses—we can actually ignore the division by 2. Here is why. By assumption, each basic step takes k units of time, meaning

`contains-flatt?` takes time

$$k \cdot \frac{n}{2}.$$

If you had a newer computer, these basic computations may run twice as fast, in which case we would use $k/2$ as the constant for basic work. Let's call this constant c and calculate:

$$k \cdot \frac{n}{2} = \frac{k}{2} \cdot n = c \cdot n,$$

That is, the abstract running time is always n multiplied by a constant, and that's all that matters to say "order of n ."

Now consider our sorting program from figure 56. Here is a hand-evaluation for a small input, listing all recursive steps:

```
(sort (list 3 1 2))
== (insert 3 (sort (list 1 2)))
== (insert 3 (insert 1 (sort (list 2))))
== (insert 3 (insert 1 (insert 2 (sort '()))))
== (insert 3 (insert 1 (insert 2 '())))
== (insert 3 (insert 1 (list 2)))
== (insert 3 (cons 2 (insert 1 '())))
== (insert 3 (list 2 1))
== (insert 3 (list 2 1))
== (list 3 2 1)
```

The evaluation shows how `sort` traverses the given list and how it sets up an application of `insert` for each number in the list. Put differently, `sort` is a two-phase program.

During the first one, the recursive steps for `sort` set up as many applications of `insert` as there are items in the list. During the second phase, each application of `insert` traverses a sorted list.

Inserting an item is similar to finding an item, so it is not surprising that `insert` and `contains-flatt?` are alike. More specifically, the applications of `insert` to a list of l items triggers between 0 and n recursive steps. On the average, we assume it requires $l/2$, which—with our new terminology—means that, on the average, `insert` takes "on the order of l steps" where l is the length of the given list.

The question is how these lists are to which `insert` adds numbers. Generalizing from the above calculation, we can see that the first one is $n - 1$ items long, the second one $n - 2$, and so on, all the way down to the empty list. Hence, we get that `insert` performs

$$\sum_{l=0}^{l=n-1} \frac{1}{2} \cdot l = \frac{1}{2} \cdot \sum_{l=0}^{n-1} l = \frac{1}{2} \cdot \frac{(n-1) \cdot n}{2} = \frac{1}{4} \cdot (n-1) \cdot n = \frac{1}{4} \cdot (n^2 - n)$$

meaning

$$\frac{1}{4} \cdot n^2 - \frac{1}{4}n$$

represents the best "guess" at the average number of insertion steps. In this last term, n^2 is the dominant factor and so we say that a sorting process takes "on the order of n^2 steps." Exercise 488 ask you to argue why it is correct to simplify this claim in this way.

See exercise 488 why this is the case.

We can also proceed with less formalism and rigor. Because `sort` uses `insert` once per item on the list, we get an "order of n " `insert` steps where n is the size of the list. Since `insert` needs $n/2$ steps, we now see that a sorting process needs $n \cdot n/2$ steps or "on the

order of n^2 .

Totaling it all up, we get that `sort` takes on the “order of n steps” plus n^2 recursive steps in `insert` for a list of n items, which yields

$$n^2 + n$$

steps. See again [exercise 488](#) for details. **Note** This analysis assumes that comparing two items on the list takes a fixed amount of time.

Our final example is the `inf` program from [Local Function Definitions](#):

```
(define (inf l)
  (cond
    [(empty? (rest l)) (first l)]
    [else (if (< (first l) (inf (rest l)))
               (first l)
               (inf (rest l)))]))
```

Let’s start with a small input: `(list 3 2 1 0)`. We know that the result is `0`. Here is the first important step of a hand-evaluation:

```
(inf (list 3 2 1 0))
==
(if (< 3 (inf (list 2 1 0)))
  3
  (inf (list 2 1 0)))
```

From here, we must evaluate the first recursive call. Because the result is `0` and the condition is thus `#false`, we must evaluate the recursion in the else-branch as well.

Once we do so, we see how it triggers two evaluations of `(inf (list 1 0))`:

```
(inf (list 2 1 0))
==
(if (< 2 (inf (list 1 0)))
  2
  (inf (list 1 0)))
```

At this point we can generalize the pattern and summarize it in a table:

original expression	requires two evaluations of
<code>(inf (list 3 2 1 0))</code>	<code>(inf (list 2 1 0))</code>
<code>(inf (list 2 1 0))</code>	<code>(inf (list 1 0))</code>
<code>(inf (list 1 0))</code>	<code>(inf (list 0))</code>

In total, the hand-evaluation requires eight recursive steps for a list of four items. If we added `4` to the front of the list, we would double the number of recursive steps again.

Speaking algebraically, `inf` needs on the order of 2^n recursive steps for a list of n numbers when the last number is the maximum, which is clearly the worst case for `inf`.

Stop! If you paid close attention, you know that the above suggestion is sloppy. The `inf` program really just needs 2^{n-1} recursive steps for a list of n items. What is going on?

Remember that we don’t really measure the exact time when we say “on the order of.” Instead we skip over all built-in predicates, selectors, constructors, arithmetic, and so on and focus on recursive steps only. Now consider this calculation:

$$2^{n-1} = \frac{1}{2} \cdot 2^n.$$

It shows that 2^{n-1} and 2^n differ by a small factor: 2, meaning “on the order of 2^{n-1} steps” describes `inf` in a world where all basic operations provided by *SL run at half the speed when compared to an `inf` program that runs at “the order of 2^n steps.” In this sense, the two expressions really mean the same thing. The question is what exactly they mean, and that is the subject of the next section.

Exercise 486. While a list sorted in descending order is clearly the worst possible input for `inf`, the analysis of `inf`’s abstract running time explains why the rewrite of `inf` with `local` reduces the running time. For convenience, we replicate this version here:

```
(define (infL l)
  (cond
    [(empty? (rest l)) (first l)]
    [else (local ((define s (infL (rest l))))
                 (if (< (first l) s) (first l) s))]))
```

Hand-evaluate `(infL (list 3 2 1 0))` in a manner similar to our evaluation of `(inf (list 3 2 1 0))`. Then argue that `infL` uses on the “order of n steps” in the best and the worst case. You may now wish to revisit [exercise 264](#), which asks you to explore a similar problem. ■

Exercise 487. A number tree is either a number or a pair of number trees. Design `sum-tree`, which determines the sum of the numbers in a tree. What is its abstract running time? What is an acceptable measure of the size of such a tree? What is the worst possible shape of the tree? What’s the best possible shape? ■

The Definition of “on the Order of”

The preceding section alluded to all the key ingredients of the phrase “on the order of.” Now it is time to introduce a rigorous description of the phrase. Let’s start with the two ideas that the preceding section develops:

1. The abstract measurement of performance is a relationship between two quantities: the size of the input and the number of recursive steps needed to determine the answer. The relationship is actually a mathematical function that maps one natural number—the size of the input—to another—the time needed.
2. Hence, a general statement about the performance of a program is a statement about a function, and a comparison of the performance of two programs calls for the comparison of two such functions.

And how do you decide whether one such function is “better” than another?

Exercise 248 tackles the different question of whether we can formulate a program that decides whether two other programs are equal. In this intermezzo, we are not interested in writing a program; we are using plain mathematical arguments.

Let’s return to the imagined programs `prog-linear` and `prog-square` from the introduction. They compute the same results but their performance differs. The `prog-square` program requires “on the order of n steps” while `prog-linear` uses “on the order of n^2 steps.” Mathematically speaking, the performance function for `prog-linear` is

$$L(n) = c_L \cdot n$$

and prog-square's associated performance function is

$$S(n) = c_S \cdot n^2$$

In these definitions, c_L is the cost for each recursive step in prog-square and c_S is the cost per step in prog-linear.

Say we figure out that $c_L = 1000$ and $c_S = 1$. Then we can tabulate these abstract running times to make the comparison concrete:

n	10	100	1000	2000
prog-square	100	10000	1000000	4000000
prog-linear	10000	100000	1000000	2000000

Like the graphs in [figure 135](#), the table at first seems to say that prog-square is better than prog-linear, because for inputs of the same size n , prog-square's result is smaller than prog-linear's. But look at the last column in the table. Once the inputs is sufficiently large, prog-square's advantage decreases until it disappears at an input size of 1000. **Thereafter** prog-square is **always** slower than prog-linear.

This last insight is the key to the precise definition of the phrase “order of.” If a function f on the natural numbers produces larger numbers than some function g **for all** natural numbers, then f is clearly larger than g . But what if this comparison fails for just a few inputs, say for 1000 or 1000000, and holds for all others? In that case, we would still like to say f is better than g . And this brings us to the following definition.

Definition Given a function g on the natural numbers, $O(g)$ (pronounced: “big-O of g ”) is a class of functions on natural numbers. A function f is a member of $O(g)$ if **there exist** numbers c and *bigEnough* such that

$$\text{for all } n \geq \text{bigEnough} \text{ it is true that } f(n) \leq c \cdot g(n).$$

Terminology If $f \in O(g)$, we say f is no worse than g .

Naturally, we would love to illustrate this definition with the example of prog-linear and prog-square from above. Recall the performance functions for prog-linear and prog-square, with the constants plugged in:

$$S(n) = 1 \cdot n^2$$

and

$$L(n) = 1000 \cdot n.$$

The key is to find the magic numbers c and *bigEnough* such that $L \in O(S)$, which would validate that prog-square's performance is no worse than prog-linear's. For now, we just tell you what these numbers are:

$$\text{bigEnough} = 1000, c = 1.$$

Using these numbers, we need to show that

$$L(n) \leq 1 \cdot S(n)$$

for every single n larger than 1000. Here is how this kind of argument is settled:

Pick some specific n_0 that satisfies the condition:

$$1000 \leq n_0.$$

We use the symbolic name n_0 so that we don't make any specific assumptions about it. Now recall from algebra that you can multiply both sides of the inequality with the same positive factor, and the inequality still holds. We use n_0 :

$$1000 \cdot n_0 \leq n_0 \cdot n_0.$$

At this point, it is time to observe that the left side of the inequality is just $H(n_0)$ and the right side is $G(n_0)$:

$$L(n_0) \leq S(n_0).$$

Since n_0 is a generic number of the right kind, we have shown exactly what we wanted to show.

Usually you find *bigEnough* and c working your way backwards through such an argument. While this kind of mathematical reasoning is fascinating, we leave it to a course on algorithms.

The definition of O also explains with mathematical rigor why we don't have to pay attention to specific constants in our comparisons of abstract running times. Say we can make each basic step of `prog-linear` go twice as fast so that we have:

$$S(n) = \frac{1}{2} \cdot n^2$$

and

$$L(n) = 1000 \cdot n.$$

The above argument goes through by doubling *bigEnough* to 2000.

Stop! Work through the argument with this suggestion.

Finally, most people use O together with a shorthand for stating functions. Thus they say how-many's running time is $O(n)$ —because they tend to think of n as an abbreviation of the (mathematical) function $id(n) = n$. Similarly, this use yields the claim that `sort`'s worst-case running time is $O(n^2)$ and `inc`'s is $O(2^n)$ —again because n^2 is short-hand for the function $sqr(n) = n^2$ and 2^n is short for $expt(n) = 2^n$.

Stop! What does it mean to say that a function's performance is $O(1)$?

Exercise 488. In the first subsection, we stated that the function $f(n) = n^2 + n$ belongs to the class $O(n^2)$. Determine the pair of numbers c and *bigEnough* that verify this claim. ▀

Exercise 489. Consider the functions $f(n) = 2^n$ and $g(n) = 1000 \cdot n$. Show that g belongs to $O(f)$, which means that f is abstractly speaking more (or at least equally) expensive than g . If the input size is guaranteed to be between 3 and 12, which function is better? ▀

Exercise 490. Compare and $g(n) = n^2$. Does f belong to $O(g)$ or g to $O(f)$? ▀

Why do Programs use Predicates and Selectors?

The notion of “on the order of” explains why the design recipes produce both well-organized and “performant” programs. We illustrate this insight with a single example, the design of a program that searches for a number in a list of numbers. Here are the signature, the purpose statement, and examples formulated as tests:

| ; Number [List-of Number] -> Boolean

```
; is x in l

(check-expect (search 0 '(3 2 1 0)) #true)
(check-expect (search 4 '(3 2 1 0)) #false)
```

Here are two definitions that live up to these expectations:

```
(define (searchL x l)
  (cond
    [(empty? l) #false]
    [else
      (or (= (first l) x)
          (searchL x (rest l))))])

(define (searchS x l)
  (cond
    [= (length l) 0] #false]
    [else
      (or (= (first l) x)
          (searchS x (rest l))))]))
```

The design of the program on the left follows the design recipe. In particular, the development of the template calls for the use of structural predicates per clause in the data definition. Following this advice yields a conditional program whose first `cond` line deals with empty lists and the second one with all others. The question in the first `cond` line uses `empty?` and the second one uses `cons?` of `else`.

The design of the program on the right fails to live up to the structural design recipe. It instead takes inspiration from the idea that lists are containers that have a size. Hence, a program can check this size for `0`, which is equivalent to checking for emptiness.

Technically `searchS` uses generative recursion.

Although this idea is functionally correct, it makes the assumption that the cost of `*SL`-provided operations is a fixed constant. If, however, `length` is more like `how-many`, `searchS` is going to be much slower than `searchL`. Using our new terminology, `searchL` is using $O(n)$ recursive steps while `searchS` needs $O(n^2)$ steps for a list of n items. In short, using arbitrary `*SL` operations to formulate conditions may shift performance from one class of functions to one that is much worse.

Let's wrap up this intermezzo with an experiment that determines whether `length` is a constant-time program or whether it consumes time proportionally to the length of the given list. The easiest way is to define a program that creates a long list and determines how much time each version of the search program takes:

```
; N -> [List Boolean Boolean]
; how long do searchS and searchL take
; to look for n in (list 0 ... (- n 1))
(define (timing n)
  (local ((define long-list (build-list n (lambda (x) x))))
    (list
      (time (searchS n long-list))
      (time (searchL n long-list)))))
```

Now run this program on, say, `10000` and `20000`. If `length` is like `empty?`, the times for the second run will be roughly twice of the first one; otherwise, the time for `searchS` will increase dramatically.

Stop! Conduct the experiment.

See [Data Representations with Accumulators](#) for how other languages track the size of a container.

Assuming you have completed the experiment, you now know that `length` takes time proportionally to the size of the given list and the “S” in `searchS` stands for “squared” because its running time is $O(n^2)$. But don’t jump to the conclusion that this kind of reasoning holds for every programming language you will encounter. Many deal with containers differently than *SL. Understanding how they do this, calls for the introduction of one more design concept—accumulators, which is what the final part of this book is about.

Exercise 491. Reconnect with the material in [Generalizing Functions](#). Explain why `render-poly` uses

```
| (empty? (rest (rest (rest p)))))
```

as its first condition? What would be an alternative? Similarly explain why `connect-docs` uses

```
| (empty? (rest p))
```

```
|
```

v.6.3.0.2

VI Accumulators

When you ask ISL+ to apply some function f to an argument a , you usually get some value v . If you evaluate $(f\ a)$ again, you get v again. As a matter of fact, you get v no matter how often you request the evaluation of $(f\ a)$. Whether the function is applied for the first time or the hundredth time, whether the application is located in DrRacket's interactions area or inside the function itself, doesn't matter. The function works according to its purpose statement, and that's all you need to know.

The function application may also loop forever or signal an error, but we ignore these possibilities. We also ignore random, which is the only exception to this rule.

This principle of context-independence plays a critical role in the design of recursive functions. When it comes to design, you are free to assume that the function computes what the purpose statement promises—even if the function isn't defined yet. In particular, you are free to use the results of recursive calls to create the code of some function, usually in one of its cond clauses. The template and coding steps of the design recipes for both structurally and generative recursive functions rely on this idea.

Although context-independence facilitates the design of functions, it also causes two problems. The general idea is that context-independence induces a loss of knowledge during a recursive evaluation; a function does not “know” whether it is called on a complete list or on a piece of that list. For structurally recursive programs this loss of knowledge means that they may have to traverse data more than once, inducing a grave performance cost. For functions that employ generative recursion, the loss means that the function may not be able to compute the result at all; instead the function loops forever for certain inputs. The preceding part illustrates this second problem with a graph traversal function that cannot find a path between two nodes for a circular graph.

This part introduces a variant of the design recipes to address this “loss of context” problem. Since we wish to retain the principle that $(f\ a)$ returns the same result no matter how often it is evaluated, our only solution is to add **an argument that represents the context** of the function call. We call this additional argument an *accumulator*. During the traversal of data, the recursive calls continue to receive new regular arguments while accumulators change in relation to the other arguments and the context of the call.

Designing functions with accumulators correctly is clearly more complex than any of the design approaches from the preceding chapters. The key is to understand the relationship between the proper arguments and the accumulators. The following chapters explain how to design functions with accumulators and how they work.

36 The Loss of Knowledge

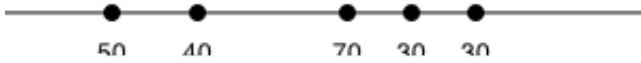
Both functions designed according to structural recipes and the generative one suffer from the loss of knowledge, though in different ways. This chapter explains with two examples—one from each category—how the lack of contextual knowledge affects the performance of functions. While the first section is about structural recursion, the second one addresses concerns in the generative realm.

36.1 A Problem with Structural Processing

Let's start with a seemingly straightforward example:

Sample Problem: You are working for a geometer team that will measure the length of roads segments. The team asked you to design a program that translates these relative distances between a series of road points into absolute distances for some starting point.

For example, we might be given a line such as this:



Each number specifies the distance between two dots. What we need is the following picture, where each dot is annotated with the distance to the left-most end:



Designing a program that performs this calculation is at this point an exercise in structural function design. Figure 136 contains the complete program. When the given list is not '`()`', the natural recursion computes the absolute distance of the remainder of the dots to the first item on `(rest l)`. Because the first item is not the actual origin and has a distance of `(first l)` to the origin, we must add `(first l)` to each number on the result of the natural recursion. This second step—adding a number to each item on a list of numbers—requires an auxiliary function.

```

; [List-of Number] -> [List-of Number]
; convert a list of relative distances to a list of absolute distances
; the first item on the list represents the distance to the origin

(check-expect (relative->absolute '(50 40 70 30 30))
              '(50 90 160 190 220))

(define (relative->absolute l)
  (cond
    [(empty? l) '()]
    [else (local ((define rest-of-l (relative->absolute (rest l)))
                  (define adjusted (add-to-each (first l) rest-of-l)))
              (cons (first l) adjusted))]))

; Number [List-of Number] -> [List-of Number]
; add n to each number on l

(check-expect (cons 50 (add-to-each 50 '(40 110 140 170)))
              '(50 90 160 190 220))

(define (add-to-each n l)
  (cond
    [(empty? l) '()]
    [else (cons (+ (first l) n) (add-to-each n (rest l))))]))
```

Figure 136: Converting relative distances to absolute distances

While designing the program is relatively straightforward, using it on larger and larger lists reveals a problem. Consider the evaluation of the following expression:

```
(relative->absolute (build-list size add1))
```

As we increase `size`, the time needed grows even faster:

The times will differ from computer to computer and year to year. These measurements were conducted in 2014 on a MacMini running OS X 10.8.5; the previous measurement took place in 1998, and the times were 100x larger.

size	1000	2000	3000	4000	5000	6000	7000
time	25	109	234	429	689	978	1365

Instead of doubling as we go from 1000 to 2000 items, the time quadruples. This is also the approximate relationship for going from 2000 to 4000, and so on. Using the terminology of [Intermezzo: The Cost of Computation](#), we say that the function's performance is $O(n^2)$ where n is the length of the given list.

Exercise 492. Reformulate `add-to-each` using `map` and `lambda`. ■

Exercise 493. Determine the abstract running time of `relative->absolute`.

Hint Evaluate the expression

```
(relative->absolute (build-list size add1))
```

by hand. Start by replacing `size` with 1, 2, and 3. How many natural recursions of `relative->absolute` and `add-to-each` are required each time? ■

Considering the simplicity of the problem, the amount of “work” that the program performs is surprising. If we were to convert the same list by hand, we would tally up the total distance and just add it to the relative distances as we take another step along the line. Why can't a program use this idea?

Let's attempt to design a second version of the function that is closer to our manual method. The new function is still a list-processing function, so we start from the appropriate template:

```
(define (relative->absolute/a l)
  (cond
    [(empty? l) ...]
    [else (... (first l) ... (relative->absolute/a (rest l)) ...)]))
```

Now imagine an “evaluation” of `(relative->absolute/a (list 3 2 7))`:

```
==  

(cons ... 3 ...  

  (relative->absolute/a (list 2 7)))  

==  

(cons ... 3 ...  

  (cons ... 2 ...  

    (relative->absolute/a (list 7))))
```

```

==  

(cons ... 3 ...  

  (cons ... 2 ...  

    (cons ... 7 ...  

      (relative->absolute/a '())))
)

```

The first item of the result list should obviously be 3, and it is easy to construct this list. But, the second one should be (+ 3 2), yet the second instance of relative->absolute/a has no way of “knowing” that the first item of the **original** list is 3. The “knowledge” is lost.

Put differently, the problem is that recursive functions are independent of their context. A function processes L in (cons N L) in the same manner as in (cons K L). Indeed, it would also process L in that manner if it were given L by itself.

To make up for the loss of “knowledge,” we equip the function with an additional parameter: accu-dist. The new parameter represents the accumulated distance, which is the tally that we keep when we convert a list of relative distances to a list of absolute distances. Its initial value must be 0. As the function processes the numbers on the list, it must add them to the tally.

Here is the revised definition:

```

(define (relative->absolute/a l accu-dist)
  (cond
    [(empty? l) '()]
    [else (local ((define tally (+ (first l) accu-dist)))
                 (cons tally (relative->absolute/a (rest l) tally)))])))

```

The recursive application consumes the rest of the list and the new absolute distance of the current point to the origin. Although this means that two arguments are changing simultaneously, the change in the second one strictly depends on the first argument. The function is still a plain list-processing procedure.

Evaluating our running example again, shows how much the use of an accumulator simplifies the conversion process:

```

(relative->absolute/a (list 3 2 7))
== (relative->absolute/a (list 3 2 7) 0)
== (cons 3 (relative->absolute/a (list 2 7) 3))
== (cons 3 (cons 5 (relative->absolute/a (list 7) 5)))
== (cons 3 (cons 5 (cons 12 (relative->absolute/a '() 12))))
== (cons 3 (cons 5 (cons 12 '())))

```

Each item in the list is processed once. When relative->absolute/a reaches the end of the argument list, the result is completely determined and no further work is needed. In general, the function performs on the order of N natural recursion steps for a list with N items.

One minor problem with the new definition is that unlike relative->absolute, the new function consumes two arguments not just one. Worse, someone might accidentally misuse relative->absolute/a by applying it to a list of numbers and a number that isn’t 0. We can solve both problems with a function definition that uses a local definition to encapsulate relative->absolute/a; figure 137 shows the result. Now, relative->absolute and relative->absolute.v2 are indistinguishable with respect to the input-

output relationship.

```
; [List-of Number] -> [List-of Number]
; convert a list of relative distances to a list of absolute distances
; the first item on the list represents the distance to the origin

(check-expect (relative->absolute.v2 '(50 40 70 30 30))
              '(50 90 160 190 220))

(define (relative->absolute.v2 l0)
  (local (; [List-of Number] Number -> [List-of Number]
          (define (relative->absolute/a l accu-dist)
            (cond
              [(empty? l) '()]
              [else
                (local ((define accu (+ (first l) accu-dist)))
                  (cons accu (relative->absolute/a (rest l) accu))))])
            (relative->absolute/a l0 0))))
```

Figure 137: Converting relative distances with an accumulator

Now let's look at how this version of the program performs. To this end, we evaluate

```
(relative->absolute.v2 (build-list size add1))
```

and tabulate the results for several values of `size`:

size	1000	2000	3000	4000	5000	6000	7000
time	0	0	0	0	0	1	1

Amazingly `relative->absolute.v2` never takes more than one second to process such lists, even for a list of 7000 numbers. Comparing this performance to the one of `relative->absolute`, you may think that accumulators are a miracle cure for all slow-running programs. Unfortunately, this isn't the case, but when a structurally recursive function has to re-process the result of the natural recursion you should definitely consider the use of accumulators. In this particular case, the performance improved from $O(n^2)$ to $O(n)$ —with an additional large reduction in the constant.

Exercise 494. With a bit of design and a bit of tinkering a friend of yours came up with the following solution for the sample problem:

Messrs. Adrian German and Mardin Yadegar suggested this exercise.

```
; [List-of Number] -> [List-of Number]
; convert a list of relative distances to a list of absolute distances
; the first item on the list represents the distance to the origin

(check-expect (relative->absolute '(50 40 70 30 30))
              '(50 90 160 190 220))

(define (relative->absolute l)
  (reverse
    (foldr (lambda (f l) (cons (+ f (first l)) l))
           (list (first l))
           (reverse (rest l)))))
```

This simple solution merely uses well-known ISL+ functions: `reverse` and `foldr`. Using `lambda`, as you know, is just a convenience. You may also recall from [Abstraction](#) that `foldr` is designable with the design recipes presented in the first two parts of the book.

Does your friend's solution mean there is no need for our complicated design in this motivational section? For a solution, see [Recognizing the Need for an Accumulator](#), but do reflect on the question first and better still try to design `reverse` on your own. ■

36.2 A Problem with Generative Recursion

Let us revisit the problem of “traveling” along a path in a graph:

Sample Problem: Design an algorithm that checks whether two nodes are connected in a simple graph. In a *simple graph*, each node has exactly one, one-directional connection to another node, possibly itself.

[Algorithms that Backtrack](#) covered the variant where the algorithm has to discover the path. This sample problem is simpler than that, because this section focuses on the design of an accumulator version of the algorithm.

Consider the sample graph in [figure 138](#). There are six nodes: *A* through *F*, and six connections. To get from *A* to *E*, you must go through *B* and *C*. It is impossible, though, to reach *F* from *A* or from any other node besides *F* itself.

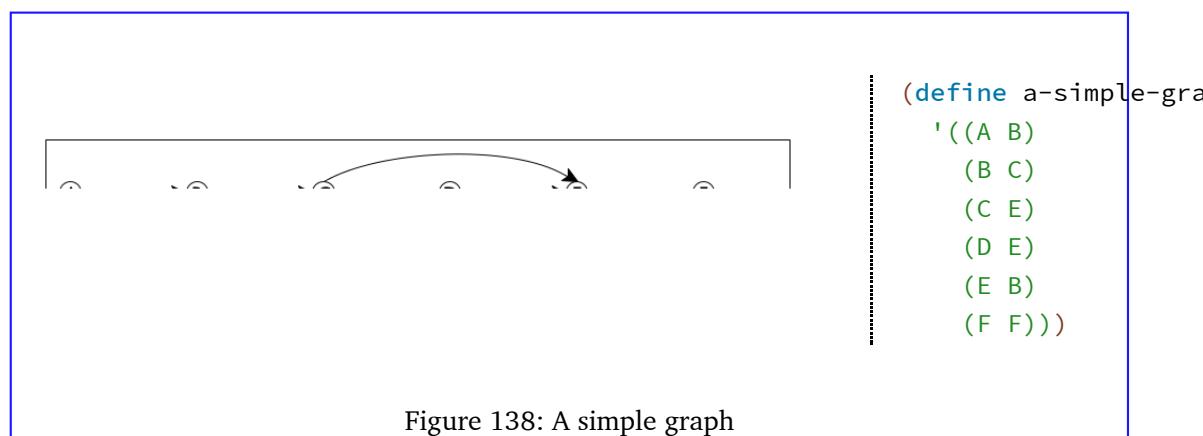


Figure 138: A simple graph

The right part of [figure 138](#) shows how to represent this graph with nested lists. Each node is represented by a list of two symbols. The first symbol is the label of the node; the second one is the single node that is reachable from the first one. Here are the relevant data definitions:

```
; A SimpleGraph is a [List-of Connection]
; A Connection is (list Node Node)
; A Node is a Symbol
```

They are straightforward translations of our informal descriptions.

We already know that the problem calls for generative recursion, and it is easy to create the header material:

```
; Node Node SimpleGraph -> Boolean
; is there a path from origination to destination in sg

(check-expect (path-exists? 'A 'E a-simple-graph) #true)
(check-expect (path-exists? 'A 'F a-simple-graph) #false)
```

```
(define (path-exists? origination destination sg)
  #false)
```

What we need are answers to the four basic questions of the recipe for generative recursion:

- The problem is trivial if the nodes `origination` and `destination` are the same.
- The trivial solution is `#true`.
- If `origination` is not the same as `destination`, there is only one thing we can do: step to the immediate neighbor and search for `destination` from there.
- There is no need to do anything if we find the solution to the new problem. If `origination`'s neighbor is connected to `destination`, then so is `origination`. Otherwise there is no connection.

From here we just need to express these answers in ISL+ to obtain a full-fledged program.

```
; Node Node SimpleGraph -> Boolean
; is there a path from origination to destination in sg

(check-expect (path-exists? 'A 'E a-simple-graph) #true)
(check-expect (path-exists? 'A 'F a-simple-graph) #false)

(define (path-exists? origination destination sg)
  (cond
    [(symbol=? origination destination) #t]
    [else (path-exists? (neighbor origination sg) destination sg)]))

; Node SimpleGraph -> Node
; determine the node that is connected to a-node in sg

(check-expect (neighbor 'A a-simple-graph) 'B)
(check-error (neighbor 'G a-simple-graph) "neighbor: not a node")

(define (neighbor a-node sg)
  (cond
    [(empty? sg) (error "neighbor: not a node")]
    [else (if (symbol=? (first (first sg)) a-node)
              (second (first sg))
              (neighbor a-node (rest sg))))]))
```

Figure 139: Finding a path in a simple graph

Figure 139 contains the complete program, including the function for looking up the neighbor of a node in a simple graph—a straightforward exercise in structural recursion—and test cases for both. Don't run the program, however. If you do, be ready with your mouse to stop the run-away program. Indeed, even a casual look at the function suggests that we have a problem. Although the function is supposed to produce `#false` if there is no path from `origination` to `destination`, the program doesn't contain `#false` anywhere. Conversely, we need to ask what the function actually does when there is no path between two nodes.

Take another look at figure 138. In this simple graph there is no path from `C` to `D`. The connection that leaves `C` passes right by `D` and instead goes to `E`. So let's look at a hand

evaluation:

```
(path-exists? 'C 'D '((A B) (B C) (C E) (D E) (E B) (F F)))
== (path-exists? 'E 'D '((A B) (B C) (C E) (D E) (E B) (F F)))
== (path-exists? 'B 'D '((A B) (B C) (C E) (D E) (E B) (F F)))
== (path-exists? 'C 'D '((A B) (B C) (C E) (D E) (E B) (F F)))
```

It confirms that as the function recurs, it calls itself with the exact same arguments again and again. In other words, the evaluation never stops.

Our problem with `path-exists?` is again a loss of “knowledge,” similar to that of `relative->absolute` in the preceding section. Like `relative->absolute`, the design of `path-exists?` uses a recipe and assumes context-independence for recursive calls. In the case of `path-exists?` this means, in particular, that the function doesn’t “know” whether a previous application in the current chain of recursions received the exact same arguments.

The solution to this design problem follows the pattern of the preceding section. We add a parameter, which we call `seen` and which represents the accumulated list of origination nodes that the function has encountered, starting with the original application. Its initial value must be `'()`. As the function checks on a specific `origination` and moves to its neighbors, `origination` is added to `seen`.

Here is a first revision of `path-exists?`, dubbed `path-exists?/a`:

```
; Node Node SimpleGraph [List-of Node] -> Boolean
; is there a path from origination to destination in sg
; assume the nodes in seen are known not to solve the problem
(define (path-exists?/a origination destination sg seen)
  (cond
    [(symbol=? origination destination) #t]
    [else (path-exists?/a (neighbor origination sg) destination sg
                           (cons origination seen))]))
```

The addition of the new parameter alone does not solve our problem, but, as the hand-evaluation of

```
(path-exists?/a 'C 'D '((A B) (B C) (C E) (D E) (E B) (F F)) '())
```

shows, provides the foundation for one:

```
== (path-exists?/a 'E 'D '((A B) (B C) (C E) (D E) (E B) (F F)) '(C))
== (path-exists?/a 'B 'D '((A B) (B C) (C E) (D E) (E B) (F F)) '(E C))
== (path-exists?/a 'C 'D '((A B) (B C) (C E) (D E) (E B) (F F)) '(B E C))
```

In contrast to the original function, the revised function no longer calls itself with the exact same arguments. While the three arguments proper are again the same for the third recursive application, the accumulator argument is different from that of the first application. Instead of `'()`, it is now `'(B E C)`. The new value represents the fact that during the search of a path from `'C` to `'D`, the function has inspected `'B`, `'E`, and `'C` as starting points.

All we need to do now, is to make the algorithm exploit the accumulated knowledge. Specifically, the algorithm can determine whether the given `origination` is already an item in `seen`. If so, the problem is also trivially solvable yielding `#false` as the solution. [Figure 140](#) contains the definition of `path-exists.v2?`, which is the revision of `path-exists?`. The definition refers to `member?`, an ISL+ function.

```

; Node Node SimpleGraph -> Boolean
; is there a path from origination to destination in sg

(check-expect (path-exists.v2? 'A 'E a-simple-graph) #true)
(check-expect (path-exists.v2? 'A 'F a-simple-graph) #false)

(define (path-exists.v2? origination destination sg)
  (local (; Node Node SimpleGraph [List-of Node] -> Boolean
          (define (path-exists?/a origination seen)
            (cond
              [(symbol=? origination destination) #t]
              [(member? origination seen) #f]
              [else (path-exists?/a (neighbor origination sg)
                                    (cons origination seen))])))
    (path-exists?/a origination '())))

```

Figure 140: Finding a path in a simple graph with an accumulator

The definition of `path-exists.v2?` also eliminates the two minor problems with the first revision. By localizing the definition of the accumulating function, we can ensure that the first call always uses `'()` as the initial value for `seen`. And, `path-exists.v2?` satisfies the exact same contract and purpose statement as the `path-exists?` function.

Still, there is a significant difference between `path-exists.v2?` and `relative-to-absolute2`. Whereas the latter was equivalent to the original function, `path-exists.v2?` improves on `path-exists?`. While the latter fails to find an answer for some inputs, `path-exists.v2?` finds a solution for any simple graph.

Exercise 495. Modify the definitions of `find-path` and `find-path/list` in figure 128 so that they produce `#false`, even if they encounter the same starting point twice. ■

37 Designing Accumulator-style Functions

The preceding chapter illustrates the need for accumulating extra knowledge with two examples. In one case, accumulation makes it easy to understand the function and yields one that is far faster than the original version. In the other case, accumulation is necessary for the function to work properly. In both cases though, the need for accumulation becomes only apparent once a properly designed function exists.

Generalizing from the preceding chapter suggests that the design of accumulator functions has two major aspects:

1. the recognition that a function benefits from an accumulator;
2. an understanding of what the accumulator represents with respect to the design.

The first two sections address these two questions. Because the second one is a difficult topic, the third section illustrates it with a series of examples that convert regular functions into accumulating ones.

37.1 Recognizing the Need for an Accumulator

Recognizing the need for accumulators is not an easy task. We have seen two reasons, and

they are the most prevalent reasons for adding accumulator parameters. In either case, it is critical that we first built a complete function based on a design recipe. Then we study the function and look for one of the following characteristics:

1. If a structurally recursive function processes the result of its natural recursion with an auxiliary, recursive function, consider the use of an accumulator parameter.

Take a look at the definition of `invert`:

```
; [List-of X] -> [List-of X]
; construct the reverse of alox

(check-expect (invert '(a b c)) '(c b a))

(define (invert alox)
  (cond
    [(empty? alox) '()]
    [else (add-as-last (first alox) (invert (rest alox)))]))

; X [List-of X] -> [List-of X]
; add an-x to the end of alox

(check-expect (add-as-last 'a '(c b)) '(c b a))

(define (add-as-last an-x alox)
  (cond
    [(empty? alox) (list an-x)]
    [else (cons (first alox) (add-as-last an-x (rest alox))))]))
```

The result of the recursive application produces the reverse of the rest of the list. It is processed by `add-as-last`, which adds the first item to the reverse of the rest and thus creates the reverse of the entire list. This second, auxiliary function is also recursive. We have thus identified a potential candidate.

It is now time to study some hand-evaluations, as we did in [A Problem with Structural Processing](#), to see whether an accumulator helps. Consider the following expression and its evaluation:

```
(invert '(a b c))
== (add-as-last 'a (invert '(b c)))
== (add-as-last 'a (add-as-last 'b (invert '(c))))
== (add-as-last 'a (add-as-last 'b (add-as-last 'c (invert '())))))
== (add-as-last 'a (add-as-last 'b (add-as-last 'c '())))
== (add-as-last 'a (add-as-last 'b '(c)))
== (add-as-last 'a '(c b))
== '(c b a)
```

Eventually `invert` reaches the end of the given list—just like `add-as-last`—and if it knew which items to put there, there would be no need for the auxiliary function.

2. If we are dealing with a function based on generative recursion, we are faced with a much more difficult task. Our goal must be to understand whether the algorithm can fail to produce a result for inputs for which we expect a result. If so, adding a parameter that accumulates knowledge may help. Because these situations are complex, we defer the discussion of an example to [More Uses of Accumulation](#).

Exercise 496. Argue that, in the terminology of [Intermezzo: The Cost of Computation](#), `invert` consumes $O(n^2)$ time when the given list consists of n items. ■

Exercise 497. Does the insertion `sort>` function from [Recursive Auxiliary Functions](#) need an accumulator? If so, why? If not, why not? ■

37.2 Adding Accumulators

Once you have decided that an existing function should be equipped with an accumulator, take these two steps:

- Determine the knowledge that the accumulator represents, what kind of data to use, and how the knowledge is acquired as data.

For example, for the conversion of relative distances to absolute distances, it suffices to accumulate the total distance encountered so far. As the function processes the list of relative distances, it adds each new relative distance found to the accumulator's current value. For the routing problem, the accumulator remembers every node encountered. As the path-checking function traverses the graph, it `conses` each new node on to the accumulator.

In general, you want to proceed as follows.

1. Create an accumulator template:

```
; Domain -> Range
(define (function d0)
  (local (; Domain AccumulatorDomain -> Range
         ; accumulator ...
         (define (function/a d a)
           ...))
        (function/a d0 a0)))
```

Sketch a manual evaluation of an application of `function` to understand the nature of the accumulator.

2. Determine the kind of data that the accumulator tracks.

Write down a statement that explains the accumulator as a relationship between the argument `d` of the auxiliary `function/a` and the original argument `d0`.

Note The relationship remains constant—also called **invariant**—over the course of the evaluation. Because of this property, an accumulator statement is also called an accumulator *invariant*.

3. Use the accumulator statement to determine the initial value `a0` for `a`.
4. Also exploit the accumulator statement to determine how to compute the accumulator for the recursive function calls within the definition of `function/a`.
- Exploit the accumulator's knowledge for the design of the auxiliary function.

For a structurally recursive function, the accumulator's value is typically used in the base case, that is, the `cond` clause that does not recur. For functions that use generative recursive functions, the accumulated knowledge might be used in an existing base case, in a new base case, or in the `cond` clauses that deal with generative recursion.

As you can see, the key is the precise description of the role of the accumulator. It is therefore important to practice this skill.

Let's take a look at the `invert` example:

```
; [List-of X] -> [List-of X]
; construct the reverse of alox0

(check-expect (invert.v2 '(a b c)) '(c b a))

(define (invert.v2 alox0)
  (local (; [List-of X] ??? -> [List-of X]
          ; construct the reverse of alox
          ; accumulator ...
          (define (invert/a alox a)
            (cond
              [(empty? alox) ...]
              [else (invert/a (rest alox) ... a ....)])))
  (invert/a alox0 ...)))
```

As illustrated in the preceding section, this template suffices to sketch a manual evaluation of an expression such as

```
(invert '(a b c))
```

Here is the idea:

```
== (invert/a '(a b c) a0)
== (invert/a '(b c) ... 'a ... a0)
== (invert/a '(c) ... 'b ... 'a ... a0)
== (invert/a '() ... 'c ... 'b ... 'a ... a0)
```

This sketch suggests that `invert/a` can keep track of all the items it has seen in a list that tracks the difference between `alox0` and `a` in reverse order. The initial value is clearly `'()`; updating the accumulator inside of `invert/a` with `cons` produces exactly the desired value when `invert/a` reaches `'()`.

Here is a refined template that includes these insights:

```
(define (invert.v2 alox0)
  (local (; [List-of X] [List-of X] -> [List-of X]
          ; construct the reverse of alox
          ; accumulator a is the list of all those items
          ; on alox0 that precede alox in reverse order
          (define (invert/a alox a)
            (cond
              [(empty? alox) a]
              [else (invert/a (rest alox) (cons (first alox) a)))))
  (invert/a alox0 '())))
```

While the body of the `local` definition initializes the accumulator with `'()`, the recursive call uses `cons` to add the current head of `alox` to the accumulator. In the base case, `invert/a` uses the knowledge in the accumulator—the reversed list.

Note how once again `invert.v2` traverses the list just. In contrast, `invert` re-processes every result of its natural recursion with add-as-last. Stop! Measure how much faster `invert.v2` runs than `invert` on the same list.

Terminology Programmers use the phrase *accumulator-style function* when they discuss functions that use an accumulator parameter. Examples of functions in accumulator-style are `relative->absolute/a`, `path-exists?/a`, and `invert/a`.

37.3 Transforming Functions into Accumulator-Style

Articulating the accumulator statement is difficult but without formulating a good invariant, it is impossible to understand an accumulator-style function. Since the goal of a programmer is to make sure that others who follow understand the code easily, practicing this skill is critical. And formulating invariants deserves a lot of practice.

The goal of this section is to study the formulation of accumulator statements with three case studies: a summation function, the factorial function, and a tree-traversal function. Each such case is about the conversion of a structurally recursive function into accumulator style. None actually call for the use of an accumulator parameter. But they are easily understood and, with the elimination of all other distractions, using such examples allows us to focus on the articulation of the accumulator invariant.

For the first example, consider the following definition of the `sum` function:

```
; [List-of Number] -> Number
; compute the sum of the numbers onalon

(check-expect (sum '(10 4 6)) 20)

(define (sum alon)
  (cond
    [(empty? alon) 0]
    [else (+ (first alon) (sum (rest alon))))]))
```

Here is the first step toward an accumulator version:

```
; [List-of Number] -> Number
; compute the sum of the numbers onalon0

(check-expect (sum.v2 '(10 4 6)) 20)

(define (sum.v2 alon0)
  (local (; [List-of Number] ??? -> Number
         ; compute the sum of the numbers onalon
         ; accumulator ...
         (define (sum/a alon a)
           (cond
             [(empty? alon) ...]
             [else (... (sum/a (rest alon) ... a ...) ...)])))
    (sum/a alon0 ...)))
```

As suggested by our first step, we have put the template for `sum/a` into a `local` definition, added an accumulator parameter, and renamed `sum 's` parameter.

Here are two side-by-side sketches of hand evaluations:

$\begin{aligned} (\text{sum } '(10 \ 4)) &= \\ &= (+ \ 10 \ (\text{sum } '(4))) \end{aligned}$	$\begin{aligned} (\text{sum}.v2 \ '(10 \ 4)) &= \\ &= (\text{sum}/a \ '(10 \ 4) \ a0) \end{aligned}$
--	--

```

== (+ 10 (+ 4 (sum '())))
== (+ 10 (+ 4 (+ 0)))
...
== 20.0

```

```

== (sum/a '(4) ... 10 ... a0)
== (sum/a '() ... 4 ... 10 ... a0)
...
== 20.0

```

A comparison immediately suggests the central idea, namely, that `sum/a` can use the accumulator to add up the numbers as it encounters.

Concerning the accumulator invariant, this analysis suggests `a` represents the sum of the numbers encountered so far:

`a` represents the sum of the numbers that `alon` lacks in comparison to `alon0`

For example, if

```

alon0 = '(10 4 6)
alon =      '(6)

```

the invariant forces the accumulator's value to be 14. In contrast, when

```

alon0 = '(10 4 6)
alon =      '()

```

`a` must be 20.

Given this precise invariant, the rest of the design is straightforward again:

```

(define (sum.v2 alon0)
  (local (; [List-of Number] ??? -> Number
          ; compute the sum of the numbers on alon
          ; accumulator a represents the sum of the numbers
          ; that alon lacks in comparison to alon0
          (define (sum/a alon a)
            (cond
              [(empty? alon) a]
              [else (sum/a (rest alon) (+ (first alon) a))]))
        (sum/a alon0 0)))

```

If `alon` is `'()`, `sum/a` returns `a` because it represents the sum of all numbers on `alon`. The invariant also implies that `0` is the initial value for `a0` and `+` updates the accumulator by adding the number that is about to be “forgotten”—(`first alon`)—to the accumulator `a`.

Exercise 498. Explain why the natural recursion maintains the correctness of the accumulator statement:

```
(sum/a (rest alon) (+ (first alon) a))
```

Study the above examples before you formulate a general argument. ■

Exercise 499. Complete the above manual evaluation of

```
(sum/a '(10 4 6))
```

Doing so shows that the `sum` and `sum.v2` add up the given numbers in reverse order. While `sum` adds up the numbers from right to left, the accumulator-style version adds them up from left to right.

Note on Numbers Remember that for exact numbers, this difference has no effect on the

final result. For inexact numbers, the difference can be significant. See the exercises at the end of [Intermezzo: The Cost of Computation](#). ■

For the second example, we turn to the well-known factorial function:

The factorial function is useful for the area of algorithmic analysis.

```
; N -> N
; compute (* n (- n 1) (- n 2) ... 1)

(check-expect (! 3) 6)

(define (! n)
  (cond
    [(zero? n) 1]
    [else (* n (! (sub1 n))))]))
```

While `relative-2-absolute` and `invert` processed lists, the factorial function works on natural numbers. Its template is that for `N` processing functions.

We proceed as before with a template for an accumulator-style version:

```
; N -> N
; compute (* n0 (- n0 1) (- n0 2) ... 1)

(check-expect (!.v2 3) 6)

(define (!.v2 n0)
  (local (; N ??? -> N
          ; compute (* n (- n 1) (- n 2) ... 1)
          ; accumulator ...
          (define (/a n a)
            (cond
              [(zero? n) ...]
              [else (... (/a (sub1 n) ... a ...) ...)])))
    (!/a n0 ...)))
```

followed by a sketch of a hand evaluation:

$(! 3) =$ $\quad\quad\quad == (* 3 (! 2))$ $\quad\quad\quad == (* 3 (* 2 (! 1)))$ $\quad\quad\quad == (* 3 (* 2 (* 1 (! 0))))$ $\quad\quad\quad == (* 3 (* 2 (* 1 1)))$ $\quad\quad\quad \dots$ $\quad\quad\quad == 6$	$(!.v2 3) =$ $\quad\quad\quad == (!/a 3 a0)$ $\quad\quad\quad == (!/a 2 ... 3 ... a0)$ $\quad\quad\quad == (!/a 1 ... 2 ... 3 ... a0)$ $\quad\quad\quad == (!/a 0 ... 1 ... 2 ... 3 ... a0)$ $\quad\quad\quad \dots$ $\quad\quad\quad == 6$
--	---

The left column shows how the original version works, the right one sketches how the accumulator-style function proceeds. Both traverse the natural number until they reach `0`. While the original version schedules only multiplications, the accumulator keeps track of each number as the structural processing descends through the given natural number.

Given the goal of multiplying these numbers, `!/a` can use the accumulator to multiply the

numbers immediately:

`a` is the product of the natural numbers in the interval $[n_0, n]$.

In particular, when n_0 is 3 and n is 1, `a` is 6.

Exercise 500. What should the value of `a` be when n_0 is 3 and n is 1? How about when n_0 is 10 and n is 8?

Using this invariant we can easily pick the initial value for `a`—it is 1—and we know that multiplication the current accumulator with n is the proper update operation:

```
(define (!.v2 n0)
  (local (; N N -> N
          ; compute (* n (- n 1) (- n 2) ... 1)
          ; accumulator a is the product of the natural
          ; numbers in the interval [n0, n].
          (define (!/a n a)
            (cond
              [(zero? n) a]
              [else (!/a (sub1 n) (* n a))])))
  (!/a n0 1)))
```

It also follows from the accumulator statement that when n is 0, the accumulator is the product of n through 1, meaning it is the desired result. So, like `sum`, `!/a` returns `a` in this case and uses the result of the recursion in the second case.

Exercise 501. Like `sum`, `!` performs the primitive computation steps—multiplication in this case—in reverse order. Surprisingly, this affects the performance of the function in a negative manner.

Measure how long it takes to evaluate `(! 20)` one thousand times. Recall that `(time an-expression)` function determines how long it takes to run `an-expression`.

For the third and last example, we use a function that measures the height of simplified binary trees. The example illustrates that accumulator-style programming applies to all kinds of data, not just those defined with single self-references. Indeed, it is as common for complicated data definitions as it is for lists and natural numbers.

Here are the relevant definitions:

```
(define-struct node [left right])
; A Tree is one of:
; - '()
; - (make-node Tree Tree)

(define example (make-node (make-node '() (make-node '() '())) '()))
```

These trees carry no information; their leafs are `'()`. Still, there are many different trees as [figure 141](#) shows; it also uses suggestive graphics to bring across what these pieces of data look like as trees.

One property one may wish to compute is the height of such a tree:

```
; Tree -> N
; measure the height of abt0
```

```
(check-expect (height example) 3)

(define (height abt)
  (cond
    [(empty? abt) 0]
    [else (+ (max (height (node-left abt)) (height (node-right abt))) 1)]))
```

The table in [figure 141](#) indicates how to measure the height of a tree though it leaves the notion somewhat ambiguous: it is either the number of nodes from the root of the tree to the highest leaf or the number of connections on such a path. The `height` function follows the second option.

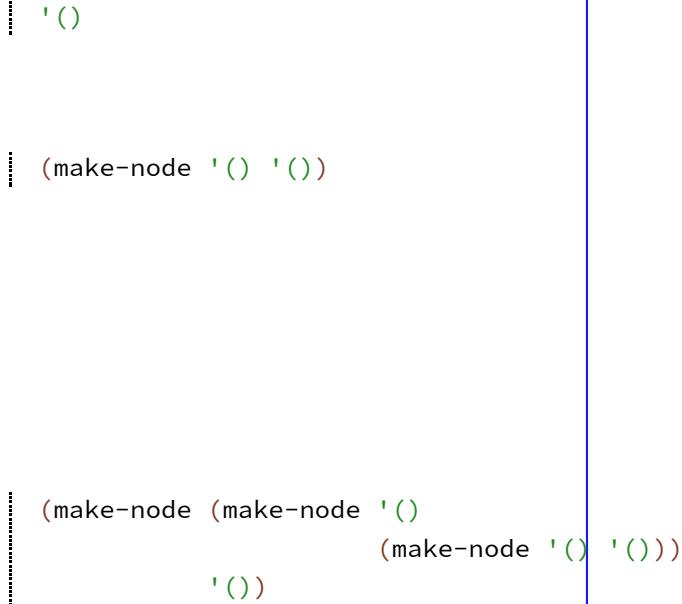


Figure 141: Some stripped-down binary trees

To transform this function into an accumulator-style function, we follow the standard path. We begin with an appropriate template:

```
; Tree -> N
; measure the height of abt0

(check-expect (height.v2 example) 3)

(define (height.v2 abt0)
  (local (; Tree ??? -> N
         ; measure the height of abt
         ; accumulator ...
         (define (height/a abt a)
           (cond
             [(empty? abt) ...]
             [else
              (... (height/a (node-left abt) ... a ...) ...
                   ... (height/a (node-right abt) ... a ...) ...)]))))
```

```
(height/a abt0 ...))
```

As always, the problem is to determine what knowledge the accumulator represents. One obvious choice is the number of traversed branches:

a is the number of steps it takes to reach abt from abt0 .

Illustrating this accumulator invariant is best done with a graphical example. Take a second look at [figure 141](#). The bottom-most tree comes with two annotations, each pointing out one subtree:

1. If abt0 is the complete tree and abt is the subtree pointed to by the circled 1, the accumulator's value must be 1 because it takes exactly one step to get from the root of abt to the root of abt0 .
2. In the same spirit, for the subtree labeled 2 the accumulator is 2 because it takes two steps to get this place.

As for the preceding two examples, the invariant basically dictates how to follow the rest of the design recipe for accumulators: the initial value for a is 0; the update operation is `add1`; and the base case uses the accumulated knowledge by returning it. Translating this into code yields the following skeleton definition:

```
(define (height.v2 abt0)
  (local (; Tree N -> N
          ; measure the height of abt
          ; accumulator a is the number of steps
          ; it takes to reach abt from abt0
          (define (height/a abt a)
            (cond
              [(empty? abt) a]
              [else
                (... (height/a (node-left abt) (+ a 1)) ...
                      ... (height/a (node-right abt) (+ a 1)) ...)])))
  (height/a abt0 0)))
```

But, in contrast to the first two examples, a is not the final result. In the second `cond` clause, the two recursive calls yield two values. The design recipe for structural functions dictate that we combine those in order to formulate an answer for this case; the dots above indicate that we still need to pick an operation that combines these values.

Following the design recipe also tells us that we need to interpret the two values to find the appropriate function. According to the purpose statement for `height/a`, the first value is the height of the left subtree, and the second one is the height of the right one. Given that we are interested in the height of abt itself and that the height is the largest number of steps it takes to reach a leaf, we use the `max` function to pick the proper one; see [figure 142](#) for the complete definition.

```
; Tree -> N
; measure the height of abt0

(check-expect (height.v2 example) 3)

(define (height.v2 abt0)
  (local (; Tree N -> N
          ; measure the height of abt
```

```

; accumulator a is the number of steps
; it takes to reach abt from abt0
(define (height/a abt a)
  (cond
    [(empty? abt) a]
    [else
      (max (height/a (node-left abt) (+ a 1))
            (height/a (node-right abt) (+ a 1))))]))
(height/a abt0 0)))

```

Figure 142: The accumulator-style version of *height*

Note on an Alternative Design In addition to counting the number of steps it takes to reach a node, an accumulator function could hold on to the largest height encountered so far. Here is the accumulator statement for the design idea:

the first accumulator, a, represents the number of steps it takes to reach abt from abt0 and the second accumulator, stands for the tallest branch in the part of abt0 that is to the left of abt.

Clearly, this statement assumes a template with two accumulator parameters:

```

(define (height.v3 abt0)
  (local (; Tree N N -> N
          ; measure the height of abt
          ; accumulator s is the number of steps
          ; it takes to reach abt from abt0
          ; accumulator m is the maximal height of
          ; the part of abt0 that is to the left of abt
          (define (h/a abt s m)
            (cond
              [(empty? abt) ...]
              [else
                (... (h/a (node-left abt) ... s ...) ... m ...)
                  ... (h/a (node-right abt) ... s ...) ... m ...)])))
        (h/a abt0 ...)))

```

Exercise 502. Complete the design of *height.v3*.

Hint In terms of the bottom-most tree of [figure 141](#), the place marked 1 has no complete paths to leafs to its left while the place marked 2 has one complete path and it consists of two steps.

This second design has a more complex accumulator invariant than the first one. By implication, its implementation requires more care than the first one. At the same time, it comes without any advantages, meaning it is inferior to the first one.

Our point is that different accumulator invariants yield different variants. You can

v.6.3.0.2

Epilogue: How Not to Program

*ROS: I mean, what exactly do you **do**?*

PLAYER: We keep to our usual stuff, more or less, only inside out. We do on stage things that are supposed to happen off. Which is a kind of integrity, if you look on every exit as being an entrance somewhere else

—Tom Stoppard **Rosencrantz and Guildenstern are Dead**

We have reached the end of this introduction to computing and program design. While there is more to learn about both subjects, this is a good point to stop, to summarize, and to look ahead.

Computing

In elementary school, you learned to calculate with numbers. At first you used numbers to count real things: three apples, five friends, twelve bagels. A bit later you encountered addition, subtraction, multiplication, and even division. Middle school introduced fractions, but it was still all about numbers. In high school, latest, you found out about variables and functions. Once again, variables were for numbers and functions related numbers to numbers. They called it **algebra**.

Because you used numbers all the way from first grade through senior year in high school, you didn't think much of numbers as a means to represent information about the real world. Yes, you had started with three bears, five wolves, and twelve horses, but by high school nobody reminded you of this relationship.

When you move from mathematical calculations to computing, the mapping from information to data and back becomes central. Nowadays computer programs process representations of music, videos, molecules, chemical compounds, business case studies, electrical diagrams, and blue prints. Fortunately, you don't need to encode all this information with numbers (or worse just two numbers: `0` and `1`) when you program; life would be unimaginable tedious otherwise. Instead, computing generalizes arithmetic and algebra so that your programs can compute with strings, booleans, characters, structures and even functions of your choice.

Like numbers, these forms of data come with basic operations. To compute means to apply these functions to data. The computation obeys laws with equational laws explaining how these operations process data.

Computing also means running functions, which like in mathematics, combine basic operations and other functions. There are two fundamental combination mechanisms: function composition and conditional expressions. The former means that the result of one function becomes the argument of another one. The latter represents a choice among several possibilities. Each of these combination mechanisms comes with a law that governs how computations proceed when they encounter a function combination.

Programs consist of many functions, sometimes thousands and tens of thousands, and to run a program means to apply one of these functions. Using the laws of data and function

combination, any programmer can, in principle, understand how any program processes its inputs and produces its outputs. But people are too slow at this task when it involves huge volumes of data and large numbers of functions and operations. Instead they leave the actual computing to computers, which are extremely fast and good at using these generalized laws of arithmetic and algebra.

Programming

Programmers design programs, meaning data representations and functions. Some of these functions are plain structural traversals, a few use generative recursion. Many of them are compositions of functions. While programmers occasionally compose their own functions, most often they use other programmer's functions.

A typical programming project requires the collaboration of many programmers. Each programmer contributes one or more components to the system, which from the perspective of this book, means a collection of functions. Since the life span of software systems also comprises many years, it is common that some programmers leave and others join project teams during this period.

In such a dynamic context, programmers cannot hope to produce high quality software without a strong discipline. The key is to understand the design of programs as a means of communication among programmers; the goal is to describe computations so that others can easily read and comprehend the code. For that reason, the design of every program and every piece of a program must rely on a method that produces code in a systematic manner. Thus when others approach this code, its very shape and organization conveys the underlying ideas.

The design recipe of this book is one of these methods. It starts with an analysis of the world of information and a description of the classes of data that represent the relevant information. Then you make a plan, a work list of functions needed. Iterative refinement dictates that you start with a subset of functions that quickly yields a (partially) working product. A client can interact with this product and make suggestions and wishes.

Designing a program, a component, or even just a function requires a rigorous understanding of what it computes. Unless you can describe the purpose of a piece of code with a concise statement, you cannot produce anything useful for future programmers. It always helps to make up and work through examples. To confirm that the program works at least on your examples, you turn these examples into a suite of tests. This test suite is even more important when it comes to future modifications of the program. Anyone who changes the code can re-run these tests and reconfirm that the program still works for the basic examples.

Eventually you will write and distribute real-world programs, meaning other programmers or perhaps real-world users get error messages from your code or find differences between expected behavior and actual behavior. In this situation, your immediate task is to formulate a test case for which your code fails. Then you work through this failure, modify the program, and re-run the complete test suite—which ensures that the remaining behavior is intact.

No matter how hard you work, a function or program isn't done the first time it passes the test suite. You must find time to inspect it for flaws, for violations of design concepts, for repetitions. If you find any patterns, form new abstractions or use existing abstractions to eliminate these patterns.

If you respect these guidelines, you will produce solid software. It will work because you

understand why and how it works. Others who must modify or enhance your software will understand it, because its shape communicates its development process. Still, to produce great software, you must practice, practice, practice. And you will have to learn a lot more about program design and computing than a first book can teach.

Exercise 533. Describe in a short essay how the design process applies to other professionals, say, architects, journalists, lawyers, photographers, or surgeons. ▀

Onward

This book uses a series of small teaching languages to introduce programming, not a full-featured programming language. Most importantly, teaching languages protect novice programmers from the incomprehensible error messages that come with real-world languages. At the same time, the careful selection of minimal features ensures that you can easily adapt the program design recipe to other languages.

As a student of program design, your next task is to learn how the design recipe applies in the setting of a full-fledged programming language. Such a language typically offers means for spelling out data definitions

(*classes* and *objects*) and for formulating signatures so that they are cross-checked before the program is run (*types*). In addition, you will also have to learn how to scale the design recipe to the use and production of so-called frameworks and components. Roughly speaking, frameworks abstract pieces of functionality that are common to many software systems, for example, graphical user interfaces, database connections, web connectivity and so on. You need to learn to instantiate these abstractions, and your programs will compose these instances to create coherent systems. Conversely, the creation of systems also calls for their organization into components that bundle pieces of functionality. Learning to create such components, is inherently a part of scaling up your skills.

Given your knowledge, it is easy for you to learn Racket, the language behind the teaching languages in this book. See [Realm of Racket](#) for one possible introduction.

As a student of computing, you will also have to expand your understanding of computing. This book has focused on the laws that describe computing processes. In order to function as a real software engineer, you need to learn what computations costs, both at a theoretical level and a practical one. The concept of big-O is a first step in this direction; being able to measure a program's performance and to allocate time consumption to its pieces is another one. Above and beyond these basic ideas, you will need knowledge about hardware, networking, layering of software, and specialized algorithms in various disciplines.

Some of you wanted to see what computer science was about, and you may never have seen a future in computing for yourself. You found out how computing is a natural outgrowth of school mathematics and that programming is all about systematic problem solving. Now, whether you become an accountant, a doctor, or a lab technician, you will have to solve problems and the design recipe will help you. You will see that the recipe's process dimension can serve as a guide in many situations and that abstraction—creating a single point of control—can reduce labor in equally many situations. So if you remember one idea from this book, as a future programmer or not,

remember the design recipe, wherever you go.

Onward!