

v.6.11.0.4

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](#)].

Released on Saturday, January 6th, 2018 10:01:30pm

Preface

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

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

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

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

everyone can design programs

and

everyone can experience the satisfaction that comes with creative design.

Indeed, we go even further and argue that

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

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

Systematic Program Design

A program interacts with people, dubbed *users*, and other programs, in which case we speak of *server* and *client* components. Hence any reasonably complete program consists of many building blocks: some deal with input, some create output, while some bridge the gap between

those two. We choose to use functions as fundamental building blocks because everyone encounters functions in pre-algebra and because the simplest programs are just such functions. The key is to discover which functions are needed, how to connect them, and how to build them from basic ingredients.

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

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

1. From Problem Analysis to Data Definitions

Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

2. Signature, Purpose Statement, Header

State what kind of data the desired function consumes and produces. Formulate a concise answer to the question *what* the function computes. Define a stub that lives up to the signature.

3. Functional Examples

Work through examples that illustrate the function’s purpose.

4. Function Template

Translate the data definitions into an outline of the function.

5. Function Definition

Fill in the gaps in the function template. Exploit the purpose statement and the examples.

6. Testing

Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

Figure 1: The basic steps of a function design recipe

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

The function-level design recipes share a common **design process**. [Figure 1](#) displays its six essential steps. The title of each step specifies the expected outcome(s); the “commands” suggest the key activities. Examples play a central role at almost every stage. For the chosen data representation in step 1, writing down examples proves how real-world information is encoded as data and how data is interpreted as information. Step 3 says that a problem-solver must work

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

through concrete scenarios to gain an understanding of what the desired function is expected to compute for specific examples. This understanding is exploited in step 5, when it is time to define the function. Finally, step 6 demands that examples are turned into automated test code, which ensures that the function works properly for some cases. Running the function on real-world data may reveal other discrepancies between expectations and results.

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

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

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

Iterative Refinement addresses the issue that problems are complex and multifaceted. Getting everything right at once is nearly impossible. Instead, computer scientists borrow iterative refinement from the physical sciences to tackle this design problem. In essence, iterative refinement recommends stripping away all inessential details at first and finding a solution for the remaining core problem. A refinement step adds in one of these omitted details and re-solves the expanded problem, using the existing solution as much as possible. A repetition, also called an iteration, of these refinement steps eventually leads to a complete solution.

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

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

DrRacket and the Teaching Languages

Learning to design programs calls for repeated hands-on practice. Just as nobody becomes a piano player without playing the piano, nobody becomes a program designer without creating

actual programs and getting them to work properly. Hence, our book comes with a modicum of software support: a language in which to write down programs and a *program development environment* with which programs are edited like word documents and with which readers can run programs.

Many people we encounter tell us they wish they knew how to code and then ask *which programming language* they should learn. Given the press that some programming languages get, this question is not surprising. But it is also wholly inappropriate. Learning to program in a currently fashionable programming language often sets up students for eventual failure. Fashion in this world is extremely short lived. A typical “quick programming in X” book or course fails to teach principles that transfer to the next fashion language. Worse, the language itself often distracts from the acquisition of transferable skills, at the level of both expressing solutions and dealing with programming mistakes.

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

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

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

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

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

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

When it comes to programming environments, we face an equally bad choice as the one for languages. A programming environment for professionals is analogous to the cockpit of a jumbo jet. It has numerous controls and displays, overwhelming anyone who first launches such a software application. Novice programmers need the equivalent of a two-seat, single-

engine propeller aircraft with which they can practice basic skills. We have therefore created DrRacket, a programming environment for novices.

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

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

Skills that Transfer

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

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

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

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

1. analyze a problem statement, typically stated as a word problem;
2. extract and express its essence, abstractly;

3. illustrate the essence with examples;
4. make outlines and plans based on this analysis;
5. evaluate results with respect to expected outcomes; and
6. revise the product in light of failed checks and tests.

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

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

This Book and Its Parts

The purpose of this book is to introduce readers without prior experience to the *systematic design of programs*. In tandem, it presents a *symbolic view of computation*, a method that explains how the application of a program to data works. Roughly speaking, this method generalizes what students learn in elementary school arithmetic and middle school algebra. But have no fear. DrRacket comes with a mechanism—the algebraic stepper—that can illustrate these step-by-step calculations.

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

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

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

- [Intermezzo 1: Beginning Student Language](#) describes the teaching language in complete detail: its vocabulary, its grammar, and its meaning. Computer scientists refer to these as syntax and semantics. Program designers use this model of computation to predict what their creations compute when run or to analyze error diagnostics.
- [Arbitrarily Large Data](#) extends [Fixed-Size Data](#) with the means to describe the most interesting and useful forms of data: arbitrarily large compound data. While a programmer may nest the kinds of data from [Fixed-Size Data](#) to represent information, the nesting is always of a fixed depth and breadth. This part shows how a subtle generalization gets us from there to data of arbitrary size. The focus then switches to the systematic design of programs that process this kind of data.
- [Intermezzo 2: Quote, Unquote](#) introduces a concise and powerful notation for writing down large pieces of data: quotation and anti-quotation.
- [Abstraction](#) acknowledges that many of the functions from [Arbitrarily Large Data](#) look alike. No programming language should force programmers to create pieces of code that are so similar to each other. Conversely, every good programming language comes with ways to eliminate such similarities. Computer scientists call both the step of eliminating similarities and its result *abstraction*, and they know that abstractions greatly increase a programmer's productivity. Hence, this part introduces design recipes for creating and using abstractions.
- [Intermezzo 3: Scope and Abstraction](#) plays two roles. On the one hand, it injects the concept of *lexical scope*, the idea that a programming language ties every occurrence of a name to a definition that a programmer can find with an inspection of the code. On the other hand, it explains a library with additional mechanisms for abstraction, including so-called *for loops*.
- [Intertwined Data](#) generalizes [Arbitrarily Large Data](#) and explicitly introduces the idea of iterative refinement into the catalog of design concepts.
- [Intermezzo 4: The Nature of Numbers](#) explains and illustrates why decimal numbers work in such strange ways in all programming languages. Every budding programmer ought to know these basic facts.
- [Generative Recursion](#) adds a new design principle. While structural design and abstraction suffice for most problems that programmers encounter, they occasionally lead to insufficiently “performant” programs. That is, structurally designed programs might need too much time or energy to compute the desired answers. Computer scientists therefore replace structurally designed programs with programs that benefit from ad hoc insights into the problem domain. This part of the book shows how to design a large class of just such programs.
- [Intermezzo 5: The Cost of Computation](#) uses examples from [Generative Recursion](#) to illustrate how computer scientists think about performance.
- [Accumulators](#) adds one final trick to the toolbox of designers: accumulators. Roughly speaking, an accumulator adds “memory” to a function. The addition of memory greatly improves the performance of structurally designed functions from the first four parts of the book. For the ad hoc programs from [Generative Recursion](#), accumulators can make the difference between finding an answer and never finding one.

[Epilogue: Moving On](#) is both an assessment and a look ahead to what's next.

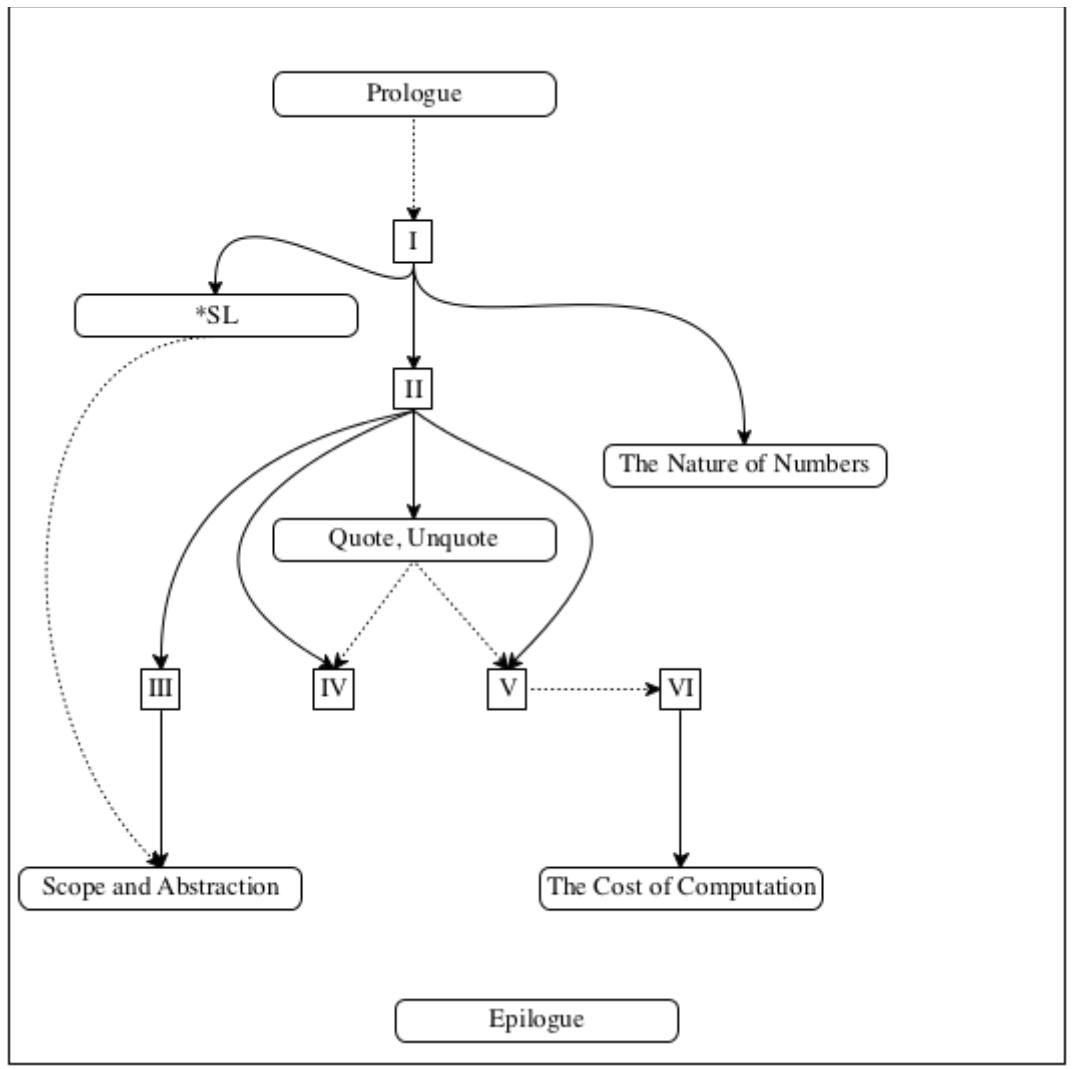


Figure 2: The dependencies among parts and intermezzos

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

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

[Figure 2](#) is a navigation chart for those who wish to pick and choose from the elements of the book. The figure is a dependency graph. A solid arrow from one element to another suggests a mandatory ordering; for example, Part II requires an understanding of Part I. In contrast, a dotted arrow is mostly a suggestion; for example, understanding the Prologue is unnecessary to get through the rest of the book.

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

- A high school instructor may want to cover (as much as possible of) parts I and II, including a small project such as a game.

- A college instructor in a quarter system may wish to focus on [Fixed-Size Data](#), [Arbitrarily Large Data](#), [Abstraction](#), and [Generative Recursion](#), plus the intermezzos on *SL and scope.
- A college instructor in a semester system may prefer to discuss performance trade-offs in designs as early as possible. In this case, it is best to cover [Fixed-Size Data](#) and [Arbitrarily Large Data](#) and then the accumulator material from [Accumulators](#) that does not depend on [Generative Recursion](#). At that point, it is possible to discuss [Intermezzo 5: The Cost of Computation](#) and to study the rest of the book from this angle.

Iteration of Sample Topics The book revisits certain exercise and sample topics time and again. For example, virtual pets are found all over [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 such sequences.

The Differences

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

1. It explicitly acknowledges the difference between designing a whole program and the functions that make up a program. Specifically, this edition focuses on two kinds of programs: event-driven (mostly GUI, but also networking) programs and batch programs.
2. The design of a program proceeds in a top-down planning phase followed by a bottom-up construction phase. We explicitly show how the interface to libraries dictates the shape of certain program elements. In particular, the very first phase of a program design yields a wish list of functions. While the concept of a wish list exists in the first edition, this second edition treats it as an explicit design element.
3. Fulfilling an entry from the wish list relies on the function design recipe, which is the subject of the six major parts.
4. A key element of structural design is the definition of functions that compose others. This design-by-composition is especially useful for the world of batch programs. Like generative recursion, it requires a *eureka!*, specifically a recognition that the creation of intermediate data by one function and processing this intermediate result by a second function simplifies the overall design. This approach also needs a wish list, but formulating these wishes calls for an insightful development of an intermediate data definition. This edition of the book weaves in a number of explicit exercises on design by composition.

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

6. This edition of the book drops the design of imperative programs. The old chapters remain available on-line. An adaptation of this material will appear in the second volume of this series, *How to Design Components*.
7. The book's examples and exercises employ new teachpacks. The preferred style is to link in these libraries via `require`, but it is still possible to add teachpacks via a menu in DrRacket.
8. Finally, this second edition differs from the first in a few aspects of terminology and notation:

Second Edition	First Edition
<code>signature</code>	<code>contract</code>
<code>itemization</code>	<code>union</code>
<code>'()</code>	<code>empty</code>
<code>#true</code>	<code>true</code>
<code>#false</code>	<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 University’s introductory course with the first author; Daniel P. Friedman, for asking the first author to rewrite *The Little LISPer* (also MIT Press) in 1984, because it started this project; John Clements, who designed, implemented, and maintains DrRacket’s stepper; and Paul Steckler, who faithfully supported the team with contributions to our suite of programming tools.

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

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

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

Acknowledgments

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

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

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

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

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

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

We also thank the following: Ennas Abdussalam, Mark Aldrich, Anisa Anuar, Saad Bashir, Aaron Bauman, Suzanne Becker, Steven Belknap, Stephen Bloch, Elijah Botkin, Joseph Bogart William Brown, Tomas Cabrera, Xuyuqun C, Colin Caine, Anthony Carrico, Rodolfo Carvalho, Estevo Castro, Maria Chacon, Stephen Chang, Tung Cheng, Nelson Chiu, Jack Clay, Richard Cleis, John Clements, Scott Crymble, Pierce Darragh, Jonas Decraecker, Qu Dongfang, Mark Engelberg, Andrew Fallows, Jiankun Fan Christopher Felleisen, Sebastian Felleisen, Vladimir Gajić, Xin Gao, Adrian German, Jack Gitelson, Kyle Gillette, Scott Greene, Ben Greenman, Ryan Golbeck, Josh Grams, Grigoris, Jane Griscti, Alberto Eleuterio Flores Guerrero, Tyler Hammond, Nan Halberg, Li Junsong, Nadeem Abdul Hamid, Jeremy Hanlon, Craig Holbrook, Connor Hetzler, Wayne Iba, John Jackaman, Jordan Johnson, Blake Johnson, Erwin Junge, Marc Kaufmann, Cole Kendrick, Gregor Kiczales, Eugene Kohlbecker, Caitlin Kramer, Roman Kunin Jackson Lawler, Devon LePage, Ben Lerner, Shicheng Li, Chen Lj, Ed Maphis, YuSheng Mei, Andres Meza, Saad Mhmood, Elena Machkasova, Jay Martin, Jay McCarthy, James McDonell, Mike McHugh, Wade McReynolds, David Moses, Ann E. Moskol, Scott Newson, Paul Ojanen, Prof. Robert Ordóñez, Laurent Orseau, Klaus Ostermann, Alanna Pasco, S. Pehlivanoğlu, Eric Parker, Nick Pleatsikas, Prathyush Pramod, Alok Rai, Norman Ramsey, Krishnan Ravikumar, Jacob Rubin, Ilnar Salimzianov, Luis Sanjuán, Brian Schack, Ryan "Havy" Scheel, Lisa Scheuing, Willi Schiegel, Vinit Shah, Nick Shelley, Edward Shen, Tubo Shi, Matthew Singer, Stephen Siegel, Milton Silva, Kartik Singhal, Joe Snikeris, Marc Smith, Matthijs Smith, Dave Smylie, Vincent St-Amour, Reed Stevens, Kevin Sullivan, Asumu Takikawa, Éric Tanter, Sam Tobin-Hochstadt, Thanos Tsouanas, Aaron Tsay, Mariska Twaalhoven, Bor Gonzalez Usach, Manuel del Valle, David Van Horn, Nick Vaughn, Simeon Veldstra, Andre Venter, Jan Vitek, Marco Villotta, Mitch Wand, Yuxu (Ewen) Wang, Michael Wijaya, G. Clifford Williams, Ewan Whittaker-Walker, Julia Wlochowski, Roelof Wobben, Mardin Yadegar, Huang Yichao, Yuwang Yin, Andrew Zipperer for comments on drafts of this second edition.

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

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

Prologue: How to Program

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

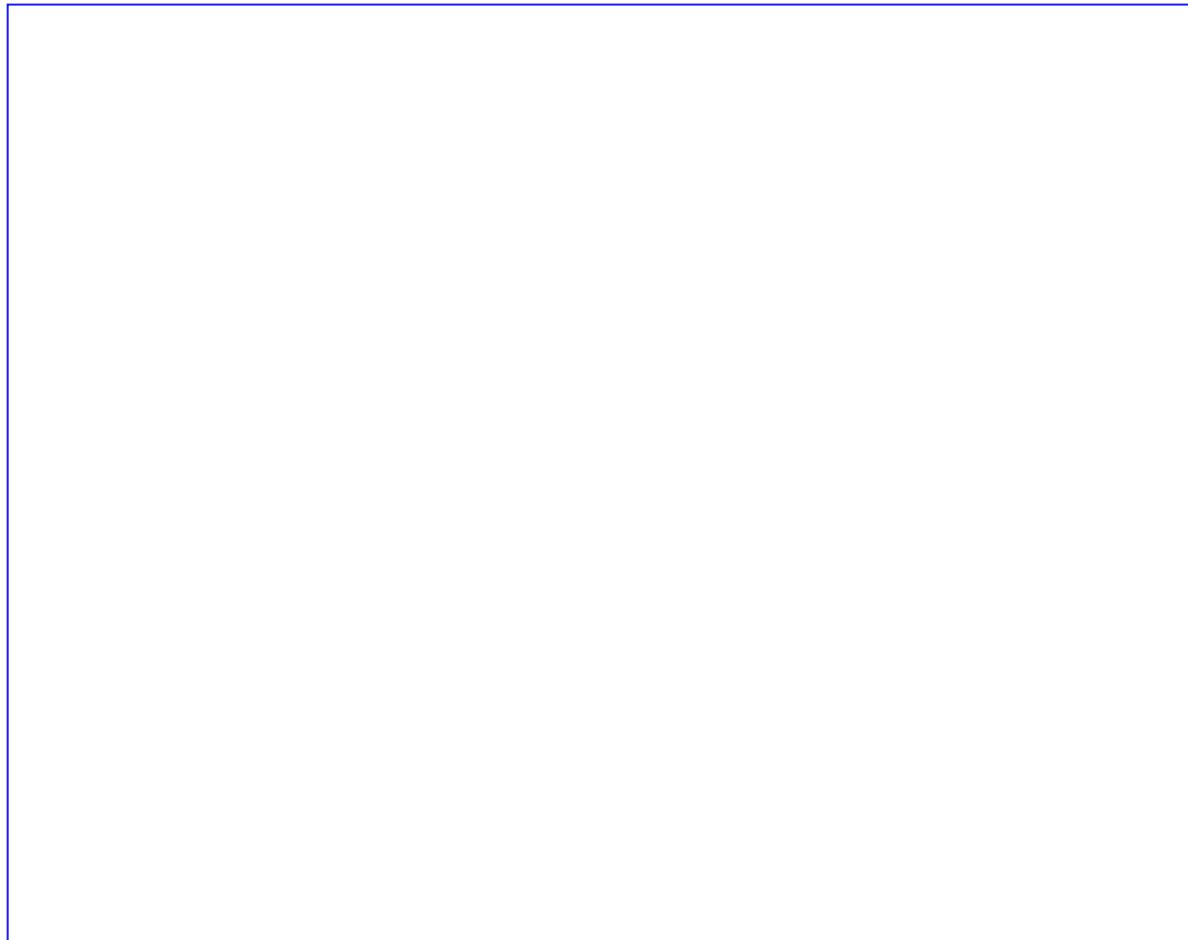
Consider a quick look at [On Teaching Part I.](#)

Download DrRacket from [its web site](#).

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

```
(+ 1 1)
```

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



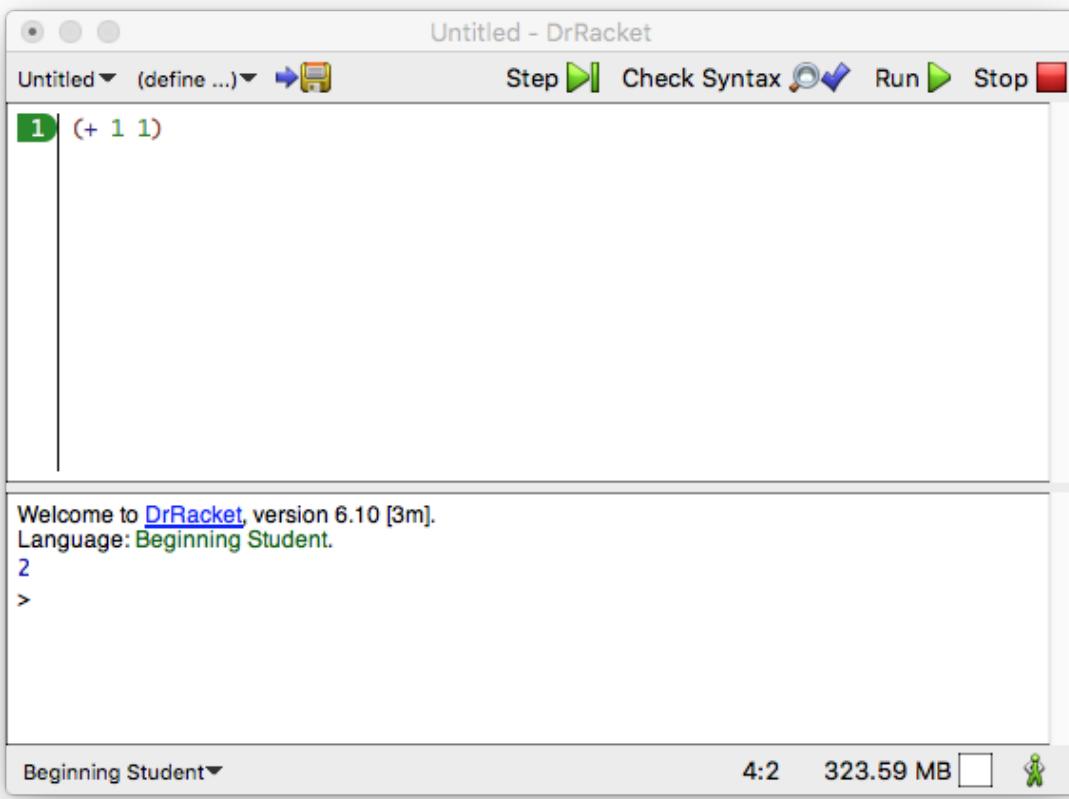


Figure 3: Meet DrRacket

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)
```

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

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

- The top half of DrRacket is called the *definitions area*. In this area, you create the programs, which is called *editing*. As soon as you add a word or change something in the definitions area, the *SAVE* button shows up in the top-left corner. When you click *SAVE* for the first time, DrRacket asks you for the name of a file so that it can store your program for good. Once your definitions area is associated with a file, clicking *SAVE* ensures that the content of the definitions area is stored safely in the file.
- *Programs* consist of *expressions*. You have seen expressions in mathematics. For now, an expression is either a plain number or something that starts with a left parenthesis "(" and ends in a matching right parenthesis ")"—which DrRacket rewards by shading the area between the pair of parentheses.

- When you click *RUN*, DrRacket evaluates the expressions in the definitions area and shows their result in the *interactions area*. Then, DrRacket, your faithful servant, awaits your commands at the *prompt* (>). The appearance of the prompt signals that DrRacket is waiting for you to enter additional expressions, which it then evaluates like those in the definitions area:

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

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

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

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

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

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

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

In BSL, every time you want to use a “calculator operation,” you write down an opening parenthesis, the operation you wish to perform, say `+`, the numbers on which the operation should work (separated by spaces or even line

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

breaks), and, finally, a closing parenthesis. The items following the operation are called the *operands*. Nested arithmetic means that you can use an expression for an operand, which is why

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

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

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

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

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

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

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

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

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

If you don’t know what an operation does for several operands, enter an example into the interactions area and hit “return”; DrRacket lets you know whether and how it works. Or use HelpDesk to read the documentation.

Second, when you read programs that others write, you will never have to wonder which expressions are evaluated first. The parentheses and the nesting will immediately tell you.

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

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

Arithmetic and Arithmetic

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

The first thing you need to know is that in BSL, text is any sequence of keyboard characters enclosed in double quotes ("").

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

We call it a string. Thus, "hello world" is a perfectly fine string; and when DrRacket evaluates this string, it just echoes it back in the interactions area, like a number:

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

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

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

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

Just like +, `string-append` is an operation; it makes a string by adding the second to the end of the first. As the first interaction shows, it does this literally, without adding anything between the two strings: no blank space, no comma, nothing. Thus, if you want to see the phrase "hello world", you really need to add a space to one of these words somewhere; that's what the second interaction shows. Of course, the most natural way to create this phrase from the two words is to enter

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

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

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

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

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

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

There is also an operation that converts strings into numbers:

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

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

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

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

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

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

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

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

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

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

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

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

What is the result of this expression? How did you figure it out? All by yourself? Or did you just type it into DrRacket’s interactions area and hit the “return” key? If you did the latter, do

you think you would know how to do this on your own? After all, if you can't predict what DrRacket does for small expressions, you may not want to trust it when you submit larger tasks than that for evaluation.

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. Or, 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.

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

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

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



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

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

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

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


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

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


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

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



Stop and reflect on this last result for a moment.

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

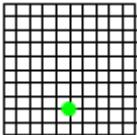
```
> (image-width (square 10 "solid" "red"))
10
> (image-width
  (overlay (rectangle 20 20 "solid" "blue")
           (circle 5 "solid" "red"))))
20
```

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

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

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

and you get this:



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

As you can see from this image, the origin (or $(0,0)$) is in the upper-left corner. Unlike in mathematics, the y-coordinate is measured **downward**, not upward. Otherwise, the image shows what you should have expected: a solid green disk at the coordinates $(50,80)$ in a 100 by 100 empty rectangle.

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

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

Inputs and Output

The programs you have written so far are pretty boring. You write down an expression or several expressions; you click *RUN*; you see some results. If you click *RUN* again, you see the exact same results. As a matter of fact, you can click *RUN* as often as you want, and the same

results show up. In short, your programs really are like calculations on a pocket calculator, except that DrRacket calculates with all kinds of data, not just numbers.

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

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

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

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

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

x =	1	2	3	4
y =	•	•	•	?

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

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

$$y = x \cdot x$$

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

y = 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 often see in school for functions is utterly misleading. In DrRacket, you therefore write functions a bit differently:

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

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

This second part means you must supply one number—for x —to determine a specific value for y . When you do, DrRacket plugs the value for x into the expression associated with the function. Here the expression is $(\star\ x\ x)$. 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)
```

Mathematics also calls $y(1)$ a function application, but your teachers 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) (\star x x))
```

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

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

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

- First,

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

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

- Second,

(FunctionName ArgumentExpression)

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

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

```
> (empty-scene 100 60)
```



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

```
> (place-image 50 23 (empty-scene 100 60))
```



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

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

```
> (place-image 50 20 (empty-scene 100 60))
```

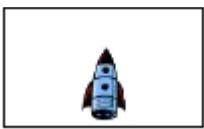


```
> (place-image 50 30 (empty-scene 100 60))
```



```
> (place-image 50 40 (empty-scene 100 60))
```





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

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

Figure 4: Landing a rocket (version 1)

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

```
(picture-of-rocket 0)
(picture-of-rocket 10)
(picture-of-rocket 20)
(picture-of-rocket 30)
```

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

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

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

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

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

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

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

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

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

[Exercise 298](#) explains how to design `animate`.

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

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

Many Ways to Compute

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

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

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

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

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

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

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

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

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

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

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

In general, a *conditional expression* has the shape

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

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

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

Figure 5: Landing a rocket (version 2)

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

See [figure 5](#) for the revised function definition. The definition uses the name `picture-of-rocket.v2` to distinguish the two versions. Using distinct names also allows us to use both

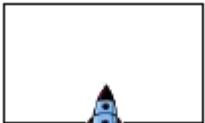
functions in the interactions area and to compare the results. Here is how the original version works:

```
> (picture-of-rocket 5555)
```



And here is the second one:

```
> (picture-of-rocket.v2 5555)
```



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

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

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

Stop! What do you think we want to see?

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

Putting one and one together you can now figure out that

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



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

Here is a first attempt:

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



Now replace the third argument in the above application with

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



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

```
(define (picture-of-rocket.v3 height)
  (cond
    [(<= height (- 60 (/ (image-height) 2)))
     (place-image (image-of-rocket) 50 height
                  (empty-scene 100 60))]
    [(> height (- 60 (/ (image-height) 2)))
     (place-image (image-of-rocket) 50 (- 60 (/ (image-height) 2))
                  (empty-scene 100 60)))]))
```

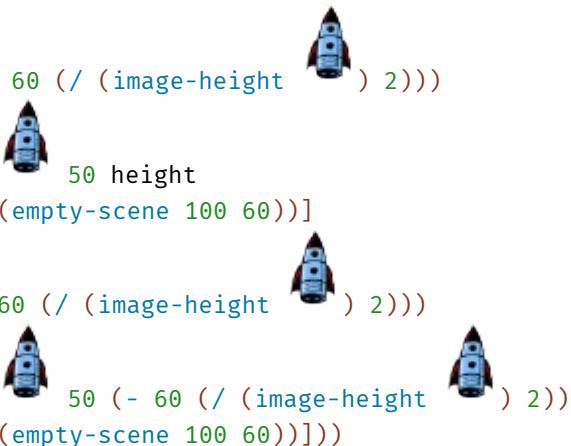


Figure 6: Landing a rocket (version 3)

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

One Program, Many Definitions

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

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

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

```
(define Name Expression)
```

Thus, for example, if you write down

```
(define HEIGHT 60)
```

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

```

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

(define WIDTH 100)
(define HEIGHT 60)

(define ROCKET )

```

Figure 7: Landing a rocket (version 4)

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

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

Confirm that the program still functions as before.

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

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

```
(- HEIGHT (/ (image-height ROCKET) 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 steps: (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:

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

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

Now substitute ROCKET-CENTER-TO-TOP for the expression `(- HEIGHT (/ (image-height ROCKET) 2))` in the rest of the program. You may be wondering whether this 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. For example,

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

causes DrRacket to complain that “CENTER is used before its definition,” 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 does not matter where you place constant definitions relative to function definitions. Indeed, if your program consists of many function definitions, their order doesn't matter either, though it is good to introduce all constant definitions first, followed by the definitions of functions in decreasing order of importance. When you start writing your own multi-definition programs, you will see why this ordering matters.

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


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

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

Figure 8: Landing a rocket (version 5)

Once you eliminate all repeated expressions, you get the program in [figure 8](#). It consists of one function definition and five constant definitions.

The program also contains two *line comments*, introduced with semicolons (“;”). While DrRacket ignores such comments, people who read programs

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:

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

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

One More Definition

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

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

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

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

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

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

$$d = v \cdot t$$

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

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

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

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

```
(define V 3)

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

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

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

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

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


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

```

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

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

```

Figure 9: Landing a rocket (version 6)

Now we can fix `picture-of-rocket` again. Instead of comparing `t` with a height, the function can use `(distance t)` to calculate how far down the rocket is. The final program is displayed in [figure 9](#). It consists of two function definitions: `picture-of-rocket.v6` and `distance`. The remaining constant definitions make the function definitions readable and modifiable. As always, you can run this program with `animate`:

```

> (animate picture-of-rocket.v6)

```

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

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

You Are a Programmer Now

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

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

In short, you have seen almost all the mechanics of putting together programs. If you read up on all the functions that are available, you can write programs that play interesting computer games, run simulations, or keep track of business accounts. The question is whether this really means you are a programmer. Are you?

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

Not!

When you look at the “programming” bookshelves in a random bookstore, you will see loads of books that promise to turn you into a programmer on the spot. Now that you have worked your way through some first examples, however, you probably realize that this cannot possibly happen.

Acquiring the mechanical skills of programming—learning to write expressions that the computer understands, getting to know which functions and libraries are available, and similar activities—isn’t helping you all that much with **real** programming. If it were, you could equally well learn a foreign language by memorizing a thousand words from the dictionary and a few rules from a grammar book.

Good programming is far more than the mechanics of acquiring a language. Most importantly, it is about keeping in mind that programmers create programs for other people to read them in the future. A good program reflects the problem statements and its important concepts. It comes with a concise self-description. Examples illustrate this description and relate it back to the problem. The examples make sure that the future reader knows why and how your code works. In short, good programming is about solving problems systematically and conveying the system within the code. Best of all, this approach to programming actually makes programming accessible to everyone—so it serves two masters at once.

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

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!

Note on What This Book Is *Not* About Many introductory books on programming contain a lot of material about the authors’ favorite application discipline for programming: puzzles, mathematics, physics, music, and so on. To some extent, including such material is natural because programming is obviously useful in all these areas. Then again, this kind of material distracts from proper programming and usually fails to tease apart the incidental from the essential. Hence 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.

I Fixed-Size Data

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

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

1 Arithmetic

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 “)”.

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

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
```

and

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

We use `==` to mean “is equal to according to the laws of computation.”

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 data into another one, retrieve “properties” of them, also in terms of atomic data, and so on. The sections of this chapter introduce some of these functions, also called *primitive operations* or *pre-defined operations*. You can find others in the documentation of BSL that comes with DrRacket.

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

1.1 The Arithmetic of Numbers

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

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

```
(+ 3 4)
```

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

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

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

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

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

When it comes to numbers, BSL programs

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

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

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

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

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

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

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

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

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

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

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

To develop the desired expression, it is best to click *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.

To confirm that the expression works properly, change `x` to `12` and `y` to `5`, then click *RUN*. 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. We will soon show you a way to define functions, which makes reusing 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 widespread prejudice about computers concerns their 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,” and the like.

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, plus a few others, about which we aren’t concerned just yet, enclosed in double quotes. In [Prologue: How to Program](#), we have seen a number of BSL strings: `"hello"`, `"world"`, `"blue"`, `"red"`, and others. 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" " 4 BSL")
"what a lovely day 4 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`, similar to those for `+`:

```
(+ 1 1) == 2  (string-append "a" "b") == "ab"
(+ 1 2) == 3  (string-append "ab" "c") == "abc"
(+ 2 2) == 4  (string-append "a" " ") == "a "
...
...
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Position means `i` characters from the left of the string, but programmers start counting at 0. Thus, the 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](#) to create an expression that deletes the `i`th position from `str`. Clearly this expression creates a shorter string than the given one. Which values for `i` are legitimate?

1.4 The Arithmetic of Images

An *Image* is a visual, rectangular piece of data, for example, a photo or a geometric figure and its frame. You can insert images in DrRacket wherever you can write down an expression because

images are values, just like numbers and strings.

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

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

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

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

Play with these operations in the interactions window:

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

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

> (star 12 "solid" "gray")

```

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

The second kind of functions on images concern image properties:

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

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

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

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

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

A proper understanding of the third kind of image-composing primitives requires the introduction of one new idea: the *anchor point*. An image isn't just a single pixel, it consists of many pixels.

Specifically, each image is like a photograph, that is, a rectangle of pixels. One of these pixels is an implicit anchor point. When you use an image primitive to compose two images, the composition happens with respect to the anchor points, unless you specify some other point explicitly:

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

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

- `empty-scene` creates a rectangle of some given width and height;
- `place-image` places an image into a scene at a specified position. If the image doesn’t fit into the given scene, it is appropriately cropped;
- `scene+line` consumes a scene, four numbers, and a color to draw a line into the given image. Experiment with it to see how it works.

arithmetic of numbers	arithmetic of images
<code>(+ 1 1) == 2</code>	<code>(overlay (square 4 "solid" "orange") (circle 6 "solid" "yellow"))</code>
	== 
<code>(+ 1 2) == 3</code>	<code>(underlay (circle 6 "solid" "yellow") (square 4 "solid" "orange"))</code>
	== 
<code>(+ 2 2) == 4</code>	<code>(place-image (circle 6 "solid" "yellow") 10 10 (empty-scene 20 20))</code>
	==  ...

Figure 10: Laws of image creation

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

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

Exercise 6. 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.

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
```

Unsurprisingly, `or` and `and` may be used with more than two expressions. Finally, there is more to `or` and `and` than these explanations suggest, but to explain the extra bit requires a second look at nested expressions.

Exercise 7. Boolean expressions can express some everyday problems. Suppose you want to decide whether today is an appropriate day to go to the mall. You go to the mall either if it is not sunny or

if today is Friday (because that is when stores post new sales items).

Nadeem Hamid suggested this formulation of the exercise.

Here is how you could go about it using your new knowledge about Booleans. First add these 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 combinations of Booleans can you associate with `sunny` and `friday`?

1.6 Mixing It Up with Booleans

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

```
(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 `(= x 0)`, `0`, and `(/ 1 x)`, three so-called *sub-expressions*. 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 is. Whatever their results are, they are also the result of the entire `if` expression.

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

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

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

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

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

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

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

```
(define x 0)
```

What do you think

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

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

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

Strings aren't compared with `=` and its relatives. Instead, you must use `string=?` or `string<=?` or `string>=?` if you ever need to compare strings. While it is obvious that `string=?` checks whether the two given strings are equal, the other two primitives are open to interpretation. Look up their documentation. Or, experiment, guess a general law, and then check in the documentation whether you guessed right.

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

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

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

```
(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 8. Add the following line to the definitions area:



```
(define cat )
```

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

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

1.7 Predicates: Know Thy Data

Remember the expression (`string-length 42`) and its result. Actually, the expression doesn't have a result, it signals an error. DrRacket lets you know about errors via red text in the interactions area and highlighting of the faulty expression (in the definitions area). This way of marking errors is particularly helpful when you use this expression (or its relatives) deeply nested within some other expression:

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

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

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

```
(define in ...)  
(string-length in)
```

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

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

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

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

```
(define in ...)

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

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

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

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

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

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

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

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

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

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

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

2 Functions and Programs

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

2.1 Functions

Programs are functions. Like functions, programs consume inputs and produce outputs. Unlike the functions you may know, programs work with a variety of data: numbers, strings, images, mixtures of all these, and so on. Furthermore, programs are triggered by events in the real world, and the

outputs of programs affect the real world. For example, a spreadsheet program may react to an accountant's key presses by filling some cells with numbers, or the calendar program on a computer may launch a monthly payroll program on the last day of every month. Lastly, a program may not consume all of its input data at once, instead it may decide to process data in an incremental manner.

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

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

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

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

That is, to define a function, we write down

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

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

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

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 its 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*, that is, the variables that follow the function name, are placeholders for **unknown** pieces of data, the inputs of the function. Mentioning a variable in the function body is the way to use these pieces of data when the function is applied and the values of the variables 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 space,
- and add “)” at the end.

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 three experiments, for example, confirm that `f` from above produces the same value no matter what input it is applied to:

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

What does `(f (circle 3 "solid" "red"))` yield?

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:

Remember to add `(require 2htdp/image)` to the definitions area.

```
> (f)
f:expects 1 argument, found none
> (f 1 2 3 4 5)
f:expects only 1 argument, 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, 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 11. Define a function that consumes two numbers, x and y , and that computes the distance of point (x,y) to the origin.

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

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

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

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

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

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

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

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

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

Exercise 19. Define the function `string-insert`, which consumes a string `str` plus a number `i` and inserts `"_"` at the `i`th position of `str`. Assume `i` is a number between `0` and the length of the

given string (inclusive). See [exercise 3](#) for ideas. Ponder how `string-insert` copes with "".

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

2.2 Computing

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

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

Here is a sample calculation for `f`:

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

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

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

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

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

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

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

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

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

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

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

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



Figure 11: The DrRacket stepper

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

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

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

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

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

```
(define (opening first-name last-name)
```

```
(string-append "Dear " first-name ",")
```

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

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

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

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

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

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

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

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

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

Does the explanation match your intuition?

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

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

Use the stepper to confirm your ideas.

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

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

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

Exercise 25. Take a look at this attempt to solve [exercise 17](#):

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

Does stepping through an application suggest a fix?

Exercise 26. What do you expect as the value of this 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

A program rarely consists of a single function definition. Typically, programs consist of a *main* definition and several other functions 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*.

```

(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" "\n"
    "last name " lst " have won our lottery. So, " "\n"
    fst ", " "hurry and pick up your prize."))

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

```

Figure 12: A batch program

Consider the program of [figure 12](#) for filling in letter templates. It consists of four functions. The first one is the main function, which produces a complete letter from the first and last name of the addressee plus a signature. The main function refers to three auxiliary functions to produce the three pieces of the letter—the opening, body, and signature—and composes the results in the correct order with `string-append`.

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

```

> (letter "Matthew" "Fisler" "Felleisen")
"Dear Matthew,\n\nWe have discovered that ... \n"

```

```
> (letter "Kathi" "Felleisen" "Findler")
"Dear Kathi,\n\nWe have discovered that ...\\n"
```

Aside The result is a long string that contains "\n", which represents a new line when the string is **printed**. Now Add (`require 2htdp/batch-io`) to your program, which adds the function `write-file` to its repertoire; it allows you to **print** this string to the console:

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

```
> (write-file 'stdout (letter "Matt" "Fiss" "Fell"))
Dear Matt,
We have discovered that all people with the
last name Fiss have won our lottery. So,
Matt, hurry and pick up your prize.
Sincerely,
Fell
'standard
```

[Programs](#) explains such batch programs in some depth. **End**

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 The owner of a monopolistic movie theater in a small town has complete freedom in setting ticket prices. The more he charges, the fewer people can afford tickets. The less he charges, the more it costs to run a show because attendance goes up. In a recent experiment the owner determined a relationship between the price of a ticket and average attendance.

At a price of \$5.00 per ticket, 120 people attend a performance. For each 10-cent change in the ticket price, the average attendance changes by 15 people. That is, if the owner charges \$5.10, some 105 people attend on the average; if the price goes down to \$4.90, average attendance increases to 135. Let's translate this idea into a mathematical formula:

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

Stop! Explain the minus sign before you proceed.

Unfortunately, the increased attendance also comes at an increased cost. Every performance comes at a fixed cost of \$180 to the owner plus a variable cost of \$0.04 per attendee.

The owner would like to know the exact relationship between profit and ticket price in order to maximize the profit.

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

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

1. The problem statement specifies how the number of attendees depends on the ticket price.

Computing this number is clearly a separate task and thus deserves its own function definition:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4 Global Constants

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

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

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

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

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

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

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

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

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

All functions in a program may refer to these global variables. A reference to a variable is just like using the corresponding constants. The advantage of using variable names instead of constants is

that a single edit of a constant definition affects all uses. For example, we may wish to add digits to ALMOST-PI or enlarge an empty scene:

```
(define ALMOST-PI 3.14159)

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

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

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

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

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

Again, we state an imperative slogan:

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

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

2.5 Programs

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

- a *batch program* consumes all of its inputs at once and computes its result. Its main function is the composition of auxiliary functions, which may refer to additional auxiliary functions, and so on. When we launch a batch program, the operating system calls the main function on its inputs and waits for the program’s output.
- an *interactive program* consumes some of its inputs, computes, produces some output, consumes more input, and so on. When an input shows up, we speak of an *event*, and we create interactive programs as *event-driven* programs. The main function of such an event-driven program uses an expression to describe which functions to call for which kinds of events. These functions are called *event handlers*.

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

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

Batch Programs As mentioned, a batch program consumes all of its inputs at once and computes the result from these inputs. Its main function expects some arguments, hands them to auxiliary functions, receives results from those, and composes these results into its own final answer.

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. Indeed, the name “batch program” dates to the early days of computing when a program read a file (or several files) from a batch of punch cards and placed the result in some other file(s), also a batch of cards. Conceptually, a batch program reads the input file(s) at once and also produces 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, save the definitions area in a file.

After the first interaction the file named `"sample.dat"` contains

212

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`.

For 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
'stdout
```

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

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

Let’s 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:

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

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 textbook, a

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?

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 formula into BSL is straightforward:

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

Recall that `5/9` is a number, a rational fraction to be precise, and 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:

```
> (C 32)
0
> (C 212)
100
> (C -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 C with existing primitive functions:

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

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

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

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

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

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

exercises in algebra merely illustrate the idea of function composition.

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

```
> (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"
```

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

For the first interaction, we use `'stdout` so that we can view what `convert` outputs in DrRacket's interactions area. For the second one, `convert` is given the name `"out.dat"`. As expected, the call to `convert` returns this string; from the description of `write-file` we also know that it deposited a Fahrenheit temperature in the file. Here we read the content of this file with `read-file`, but you could also view it with a text editor.

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

Exercise 31. Recall the `letter` program from [Composing Functions](#). Here is how to launch the program and have it write its output to the interactions area:

```
> (write-file
  'stdout
  (letter "Matthew" "Fisler" "Felleisen"))
Dear Matthew,
We have discovered that all people with the
last name Fisler have won our lottery. So,
Matthew, hurry and pick up your prize.
Sincerely,
Felleisen
'stdout
```

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

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

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

The function consumes four strings: the first three are the names of input files and the last one serves as an output file. It uses the first three to read one string each from the three named files, hands these strings to `letter`, and eventually writes the result of this function call into the file named by `out`, the fourth argument to `main`.

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

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

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

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

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

Naturally, different programs have different needs. One program may interpret keystrokes as signals to control a nuclear reactor; another passes them to a word processor. To make a general-purpose computer work on these radically different tasks, different programs install different event handlers. That is, a rocket-launching program uses one kind of function to deal with clock ticks while an oven's software uses a different kind.

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

DrRacket is a small operating system, and BSL is one of its programming languages. The latter comes with the `2htdp/universe` library, which provides `big-bang`, a mechanism for telling the operating system which function deals with which event. In addition, `big-bang` keeps track of the

state of the program. To this end, it comes with one required sub-expression, whose value becomes the *initial state* of the program. Otherwise **big-bang** consists of one required clause and many optional clauses. The required **to-draw** clause tells DrRacket how to render the state of the program, including the initial one. Each of the other, optional clauses tells the operating system that a certain function takes care of a certain event. Taking care of an event in BSL means that the function consumes the state of the program and a description of the event, and that it produces the next state of the program. We therefore speak of the *current state* of the program.

Terminology In a sense, a **big-bang** expression describes how a program connects with a small segment of the world. This world might be a game that the program's users play, an animation that the user watches, or a text editor that the user employs to manipulate some notes.

Programming language researchers therefore often say that **big-bang** is a description of a small world: its initial state, how states are transformed, how states are rendered, and how **big-bang** may determine other attributes of the current state. In this spirit, we also speak of the *state of the world* and even call **big-bang** programs *world programs*. **End**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 100×100 pixels to 1×1 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 then returns a new state:

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

Stop! Explain what happens when you hit “return”, count to 10, and finally press “a”.

What you will see is that the red square shrinks at the rate of one pixel per clock tick. As soon as you press the “a” key, though, the red square reinflates 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’s look at a schematic version:

```
(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`—and a `stop-when` clause.

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 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 clicked, 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, clicking 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 `##true`, `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 keystroke	<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 13: How `big-bang` works

The table in figure 13 concisely summarizes this process. In the first row, it lists names for the current states. The second row enumerates names for the events that DrRacket encounters: e_0 , e_1 , and so on. 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_0 is a clock tick, `big-bang` evaluates `(tock cw0)` to produce cw_1 .
- If e_0 is a key event, `(ke-h cw0 e0)` is evaluated and yields cw_1 . The handler must be applied to the event itself because, in general, programs are going to react to each key differently.

- If e_0 is a mouse event, `big-bang` runs `(me-h cw0 e0 ...)` to get cw_1 . The call is a sketch 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.
- Finally, `render` turns the current state into an image, which is indicated by the last row. DrRacket displays these images in the separate window.

The column below cw_1 shows how cw_2 is generated, depending on what kind of event e_1 takes place.

Let's interpret this table with the specific sequence of events: the user presses the “a” key, then the clock ticks, and finally the user clicks the mouse to trigger a “button down” event at position (90,100). Then, in Racket notation,

1. cw_1 is the result of `(ke-h cw0 "a")`;
2. cw_2 is the result of `(tock cw1)`; 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 90 100 "button-down"))
```

Stop! How does `big-bang` display each of these three states?

Now let's consider a sequence of three clock ticks. In that case,

1. cw_1 is the result of `(tock cw0)`;
2. cw_2 is the result of `(tock cw1)`; and
3. cw_3 is the result of `(tock cw2)`.

Or, reformulated in BSL:

```
(define cw1 (tock cw0))
(define cw2 (tock cw1))
(define cw3 (tock cw2))
```

Indeed, we can also determine cw_3 via a single expression:

```
(tock (tock (tock cw0)))
```

This determines the state that `big-bang` computes after three clock ticks. Stop! Reformulate the first sequence of events as an expression.

```
(define BACKGROUND (empty-scene 100 100))
(define DOT (circle 3 "solid" "red"))

(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 DOT 50 y BACKGROUND))

(define (stop y ke)
  0)

```

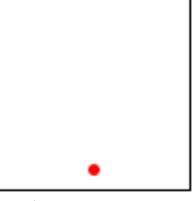
Figure 14: A first interactive program

In short, the sequence of events determines in which order **big-bang** conceptually traverses the above table of possible states to arrive at the current state for each time slot. Of course, **big-bang** 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. See [figure 14](#). The program consists of two constant definitions followed by three function definitions: **main**, which launches a **big-bang** interactive program; **place-dot-at**, which translates the current state into an image; and **stop**, which 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:

```

> (place-dot-at 89)

> (stop 89 "q")
0

```

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

One way to find out whether your conjecture is correct is to launch the **main** function on some reasonable number:

```
> (main 90)
```

Relax.

By now, you may feel that these first two chapters are overwhelming. They introduce many new concepts, including a new language, its vocabulary, its meaning, its idioms, a tool for writing down texts in this vocabulary, and a way of running these programs. Confronted with this plethora of ideas, you may wonder how one creates a program when presented with a problem statement. To answer this central question, the next chapter takes a step back and explicitly addresses the systematic design of programs. So take a breather and continue when ready.

The first few chapters of this book show that learning to program requires some mastery of many concepts. On the one hand, programming needs a 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. Both come with vocabulary, grammar, and an understanding of what “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 reread 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 33. Research the “year 2000” problem.

Here 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 that consumes some information and produces 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 represent 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 object 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 comingle 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 two preinstalled teachpacks to demonstrate the separation of data and information: `2htdp/batch-io` and `2htdp/universe`. Starting with this chapter, we develop design recipes for **batch** and **interactive** programs to give you an idea of how complete programs are designed. Do keep in mind that the libraries of full-fledged programming languages offer many more contexts for complete programs, and that you will need to adapt the design recipes appropriately.

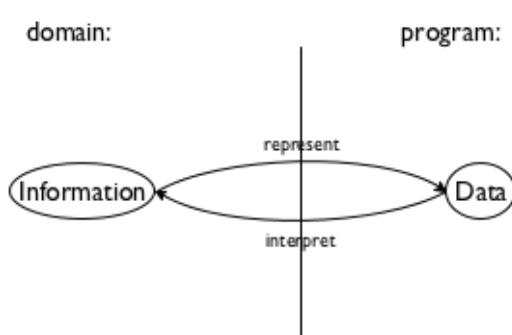


Figure 15: From information to data, and back

Given the central role of information and data, program design must 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 15](#) explains this idea with an abstract diagram.

To make this idea concrete, let's work through some examples. Suppose you are designing a program that consumes and produces information in the form of numbers. While choosing a representation is easy, an interpretation requires explaining what a number such as `42` denotes 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*. A data definition serves two purposes. First, it names a collection of data—a *class*—using a meaningful word. Second, it informs readers how to create elements of this class and how to decide whether some arbitrary piece of data belongs to the collection.

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

Here is a data definition for one of the above examples:

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

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 -274° C, you may wonder whether it is possible to express this knowledge in a data definition. Since our data definitions 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`.”

So far, you have encountered the names of four classes of data: `Number`, `String`, `Image`, and `Boolean`. With that, formulating a new data definition means nothing more than introducing a new name for an existing form of data, say, “temperature” for numbers. Even this limited knowledge, though, suffices to explain the outline of our design process.

The Design 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:

; We use numbers to represent centimeters.

At this point, you may wish to reread the section on [Systematic Program Design](#) in the Preface, especially figure 1.

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

A *function signature* is a comment that tells the readers of your design how many inputs your function consumes, from which classes they are drawn, and what kind of data it produces. Here are three examples for functions that respectively

- o consume one [String](#) and produce a [Number](#):

; String -> Number

- o consume a [Temperature](#) and produce 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 that you stay away from aliasing data definitions for now. A proliferation of such names can cause quite a lot of confusion. It takes practice to balance the need for new names and the readability of programs, and there are more important ideas to understand right now.

- o consume a [Number](#), a [String](#), and an [Image](#):

; Number String Image -> Image

Stop! What does this function produce?

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 variable name for each class of input in the signature; the body of the function can be any piece of data from the output class. These three function headers match the above three signatures:

- o ([define](#) (f a-string) 0)

- o `(define (g n) "a")`
- o `(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 the top and 10 from 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. Assuming you have done the first process step according to the recipe, you add the examples between the purpose statement and the header and get this:

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

4. 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 Stephen Bloch.

For now, the template contains just the parameters, so that the preceding example looks like this:

```
(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. As soon as we introduce new forms of data, templates become interesting.

5. 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 with side len
; given: 2, expect: 4
; given: 7, expect: 49
(define (area-of-square len)
  (sqr len))
```

```
; 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 ...)
;   where ... is (empty-scene 100 100)
(define (add-image y s img)
  (place-image (text s 10 "red") 10 y img))
```

Figure 16: The completion of design step 5

To complete the `add-image` function takes a bit more work than that: see [figure 16](#). In particular, the function needs to turn the given string `s` into an image, which is then placed into the given scene.

6. The last step of a proper design is to test the function on the examples that you worked out before. For now, testing works like this. 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 reassure 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 34. Design the function `string-first`, which extracts the first character from a non-empty string. Don’t worry about empty strings.

Exercise 35. Design the function `string-last`, which extracts the last character from a non-empty string.

Exercise 36. Design the function `image-area`, which counts the number of pixels in a given image.

Exercise 37. Design the function `string-rest`, which produces a string like the given one with the first character removed.

Exercise 38. 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, and so on. 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. Mathematics is at the intersection of many, but not all, domains. Hence, programmers must often pick up new languages as they work through problems with domain experts.
2. 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 your task involves graphics, 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 every function systematically, though global constants as well as auxiliary functions change the design process a bit.

When you have defined global constants, your functions may use them 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

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

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.

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:

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

Testing the function's examples calls for 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 have 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

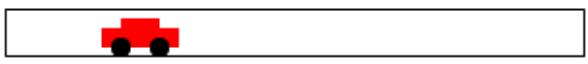
(check-expect (f2c -40) -40)
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)

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

Figure 17: 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. Figure 17 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, the next click retests 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 tests such as the following:

```
(check-expect (render 50)
  )
(check-expect (render 200)
  )
```

Alternatively, you could write them like this:

For additional ways of formulating tests, see intermezzo 1.

```
(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. One way to develop such expressions is to experiment in the interactions area.

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.

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 also helps you create world programs systematically. It starts with a brief summary of the *2htdp/universe* library based on data definitions and function signatures. Then it spells out the design recipe for world programs.

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, keystrokes, 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*.

Figure 18 spells out this idea in a schematic and simplified way.

```
; WorldState: data that represents the state of the world (cw)
; WorldState -> Image
; when needed, big-bang obtains the image of the current
; state of the world by evaluating (render cw)
(define (render ws) ...)

; WorldState -> WorldState
; for each tick of the clock, big-bang obtains the next
; state of the world from (clock-tick-handler cw)
(define (clock-tick-handler cw) ...)

; WorldState String -> WorldState
; for each keystroke, big-bang obtains the next state
; from (keystroke-handler cw ke); ke represents the key
(define (keystroke-handler cw ke) ...)

; WorldState Number Number String -> WorldState
; for each mouse gesture, big-bang obtains the next state
; from (mouse-event-handler cw x y me) where x and y are
; the coordinates of the event and me is its description
```

```
(define (mouse-event-handler cw x y me) ...)

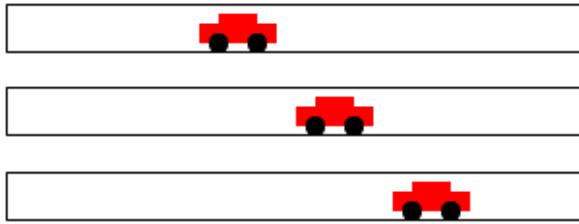
; WorldState -> Boolean
; after each event, big-bang evaluates (end? cw)
(define (end? cw) ...)
```

Figure 18: The wish list for designing world programs

Assuming that you have a rudimentary understanding of the 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 from left to right on the world canvas, three pixels per clock tick.

For this problem statement, it is easy to imagine scenes for 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:

```
(define WHEEL
  (circle WHEEL-RADIUS "solid" "black"))
```

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

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

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

It is good practice to annotate constant definitions with a comment that explains what they mean.

2. Those properties that change over time—in reaction to clock ticks, keystrokes, 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 from 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:

```
; A WorldState is a Number.
; interpretation the number of pixels between
; the left border of the scene 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
; keystroke-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 18](#). Adapt these generic purpose statements to the particular problems you solve so that readers know what they 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's 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 or testing. Its sole reason for existing is that you can **launch** your world program conveniently from DrRacket's interactions area.

The one decision you must make concerns `main`'s arguments. For our sample problem, we opt for one argument: the initial state of the world. Here we go:

```
; 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 start at 13 pixels from 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`, `end?`, or `render`. 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's 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 39. 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 the size of the car image. This kind of program organization is dubbed *single point of control*, and good design employs this idea as much as possible.

Develop your favorite image of an automobile so that WHEEL-RADIUS remains the single point of control.

The next entry on the wish list is the clock tick handling function:

```
; WorldState -> WorldState
; moves the car by 3 pixels for every clock tick
(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 also makes it easy to create examples and to define the function:

```
; WorldState -> WorldState
; moves the car by 3 pixels for every clock tick
; examples:
;   given: 20, expect 23
;   given: 78, expect 81
(define (tock ws)
  (+ ws 3))
```

The last design step calls for confirmation that the examples work as expected. So we click the RUN button and evaluate these expressions:

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

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

Exercise 40. Formulate the examples as BSL tests, that is, using the `check-expect` form. Introduce a mistake. Re-run the 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 the BACKGROUND scene,
; according to the given world state
(define (render ws)
  BACKGROUND)
```

To make examples for a rendering function, we suggest arranging a table like the upper half of [figure 19](#). It lists the given world states and the desired scenes. For your first few rendering functions, you may wish to draw these images by hand.

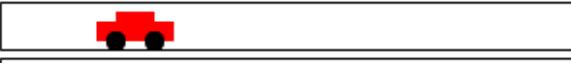
ws	its image
50	
100	
150	
200	
ws	an expression
50	(place-image CAR 50 Y-CAR BACKGROUND)
100	(place-image CAR 100 Y-CAR BACKGROUND)
150	(place-image CAR 150 Y-CAR BACKGROUND)
200	(place-image CAR 200 Y-CAR BACKGROUND)

Figure 19: Examples for a moving car program

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 like those in the lower half of [figure 19](#) that create the images in the table. 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 the BACKGROUND 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 41. Finish the sample problem and get the program to run. That is, assuming that you have solved [exercise 39](#), 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 the 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.

After settling on an initial 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 suit your taste.

Exercise 42. 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 43. Let's work through the same problem statement with a time-based data definition:

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

How do you think this program relates to `animate` 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.)

We end the section with an illustration of mouse event handling, which also illustrates the advantages that a separation of view and model provide.

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

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 click.**

The bold part is the expansion of the sample problem from above.

When we are confronted with a modified problem, we use the design process to guide us to the necessary changes. If used properly, this process naturally determines what we need to add to our existing program to cope with the expansion of the problem statement. So here we go:

1. There are no new properties, meaning we do not need new constants.
2. The program is still concerned with just one property that changes over time, the x-coordinate of the car. Hence, the 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 x-mouse
```

```

; if the given 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 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 x-mouse
; if the given 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 44. 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 x-mouse
; if the given 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 44](#), rerun 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. As with 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 45. 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 reappears on the left. You may wish to read up on the `modulo` function.

Exercise 46. Improve the cat animation with a slightly different image:



```
(define cat2 )
```

Adjust the rendering function from [exercise 45](#) so that it uses one cat image or the other based on whether the x-coordinate is odd. Read up on `odd?` in the HelpDesk, and use a `cond` expression to select cat images.

Exercise 47. 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 45](#). 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, and Itemizations

At the moment, 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 other programming languages) don't suffice. Actual designers need additional ways of representing information as data.

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 of how to write such functions. Neither section uses design, however. Both just present some new construct in your favorite programming language (that's BSL), and offer some examples on how to use it.

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

In this chapter, we discuss a general design for enumerations and intervals, new forms of data descriptions. 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. The last one, itemizations, mixes the first two, specifying ranges in one clause of its definition and specific pieces of data in another. The chapter ends with 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's take a close look at its general shape:

(`cond`

```
[ConditionExpression1 ResultExpression1]  
[ConditionExpression2 ResultExpression2]
```

```
...
```

Brackets make `cond` lines stand out. It is fine to use `(...)` in place of `[...]`.

```
[ConditionExpressionN ResultExpressionN])
```

A `cond` expression starts with `(cond`, its *keyword*, and ends in `)`. 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 `]`.

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 for 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 award at the end of the game. Here is its header:

```
; A PositiveNumber is a Number greater than/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:

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

The variant on the left uses 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 because

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

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's 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 varying 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. Hence,

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

At this point, DrRacket evaluates one condition at a time. For the **first** one to evaluate to `#true`, it continues 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"
```

Here the first condition holds because `3` is between `0` and `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` and thus the calculation proceeds to the third `cond` line. The `else` tells DrRacket to replace the entire `cond` expression with the answer from this clause.

Exercise 48. Enter the definition of `reward` followed by (`reward 18`) into the definitions area of DrRacket and use the stepper to find out **how** DrRacket evaluates applications of the function.

Exercise 49. 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 `y` as `100` and `210`.

```

(define WIDTH 100)
(define HEIGHT 60)
(define MTSCN (empty-scene WIDTH HEIGHT))


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

(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 20: Recall from [One Program, Many Definitions](#)

Nesting `cond` expressions can eliminate common expressions. Consider the function for launching a rocket, repeated in [figure 20](#). Both branches of the `cond` expression have the same shape except as indicated with `....`:

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

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.

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 is one of the following Strings:  
; - "red"  
; - "green"  
; - "yellow"  
; interpretation the three strings represent the three  
; possible states that a traffic light may assume
```

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.

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

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  
; yields the next state given current state 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 50. 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 51. 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 keys on the keyboard
```

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"
; - "\r"
; - "\b"
```

If you look at your keyboard, you find `<`, `↑`, and similar labels. Our chosen programming language, BSL, uses its own data definition to represent this information. Here is an excerpt:

```
; A KeyEvent is one of:
; - 1String
; - "left"
```

You know where to find the full definition.

```
; - "right"
; - "up"
; - ...
```

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.

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

Figure 21: Conditional functions and special enumerations

When programs rely on data definitions that come with the chosen 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 21 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 reordering is appropriate because only two of the `cond`-lines are relevant, and they can be cleanly separated from other lines. Naturally, this kind of rearrangement 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 something like [figure 22](#). Stop! Study the definitions and replace the dots before you read on.

```
; A WorldState is a Number.
; interpretation number of pixels between the top and the UFO

(define WIDTH 300) ; distances in terms of pixels
(define HEIGHT 100)
(define CLOSE (/ HEIGHT 3))
(define MTSCN (empty-scene WIDTH HEIGHT))
(define UFO (overlay (circle 10 "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
; places UFO at given height into the center of MTSCN
(check-expect (render 11) (place-image UFO ... 11 MTSCN))
(define (render y)
  (place-image UFO ... y MTSCN))
```

Figure 22: UFO, descending

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

Sample Problem Add a status line. It 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."

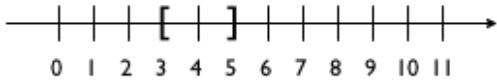
You are free to use appropriate colors for the status line.

In this case, we don't have a finite enumeration of distinct elements or distinct sub-classes 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.

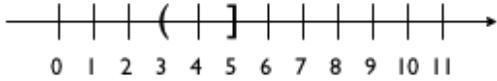
An *interval* is a description of a class of numbers via boundaries. The simplest interval has two boundaries: left and right. If the left boundary is to be included in the interval, we say it is *closed* on the left. Similarly, a right-closed interval includes its right boundary. Finally, if an interval 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 such intervals:

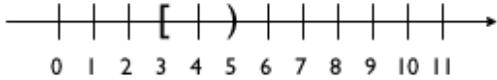
- $[3,5]$ is a closed interval:



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



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



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

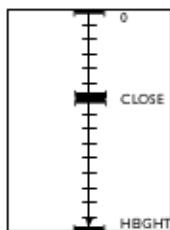


Exercise 52. Which integers are contained in the four intervals above?

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

```
; 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 (—). 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.

On a plain number line, the last interval starts at HEIGHT and goes on forever.

Visualizing the data definition in this manner helps with the design of functions in two 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 tells 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 uses `and` to combine a strictly-less check with a less-than-or-equal check instead of `f`'s `<=` with three arguments.

```
; 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))]))

```

Figure 23: Rendering with a status line

Given all that, we can complete the definition of the function that adds the requested status line to our UFO animation; see [figure 23](#) for the complete definition. The function uses a `cond` expression to distinguish the three intervals. In each `cond` clause, the *ResultExpression* uses `render` (from [figure 22](#)) to create the image with the descending UFO and then places an appropriate text at position (10,10) with `place-image`.

To run this version, you need to change `main` from [figure 22](#) a 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's 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 of the canvas. ...

This could be the response of a client who has watched your animation for a first time.

```

; WorldState -> Image
; adds a status line to the scene created by render

(check-expect (render/status 42)
              (place-image (text "closing in" 11 "orange")
                          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")]))))

```

```

  (text "closing in" 11 "orange")]
  [(> y HEIGHT)
   (text "landed" 11 "red")])
20 20
(render y)))

```

Figure 24: Rendering with a status line, revised

At this point, you have no choice but to change the function `render/status` at **six** 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. 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; see [figure 24](#) and compare with [figure 23](#). 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 sub-classes of numbers, which, in principle, is an infinitely large class. An enumeration spells out item for item the useful elements of an existing class of data. Some data definitions need to include elements from both. They use *itemizations*, which generalize intervals and enumerations. They allow the combination of any already-defined data classes with each other and with individual pieces of data.

Consider the following example, a rewrite of an important data definition from [Enumerations](#):

```

; A KeyEvent is one of:
; - 1String
; - "left"
; - "right"
; - "up"
; - ...

```

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 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:

```

; An NorF is one of:
; - #false
; - a Number

```

This itemization combines one piece of data (`#false`) with a large, and distinct, 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 study 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 representation with two classes of states:

```
; An LR (short for launching rocket) is one of:
; – "resting"
; – NonnegativeNumber
; interpretation "resting" represents a grounded rocket
; a number denotes the height of a rocket in flight
```

While the interpretation of "resting" is obvious, the interpretation of numbers is ambiguous in its notion 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 57](#) below asks you to solve the exercises of this section using the first interpretation of height.

Exercise 53. 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. It's 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 countdowns:

Sample Problem Design a program that launches a rocket when the user presses the space bar. At that point, the simulation starts a countdown 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 program design recipe, we first collect constants:

```
(define HEIGHT 300) ; distances in pixels
(define WIDTH 100)
(define YDELTA 3)

(define BACKG (empty-scene WIDTH HEIGHT))
(define ROCKET (rectangle 5 30 "solid" "red"))

(define CENTER (/ (image-height ROCKET) 2))
```

While `WIDTH` and `HEIGHT` describe the dimensions of the canvas and the background scene, `YDELTA` describes how fast the rocket moves along the y-axis, as specified in the problem statement. The `CENTER` constant is the **computed** center of 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:

```
; An LRCD (for launching rocket countdown) is one of:
; - "resting"
; - a Number between -3 and -1
; - a NonnegativeNumber
; interpretation a grounded rocket, in countdown mode,
; a number denotes the number of pixels between the
; top of the canvas and the rocket (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 countdown 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 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 BACKG))

(check-expect
  (show -2)
  (place-image (text "-2" 20 "red")
              10 (* 3/4 WIDTH)
              (place-image ROCKET 10 HEIGHT BACKG)))

(check-expect
  (show 53)
  (place-image ROCKET 10 53 BACKG))
```

As before in this chapter, we make one test per sub-class in the data definition. The first example shows the resting state, the second the middle of a countdown, and the last one the rocket in flight. Furthermore, we express the expected values as expressions that draw appropriate images. We used DrRacket's interactions area to create these images; what would **you** do?

A close 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 countdown, yet it adds a nice touch. Lastly, if you solved [exercise 53](#) you also know that `0` and `HEIGHT` 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 sub-class of `LRCD` is a (finite) interval and the third one is a half-open interval, let's take a 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 countdown numbers differ.
- The case for `(show HEIGHT)` is different. According to our agreement, the value `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 interactions area, you see that the rocket is halfway underground. This shouldn't be the case, of course, meaning that we need to adjust this test case and the above:

```
(check-expect
  (show HEIGHT)
```

```
(place-image ROCKET 10 (- HEIGHT CENTER) BACKG))

(check-expect
  (show 53)
  (place-image ROCKET 10 (- 53 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 sub-class with a precise condition: (`string?` `x`) picks the first sub-class, which consists of just one element, the string "resting"; (`<= -3 x -1`) completely describes the second sub-class of data; and (`>= x 0`) is a test for all non-negative numbers.

Exercise 54. 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 sub-class 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 CENTER) BACKG)]
    [(<= -3 x -1)
     (place-image (text (number->string x) 20 "red")
                 10 (* 3/4 WIDTH))
     (place-image ROCKET
                 10 (- HEIGHT CENTER)
                 BACKG))]
    [(>= x 0)
     (place-image ROCKET 10 (- x CENTER) BACKG)])))
```

Indeed, this way of defining functions is highly effective and is an essential element of the full-fledged design approach in this book.

Exercise 55. Take another look at `show`. It contains three instances of an expression with the approximate shape:

```
(place-image ROCKET 10 (- ... CENTER) BACKG)
```

This expression appears three 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? You may wish to reread [Prologue: How to Program](#).

Let's 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 sub-class of [LRCD](#), the third and fourth concern the countdown, 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 sub-class of data. Meaning, `launch` should produce `x` 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 countdown 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 countdown. 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 25: Launching a countdown and a liftoff

The design of `fly`—the clock-tick handler—proceeds just like the design of the preceding two functions, and [figure 25](#) displays the result of the design process. Once again the key is to cover the space of possible input data with a goodly bunch of examples, especially for the two intervals. These examples ensure that the countdown and the transition from the countdown to the liftoff work properly.

Exercise 56. 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 liftoff stops gracefully when the rocket is out of sight.

The solution of [exercise 56](#) yields a complete, working program, but one that behaves a bit strangely. Experienced programmers tell you that using negative numbers to represent the countdown phase is too “brittle.” The next chapter introduces the means to provide a good data definition for this problem. Before we go there, however, the next section spells out in detail how to design programs that consume data described by itemizations.

Exercise 57. Recall that the word “height” forced us to choose one of two possible interpretations. 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 reflect 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 (5.00%) markup.

Design a function for a cash register that, given the price of an item, computes the sales tax.

Keep this problem in mind as we revise the steps of the 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 sub-class of data or in some cases just individual pieces of data. Each clause specifies a data representation for a particular sub-class of information. The key is that each sub-class 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:

```
; A Price falls into one of three intervals:  
; - 0 through 1000  
; - 1000 through 10000  
; - 10000 and above.  
; interpretation the price of an item
```

Do you understand how these 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:

Developers in the real world do not use plain numbers in the chosen programming language for representing amounts of money. See intermezzo 4 for some problems with numbers.

```
; Price -> Number  
; computes the amount of tax charged for p  
(define (sales-tax p) 0)
```

- For functional examples, however, it is imperative that you pick at least one example from each sub-class in the data definition. Also, if a sub-class 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's 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! Try to calculate the tax for each of these prices.

Here is our first attempt, with rounded tax amounts:

```
0 537 1000 1282 10000 12017
```

```
0    0    ???    64    ???    961
```

The question marks point out that the problem statement uses the vague phrase “those costing less than \$1,000” and “more than \$10,000” to specify the tax table. While a programmer may jump to the conclusion that these words mean “strictly less” or “strictly more,” the lawmakers may have meant to say “less than or equal to” or “more than or equal to,” 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 at a tax company would have to ask a tax-law specialist.

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’s 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 sub-classes in the data definition.

4. The biggest novelty is the conditional template. In general,

the template mirrors the organization of sub-classes 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 sub-classes in the data definition. If the data definition mentions three distinct sub-classes of input data, you need three `cond` clauses; if it has seventeen sub-classes, 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 sub-classes of data in the data definition:

```
(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 so 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.

```
(define (sales-tax p)
  (cond
    [(and (<= 0 p) (< p 1000)) 0]
    [(and (<= 1000 p) (< p 10000)) (* 0.05 p)]
    [(>= p 10000) (* 0.08 p)]))
```

6. Finally, run the tests and ensure that they cover all `cond` clauses.

What do you do when one of your test cases fails? Review the end of [Designing Functions](#) concerning test failures.

Exercise 58. Introduce constant definitions that separate the intervals for low prices and luxury prices from the others so that the legislators in Tax Land can easily raise the taxes even more.

4.7 Finite State Worlds

With the design knowledge in this chapter, you can develop a complete simulation of American traffic lights. When such a light is green and it is time to stop the traffic, the light turns yellow, and, after that, it turns red. When the light is red and it is time to get the traffic going, the light simply switches to green.

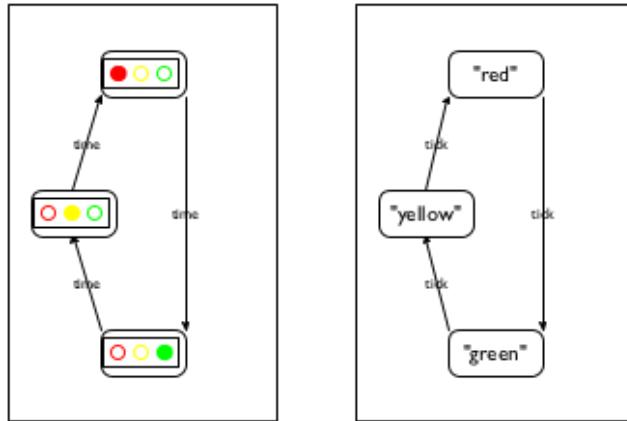


Figure 26: How a traffic light functions

The left-hand side of [Figure 26](#) 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* (FSM), also known as *finite state automata* (FSA). Despite their simplicity, FSMs/FSAs 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, we reuse the string-based data definition of [TrafficLight](#) from above.

The right-hand side of [figure 26](#) is a diagrammatic interpretation of the [TrafficLight](#) data definition. Like the diagram in [figure 26](#), 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 light, we might choose transitions based on keystrokes.

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
; yields the next state, given current state cs
(define (tl-next cs) cs)

; TrafficLight -> Image
; renders the current state cs as an image
(define (tl-render current-state)
  (empty-scene 90 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 59. Finish the design of a world program that simulates the traffic light FSA. Here is the main function:

```
; TrafficLight -> TrafficLight
; simulates a clock-based American traffic light
(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 redraw the state of the world with `tl-render` and to handle clock ticks with `tl-next`. Also note that it informs the computer that the clock should tick once per second.

Complete the design of `tl-render` and `tl-next`. Start with copying `TrafficLight`, `tl-next`, and `tl-render` into DrRacket's definitions area.

Here are some test cases for the design of the latter:

```
(check-expect (tl-render "red") 
)
(check-expect (tl-render "yellow") 
)
```

Your function may use 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 can place bulbs into a background scene.

Exercise 60. An alternative data representation for a traffic light program may use numbers instead of strings:

```
; An N-TrafficLight is one of:
; - 0 interpretation the traffic light shows red
; - 1 interpretation the traffic light shows green
; - 2 interpretation the traffic light shows yellow
```

It greatly simplifies the definition of tl-next:

```
; N-TrafficLight -> N-TrafficLight
; yields the next state, given current state cs
(define (tl-next-numeric cs) (modulo (+ cs 1) 3))
```

Reformulate tl-next's tests 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 61. As [From Functions to Programs](#) says, programs must define constants and use names instead of actual constants. In this spirit, a data definition for traffic lights must use constants, too:

```
(define RED 0)
(define GREEN 1)
(define YELLOW 2)
```

This form of data definition is what a seasoned designer would use.

```
; An S-TrafficLight is one of:
; – RED
; – GREEN
; – YELLOW
```

If the names are chosen properly, the data definition does not need an interpretation statement.

```
; S-TrafficLight -> S-TrafficLight
; yields the next state, given current state cs

(check-expect (tl-next- ... RED) YELLOW)
(check-expect (tl-next- ... YELLOW) GREEN)

(define (tl-next-numeric cs)      (define (tl-next-symbolic cs)
  (modulo (+ cs 1) 3))           (cond
                                     [(equal? cs RED) GREEN]
                                     [(equal? cs GREEN) YELLOW]
                                     [(equal? cs YELLOW) RED]))
```

Figure 27: A symbolic traffic light

[Figure 27](#) displays two different functions that switch the state of a traffic light in a simulation program. 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 the following

```
(define RED "red")
(define GREEN "green")
(define YELLOW "yellow")
```

Does this help you answer the questions?

Aside The `equal?` function in [figure 27](#) compares two arbitrary values, regardless of what these values are. Equality is a complicated topic in the world of programming. **End**

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 [figure 27](#). 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.

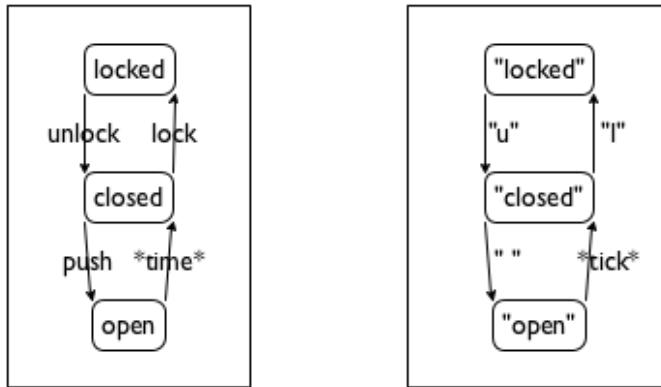


Figure 28: A transition diagram for a door with an automatic closer

Following our recipe, we start with a translation of the three real-world states into BSL data:

```
(define LOCKED "locked") ; A DoorState is one of:
(define CLOSED "closed") ; - LOCKED
(define OPEN "open")      ; - CLOSED
                           ; - OPEN
```

We also keep in mind the lesson of [exercise 61](#), namely, that it is best to define symbolic constants and formulate data definitions in terms of such constants.

The next step of a world design demands that we translate the chosen 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 clock ticks. For the other arrows, we could use either key presses or mouse clicks. Let's 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 of [figure 28](#) expresses these choices graphically; it translates the state-machine diagram from the world of information into the world of data in BSL.

Once we have decided to use the passing of time for one action and key presses for the others, we must design functions that render the current state of the world—represented as **DoorState**—and that transform it into the next state of the world. And that, of course, amounts to a wish list of **big-bang** functions:

- **door-closer**, which closes the door during one tick;
- **door-action**, which acts on it in response to pressing a key; and
- **door-render**, which translates the current state into an image.

Stop! Formulate appropriate signatures.

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 given above:

given state	desired state
LOCKED	LOCKED
CLOSED	CLOSED
OPEN	CLOSED

Stop! Express these examples as BSL tests.

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 your tests.

The second function, door-action, 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
; turns key event k into an action on state s
(define (door-action s k)
  s)
```

We once again present the examples in tabular form:

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-action LOCKED "u") CLOSED)
(check-expect (door-action CLOSED "l") LOCKED)
(check-expect (door-action CLOSED "") OPEN)
(check-expect (door-action OPEN "a") OPEN)
(check-expect (door-action CLOSED "a") CLOSED)

(define (door-action 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.

Lastly, we need to render the state of the world as a scene:

```
; DoorState -> Image
; translates the state s into a large text image
(check-expect (door-render CLOSED)
              (text CLOSED 40 "red"))
(define (door-render s)
  (text s 40 "red"))
```

This simplistic function uses large text. Here is how we run it all:

```
; 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-action]
            [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 62. During a door simulation the “open” state is barely visible. Modify `door-simulation` so that the clock ticks once every three seconds. Rerun the simulation.

5 Adding Structure

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 position 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.

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.

Here is another scenario that raises the same question. Your cell phone is mostly a few million lines of code wrapped in plastic. Among other things, it administers your 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 to combine 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 From Positions to posn Structures

A position 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 positions 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.

Before doing anything else, 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. Then again, the concept of a posn ought to look like the Cartesian points or positions in the plane you may have encountered before.

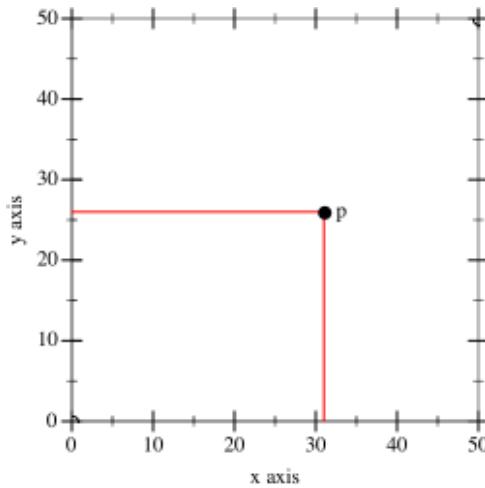


Figure 29: A Cartesian point

Selecting a Cartesian point's pieces is also a familiar process. For example, when a teacher says, "take a look at the graph of figure 29 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 thank Neil Toronto for the plot library.

We can express this idea in BSL. Assume you add

```
(define p (make-posn 31 26))
```

to the definitions area, click *RUN*, and perform these interactions:

```
> (posn-x p)
31
> (posn-y p)
26
```

Defining p is like marking the point in a Cartesian plane; using `posn-x` and `posn-y` is like subscripting p with indexes: 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
```

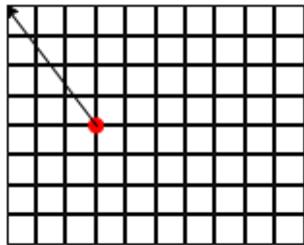
DrRacket uses these equations during computations. Here is an example of a computation involving `posn` structures:

```
(posn-x p)
== ; DrRacket replaces p with (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. Also use DrRacket's stepper to double-check.

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—“as the crow flies”—from the designated point to the top-left corner of the canvas.

Here are the purpose statement and the header:

```
; computes the distance of ap to the origin
(define (distance-to-0 ap)
  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 our 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:

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

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 ap)` and `(posn-y ap)` because `ap` is the name of the given, unknown `posn` structure:

```
(define (distance-to-0 ap)
  (... (posn-x ap) ...)
  ... (posn-y ap) ...))
```

Using this template and the examples, the rest is straightforward:

```
(define (distance-to-0 ap)
  (sqrt
    (+ (sqr (posn-x ap))
        (sqr (posn-y ap)))))
```

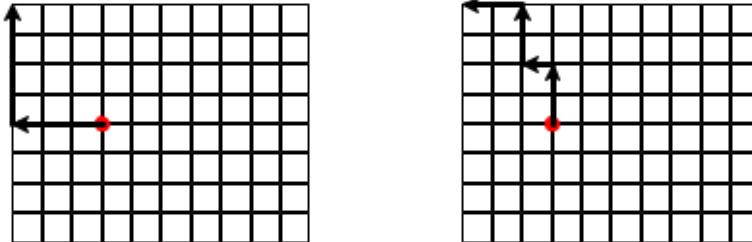
The function squares `(posn-x ap)` and `(posn-y ap)`, which represent the x- and y-coordinates, sums up the results, and takes the square root. With DrRacket, we can also quickly check that our new function produces the proper results for our examples.

Exercise 63. 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 64. 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 left as needed, followed by as many upward steps as needed. In comparison, the right one shows a “random walk” strategy, going some blocks leftward, some upward, 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.

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])
```

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*.

This 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 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*. It takes 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;
- one *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.

Curiously, a structure type definition makes up names for the various new operations it creates. 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 and perhaps even confusing. But, with a little bit of practice, you’ll get the hang of it. It also 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.

Exercise 65. Take a look at 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])`

Write down the names of the functions (constructors, selectors, and predicates) that each introduces.

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 consume one instance of `entry` and select one of the three field values; and
- `entry?`, the predicate.

Since each `entry` combines three values, the expression

```
(make-entry "Al Abe" "666-7771" "lee@x.me")
```

creates an `entry` structure with `"Al Abe"` in the `name` field, `"666-7771"` in the `phone` field, and `"lee@x.me"` in the `email` field.

Exercise 66. Revisit the structure type definitions of [exercise 65](#). 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.

In this context, let's study another programming problem:

Sample Problem Develop a structure type definition for a program that deals with “bouncing balls,” briefly mentioned at the very beginning of this chapter. The ball’s location is a single number, namely the distance of pixels from the top. Its constant *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 program 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 of the code into thinking that the expression is about a y-coordinate.

Exercise 67. Here is another way to represent bouncing balls:

```
(define SPEED 3)
```

```
(define-struct balld [location direction])
(make-balld 10 "up")
```

Interpret this code fragment and 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. Unlike bouncing balls, such objects don't always move along vertical lines. Instead, 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 will be next.

It is physics that tells you to add an object's velocity to its location to obtain its next location. Developers need to learn whom to ask about which domain.

Clearly, posn structures can represent locations. For the velocities, we define the vel structure type:

```
(define-struct vel [deltax deltay])
```

It comes with two fields: deltax and deltay. The word "delta" is commonly used to speak of change when it comes to simulations of physical activities, and the x and y parts indicate which axis is concerned.

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 and 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 68. An alternative to the nested data representation of balls uses four fields to keep track of the four properties:

```
(define-struct ballf [x y deltax deltay])
```

Yet another alternative is to use *complex numbers*. If you know about them, contemplate a data representation that uses them for both location and velocity. For example, in BSL, 4-3i is a complex number and could be used to represent the location or velocity (4,-3).

Programmers 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's 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-entry "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

Structure types generalize 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. This helps programmers read code because it is much easier to remember that a family name is available in a field called `last-name` than in the 7th field.

Most programming languages also support structure-like data that use numeric field names.

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 lockboxes with as many compartments as there are fields. Here is a representation of

```
(define pl (make-entry "Al Abe" "666-7771" "lee@x.me"))
```

as such a diagram:

entry		
name	phone	email
"Al Abe"	"666-7771"	"lee@x.me"

The box's italicized label identifies it as an instance of a specific structure type; each compartment is labeled, too. Here is another instance:

```
(make-entry "Tara Harp" "666-7770" "th@smlu.edu")
```

corresponds to a similar box diagram, though the content differs:

entry		
name	phone	email
"Tara Harp"	"666-7770"	"th@smlu.edu"

Not surprisingly, nested structure instances have a diagram of boxes nested in boxes. Thus, `ball1` from above is equivalent to this diagram:

ball	
location	velocity
posn	vel
x 30	deltax -10
y 40	deltay +5

In this case, the outer box contains two boxes, one per field.

Exercise 69. Draw box representations for the solution of [exercise 65](#).

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` from above yields a string:

```
> (entry-name pl)
"Al Abe"
```

But `entry-name` applied to a `posn` structure signals an error:

```
> (entry-name (make-posn 42 5))
entry-name:expects an entry, given (posn 42 5)
```

If a compartment contains a box, it might be necessary to use two selectors in a row to get to the desired number:

```
> (ball-velocity ball1)
(make-vel -10 5)
```

Applying `ball-velocity` to `ball1` extracts the value of the `velocity` field, which is an instance of `vel`. To get to the velocity along the x axis, we apply a selector to the result of the first selection:

```
> (vel-deltax (ball-velocity ball1))
-10
```

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 interactions also show that structure instances are values. DrRacket prints them exactly as entered, just like for plain values such as numbers:

```
> (make-vel -10 5)
(make-vel -10 5)
> (make-entry "Tara Harp" "666-7770" "th@smlu.edu")
(make-entry "Tara Harp" "666-7770" "th@smlu.edu")
> (make-centry
  "Shriram Fisler"
  (make-phone 207 "363-2421")
  (make-phone 101 "776-1099")
  (make-phone 208 "112-9981"))
(make-centry ...)
```

Stop! Try this last interaction at home, so you see the proper result.

Generally speaking, a structure type definition not only creates new functions and new ways to create values, but it also adds new laws of computation to DrRacket's knowledge. These laws generalize those for `posn` structures in [Computing with posns](#), and they are best understood by example.

When DrRacket encounters a structure type definition with two fields,

```
(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 analogous laws, Thus,

```
(define-struct vel [deltax deltay])
```

DrRacket adds these two laws to its knowledge:

```
(vel-deltax (make-vel dx0 dy0)) == dx0  
(vel-deltay (make-vel dx0 dy0)) == dy0
```

Using these 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
```

Exercise 70. Spell out the laws for these structure type definitions:

```
(define-struct centry [name home office cell])  
(define-struct phone [area number])
```

Use DrRacket's stepper to confirm 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")))) ""
```

Predicates are the last idea that we must discuss to 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 convey 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 ap (make-posn 7 0))  
  
(define pl (make-entry "Al Abe" "666-7771" "lee@x.me"))
```

If `posn?` is a predicate that distinguishes posns from all other values, we should expect that it yields `#false` for numbers and `#true` for ap:

```
> (posn? ap)
#true
> (posn? 42)
#false
> (posn? #true)
#false
> (posn? (make-posn 3 4))
#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 1: Beginning Student Language](#) explains this law in detail, and it also collects the laws of computing for BSL in one place.

Exercise 71. Place the following into DrRacket's definitions area:

```
; distances in terms of pixels:
(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 that 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 a point x pixels from left, y from top
```

It doesn't make any sense to use other kinds of data to create a posn. Similarly, all fields 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, phone#, and email
```

For both posn and entry, a reader can easily 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 distance to top and velocity
; interpretation 2 distance to left and 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 a 2-dimensional position and 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 [per tick] 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 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 definitions refers to another.

Exercise 72. 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 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 100×100 canvas and allows players to use the mouse to reset the dot. Here is how far you got together:

```
(define MTS (empty-scene 100 100))
(define DOT (circle 3 "solid" "red"))

; A Posn represents the state of the world.

; 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 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 [Designing Functions](#) mentions, it uses the function's parameter to express what the function computes.

Now we work out a couple of examples and formulate them as tests:

```
(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. Let's resume our sample problem from above because it includes just such a task:

Sample Problem A 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 easy now. Since the desired result is a `Posn`, the function uses `make-posn` to combine the pieces:

```
(define (x+ p)
  (make-posn (+ (posn-x p) 3) (posn-y p)))
```

Exercise 73. 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 often called *updaters* or *functional setters*. They are extremely useful when you write large programs.

A function may also produce instances from atomic data. While `make-posn` is a built-in primitive that does so, our running problem provides another fitting illustration:

Sample Problem Another colleague is tasked to design `reset-dot`, a function that resets the dot when the mouse is clicked.

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`-coordinates of the mouse click, and a `MouseEvt`.

By adding the knowledge from the sample problem to the program design recipe, we get a signature, a purpose statement, and a header:

```
; Posn Number Number MouseEvt -> Posn
; for mouse clicks, (make-posn x y); otherwise p
(define (reset-dot p x y me) p)
```

Examples for mouse-event handlers need a `Posn`, two numbers, and a `MouseEvt`, which is just a special kind of `String`. A mouse click, for example, is represented with one of two strings: "button-down" and "button-up". The first one signals that a user clicked the mouse button, the latter signals its release. With this in mind, here are two examples, which you may wish to study and interpret:

```
(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))
```

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` creates instances of `Posn`, but you know this and we don't need to remind you constantly.

Exercise 74. Copy all relevant constant and function definitions to DrRacket's definitions area. Add the tests and make sure they pass. Then run the program and use the mouse to place the red dot.

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:

Remember, it's about physics.

```
(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
```

It is your task to develop `ufo-move-1`. The function computes the location of a given `UFO` after one clock tick passes.

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))
```

The order of these definitions matters. See
[Intermezzo 1: Beginning Student Language](#).

```
(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 always, 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 selector expressions raise 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 of nesting.

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`—because that's what our physics knowledge tells us. Specifically, it says to “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.

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))
```

In geometry, the operation corresponding to `posn+` is called a *translation*.

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 ensure that 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. This yields two expressions, one per new coordinate. With `make-posn` we can combine them into a single `Posn` again.

Exercise 75. Enter these definitions and their test cases into the definitions area of DrRacket and make sure they work. This is the first time that you have dealt with 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 programs manipulate. This universe of data is a collection that not only contains all built-in data but also any piece of data that any program may ever create.

Remember that mathematicians call data collections or data classes sets.

The left side of [figure 30](#) shows one way to imagine the universe of BSL. Since there are 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 30: The universe of data

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 on the right side in [figure 30](#) depicts the following data definition:

```
; A BS is one of:  
; - "hello",  
; - "world", or  
; - pi.
```

While this particular data definition looks 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 definition of structure types completely revised the picture. When a programmer defines a structure type, the universe expands with all possible structure instances. For example, the addition of posn means that instances of posn with all possible values in the two fields appear. The middle bubble in [figure 31](#) depicts the addition of these values, including such seeming nonsense as `(make-posn "hello" 0)` and `(make-posn (make-posn 0 1) 2)`. And yes, some of these instances of posn make no sense to us. But, a BSL program may construct any of them.

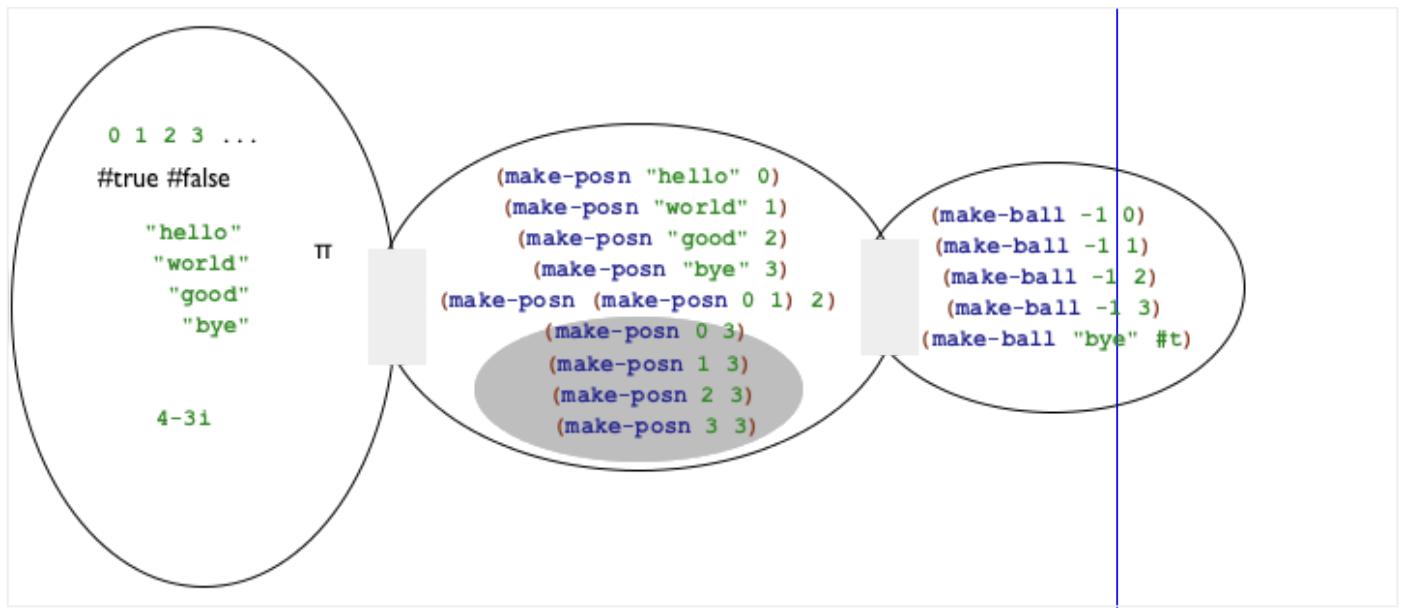


Figure 31: Adding structure to a universe

Adding yet another structure type definition mixes and matches everything again. Say we add the definition for ball, also with two fields. As the third bubble in figure 31 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.

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

```
; 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 in gray in the center bubble of the universe in figure 31, 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 76. 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 77. 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 78. Provide a structure type and a data definition for representing three-letter words. A word consists of lowercase letters, represented with the 1Strings "a" through "z" plus #false.

Note This exercise is a part of the design of a hangman game; see [exercise 396](#).

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. **End**

- 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 79. 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. **End**

- ; H is a Number between 0 and 100.
 ; interpretation represents a happiness value

- ```
(define-struct person [fname lname male?])
; A Person is a structure:
; (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])
; A Dog is a structure:
; (make-dog Person String PositiveInteger H)
```

Add an interpretation to this data definition, too.

- ```
; A Weapon is one of:
; - #false
; - Posn
; interpretation #false means the missile hasn't
; been fired yet; a Posn means it is in flight
```

The last definition is an unusual itemization, combining built-in data with a structure type. The next chapter deals with such definitions in depth.

## 5.8 Designing with Structures

The introduction of structure types reinforces the need for all six steps in the design recipe. 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.

This section adds a design recipe, illustrating it with the following:

**Sample Problem** Design a function that computes the distance of objects in a 3-dimensional space to the origin.

Here we go:

- 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 correspond 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])
; An R3 is a structure:
; (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 they remain the same. Stop! Do it for 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, add a selector for each field to the templates for such functions.

Here is what we have for the sample problem:

```
; 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) ...))
```

You may want to write down next to each selector expression what kind of data it extracts from the given structure; you can find this information in the data definition. Stop! Just do it!

5. Use the selector expressions from the template when you define the function. Keep in mind that you may not need some of them.
6. Test. Test as soon as the function header is written. Test until all expressions have been covered. 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.

There you will find a formula such as  $\sqrt{x^2 + y^2 + z^2}$ .

**Exercise 80.** Create templates for functions that consume instances of the following structure types:

- (`(define-struct movie [title director year])`)
- (`(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 81.** Design the function `time->seconds`, which consumes instances of time structures (see [exercise 77](#)) 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 82.** Design the function `compare-word`. The function consumes two three-letter words (see [exercise 78](#)). 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 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 that consists of a UFO and a tank. The UFO descends along a straight vertical line and a 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 pieces of information. Creating a data representation for that kind of information naturally leads to nested structures.

Let’s add a modicum of spice to our imaginary space invader game. 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 nontrivial 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:

```
; A SpaceGame is a structure:
; (make-space-game Posn Number).
; interpretation (make-space-game (make-posn ux uy) tx)
; describes a configuration where the UFO is
; at (ux,uy) and the tank's x-coordinate is tx
```

Understanding what kind of data representations are needed for world programs takes practice. The following two sections introduce several reasonably complex problem statements. Solve them before moving on to the kind of games that you might like to design on your own.

---

## 5.10 A Graphical Editor

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 editors or even graphical editors.

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:



Someone might have entered the text “helloworld” and hit the left arrow key five times, causing the cursor to move from the end of the text to the position between “o” and “w.” Pressing the space bar would now cause the editor to change its display as follows:



In short, the action inserts “ ” and places the cursor between it and “w.”

Given this much, an editor must track two pieces of information:

1. the text entered so far, and
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])
; An Editor is a structure:
; (make-editor String String)
; interpretation (make-editor s t) describes an editor
; whose visible text is (string-append s t) with
; the cursor displayed between s and t
```

Solve the next few exercises based on this data representation.

**Exercise 83.** 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  $200 \times 20$  pixels. For the cursor, use a  $1 \times 20$  red rectangle and for the strings, black text of size 16.

Develop the image for a sample string in DrRacket’s interactions 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 84.** 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 goodly 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 85.** Define the function `run`. Given the `pre` field of an editor, it launches an interactive editor, using `render` and `edit` from the preceding two exercises for the `to-draw` and `on-key` clauses, respectively.

**Exercise 86.** 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 84](#) 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 87.** Develop a data representation for an editor based on our first idea, using a string and an index. Then solve the preceding exercises again. **Retrace the design recipe.** **Hint** if you haven't done so, solve the exercises in [Functions](#).

**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.

---

## 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 in this section is to create another virtual, happy pet.

**Exercise 88.** 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.

**Exercise 89.** Design the happy-cat world program, which manages a walking cat and its happiness level. 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 90.** Modify the happy-cat program from the preceding exercises so that it stops whenever the cat's happiness falls to 0.

**Exercise 91.** Extend your structure type definition and data definition from [exercise 88](#) 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)
```

The above 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:

```
(define background
 (rectangle (image-width cham)
 (image-height cham)
 "solid"
 "red"))

(overlay cham background)
```

**Exercise 92.** 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 93.** Copy your solution to [exercise 92](#) and modify the copy so that the chameleon walks across a tricolor background. Our solution uses these colors:

```
(define BACKGROUND
 (beside (empty-scene WIDTH HEIGHT "green")
```

Have some Italian pizza when you’re done.

```
(empty-scene WIDTH HEIGHT "white")
(empty-scene WIDTH HEIGHT "red")))
```

but you may use any colors. 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 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

The preceding two chapters introduce two ways of formulating 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 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 to 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 the time after the missile has been fired.

No worries, the next part of the book 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's 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 94.** 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 and graphical constants that describe the dimensions of the world (canvas) and its objects. Also develop 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 represents the state of the world; another is to describe all possible pieces of data that the event-handling functions of the world program may consume. Since we haven’t seen itemizations that include structures, this first subsection introduces this idea. While this 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 94](#), we formulate the data definitions like this:

```
; A UFO is a Posn.
; interpretation (make-posn x y) is the UFO's location
; (using the top-down, left-to-right convention)

(define-struct tank [loc vel])
; A Tank is a structure:
; (make-tank Number Number).
; interpretation (make-tank x dx) specifies the position:
```

```

; (x, HEIGHT) and the tank's speed: dx pixels/tick

; A Missile is a Posn.
; interpretation (make-posn x y) is the missile's place

```

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 complete state of a
; 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 the collection of 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 about to collide with the UFO:

```
(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 95.** Explain why the three instances are generated according to the first or second clause of the data definition.

**Exercise 96.** Sketch how each of the three game states could be rendered assuming a  $200 \times 200$  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 may refer to other data definitions. Hence, 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 refer to this auxiliary definition.

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.
4. The development of the template now exploits two different dimensions: the itemization itself and the use of structures in 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 sub-class 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 sub-class of data described in the item.

When you choose to describe the data with a separate data definition, however, you do **not** add selector expressions. Instead, you create a template for the separate data definition to the task at hand and refer to that template with a function call. The latter indicates that this sub-class 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. The more complex you make your data definitions, the more complex this step becomes. The good news is that this design recipe can help in many situations.

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 (plus some constants) to compute the desired output. Keep in mind that you might need 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.

6. 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](#), reread the description of the simple design recipe, and compare it to this revision.

Let’s illustrate the design recipe with the design of a rendering function for the sample problem at the beginning of this section. 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 keystroke.

The signature of this rendering function says that it maps an element of the state-of-the-world class to the class of [Images](#):

```
; SIGS -> Image
; adds TANK, UFO, and possibly MISSILE to
; the BACKGROUND scene
(define (si-render s) BACKGROUND)
```

Here TANK, UFO, MISSILE, and BACKGROUND are the requested image constants from [exercise 94](#).

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.

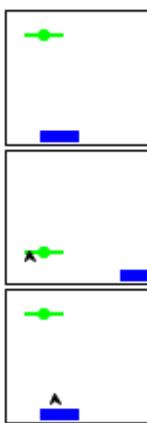
| s                                                                                                                                                                                                                                                                                  | <code>(si-render s)</code>                                                          |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| <code>(make-aim<br/>  (make-posn 10 20)<br/>  (make-tank 28 -3))<br/>(make-fired<br/>  (make-posn 20 100)<br/>  (make-tank 100 3)<br/>  (make-posn 22 103))<br/>(make-fired<br/>  (make-posn 10 20)<br/>  (make-tank 28 -3)<br/>  (make-posn 32 (- HEIGHT TANK-HEIGHT 10)))</code> |  |
|                                                                                                                                                                                                                                                                                    |                                                                                     |

Figure 32: Rendering space invader game states, by example

Since the itemization in the data definition consists of two items, let’s make three examples, using the data examples from above. See [figure 32](#). 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 every `cond` clause that deals with structures. 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) ...))])
```

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 in order 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 32](#), 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:

```
; 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)
```

```
; SIGS -> Image
; renders the given game state on top of BACKGROUND
; for examples see figure 32
(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))))]

```

Figure 33: The complete rendering function

With a bit of analogy, we can deal with the second `cond` clause in the same way. Figure 33 shows the complete definition. Best of all, we can immediately reuse 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 97.** Design the functions `tank-render`, `ufo-render`, and `missile-render`. Compare this expression:

```

(tank-render
 (fired-tank s)
 (ufo-render (fired-ufo s)
 (missile-render (fired-missile s)
 BACKGROUND)))

```

with this one:

```

(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 98.** 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 100](#).

**Exercise 99.** Design `si-move`. This function is called for every clock tick to determine to which position the objects move now. Accordingly, it consumes an element of `SIGS` and produces another one.

Moving the tank and the missile (if any) is relatively straightforward. They move in straight lines at a constant speed. Moving the UFO calls for small random jumps to the left or the right. Since you have never dealt with functions that create random numbers, the rest of this exercise is a longish hint on how to deal with this issue.

BSL comes with a function that creates random numbers. Introducing this function illustrates why the signatures and purpose statements play such an important role during the design. Here is the

relevant material for the function you need:

```
; Number -> Number
; produces a number in the interval [0,n),
; possibly a different one each time it is called
(define (random n) ...)
```

Since the signature and purpose statement precisely describe what a function computes, you can now experiment with `random` in DrRacket's interactions area. Stop! Do so!

If `random` produces different numbers (almost) every time it is called, testing functions that use `random` is difficult. To start with, separate `si-move` and its proper functionality into two parts:

The idea that you must use `random` is BSL knowledge, not a part of the design skills you must acquire, which is why we provide this hint. Also, `random` is the first and only BSL primitive that is not a mathematical function. Functions in programming are inspired by mathematical functions, but they are not identical concepts.

```
(define (si-move w)
 (si-move-proper w (random ...)))

; SIGS Number -> SIGS
; moves the space-invader objects predictably by delta
(define (si-move-proper w delta)
 w)
```

With this definition you separate the creation of a random number from the act of moving the game objects. While `random` may produce different results every time it is called, `si-move-proper` can be tested on specific numeric inputs and is thus guaranteed to return the same result when given the same inputs. In short, most of the code remains testable.

Instead of calling `random` directly, you may wish to design a function that creates a random x-coordinate for the UFO. Consider using `check-random` from BSL's testing framework to test such a function.

**Exercise 100.** 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. It reacts to three different keys:

- 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!

```
; SIGS.v2 -> Image
; renders the given game state on top of BACKGROUND
(define (si-render.v2 s)
 (tank-render
```

```

(sigs-tank s)
(ufo-render (sigz-ufo s)
 (missile-render.v2 (sigz-missile s)
 BACKGROUND)))

```

Figure 34: 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:

```

(define-struct sigs [ufo tank missile])
; A SIGS.v2 (short for SIGS version 2) is a structure:
; (make-sigs UFO Tank MissileOrNot)
; interpretation represents the complete state of a
; space invader game

; A MissileOrNot is one of:
; - #false
; - Posn
; interpretation#false means the missile is in the tank;
; Posn says the missile is at that 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.

As a result, the functions for this second data representation of states differ from the functions for the first one. In particular, functions that consume an element of `SIGS.v2` do not use a `cond` expression because there is only one kind of element in the collection. In terms of design approach, the design recipe for structures from [Designing with Structures](#) suffices. [Figure 34](#) shows the result of designing the rendering function for this data representation.

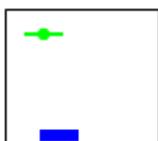
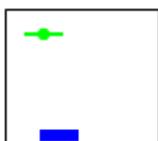
In contrast, the design of functions using `MissileOrNot` requires the recipe from this section. Let's look at the design of `missile-render.v2`, whose job it is to add a missile to an image. Here is the header material:

```

; MissileOrNot Image -> Image
; adds an image of missile m to scene s
(define (missile-render.v2 m s)
 s)

```

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. [Figure 35](#) demonstrates the workings of the function with two distinct scenarios.

|                         |                                                                                     |
|-------------------------|-------------------------------------------------------------------------------------|
| <code>m</code>          | <code>(missile-render.v2 m s)</code>                                                |
| <code>#false</code>     |  |
| <code>(make-posn</code> |  |

32

( - HEIGHT  
TANK-HEIGHT  
10))

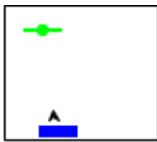


Figure 35: Rendering the space invader games, with tanks

**Exercise 101.** Turn the examples in [figure 35](#) into test cases.

Now we are ready to develop the template. Because 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 s)
 (cond
 [(boolean? m) ...]
 [(posn? m) ...]))
```

Following the data definition again, the first `cond` clause checks whether `m` is a `Boolean` value and the second one checks whether it is an element of `Posn`. And, if someone were to accidentally apply `missile-render.v2` to `#true` and to 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 s)
 (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 s)
 (cond
 [(boolean? m) s]
 [(posn? m)
 (place-image MISSILE (posn-x m) (posn-y m) s)]))
```

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 `s`. 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 `s`, so you have to figure out the best combination of primitive operations and your own functions to combine the four values. The choice of this combining operation is precisely where your creative insight as a programmer comes into play.

**Exercise 102.** Design all other functions that are needed to complete the game for this second data definition.

**Exercise 103.** 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 constrictors**, whose attributes include length and girth; and
- **armadillos**, for which you must determine appropriate attributes, including one that determines the space needed for transport.

Develop a template for functions that consume zoo animals.

Design the `fits?` function, which consumes a zoo animal and a description of a cage. It determines whether the cage's volume is large enough for the animal.

**Exercise 104.** Your home town manages a fleet of vehicles: automobiles, vans, buses, and SUVs. Develop a data representation for vehicles. The representation of each vehicle must describe the number of passengers that it can carry, its license plate number, and its fuel consumption (miles per gallon). Develop a template for functions that consume vehicles.

**Exercise 105.** Some program contains the following data definition:

```
; A Coordinate is one of:
; - a NegativeNumber
; interpretation on the y axis, distance from top
; - a PositiveNumber
; interpretation on the x axis, distance from left
; - a Posn
; interpretation an ordinary Cartesian point
```

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 106.** In [More Virtual Pets](#) we discussed the creation of virtual pets that come with happiness gauges. One of the virtual pets is a cat; the other one, a chameleon. Each program is dedicated to a single pet, however.

Design the `cat-cham` world program. Given both a location and an animal, it walks the latter across the canvas, starting from the given location. Here is the chosen data representation for animals:

```
; A VAnimal is either
; - a VCat
; - a VCham
```

where `VCat` and `VCham` are your data definitions from [exercises 88 and 92](#).

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 animal—as applicable.

It remains impossible to change the color of a cat or to pet a chameleon.

**Exercise 107.** 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 a data representation that specifies a *focus* animal, that is, the animal that 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 108.** In its default state, a pedestrian crossing light shows an orange person standing on a red background. When it is time to allow the 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.

**Exercise 109.** Design a world program that recognizes a pattern in a sequence of `KeyEvents`.

Initially the program shows a 100 by 100 white rectangle. Once your program has encountered the first desired letter, it displays a yellow rectangle of the same size. After encountering the final letter, the color of the rectangle turns green. If any “bad” key event occurs, the program displays a red rectangle.

| conventional                               | defined abbreviations                     |
|--------------------------------------------|-------------------------------------------|
| <code>; ExpectsToSee.v1 is one of:</code>  | <code>; ExpectsToSee.v2 is one of:</code> |
| <code>; - "start, expect an 'a'"</code>    | <code>; - AA</code>                       |
| <code>; - "expect 'b', 'c', or 'd'"</code> | <code>; - BB</code>                       |
| <code>; - "finished"</code>                | <code>; - DD</code>                       |
| <code>; - "error, illegal key"</code>      | <code>; - ER</code>                       |

```
(define AA "start, ...")
(define BB "expect ...")
(define DD "finished")
(define ER "error, ...")
```

Figure 36: Two ways of writing a data definition for FSMs

The specific sequences that your program looks for start with "a", followed by an arbitrarily long mix of "b" and "c", and ended by a "d". Clearly, "acbd" is one example of an acceptable string; two others are "ad" and "abcbabcd". Of course, "da", "aa", or "d" do not match.

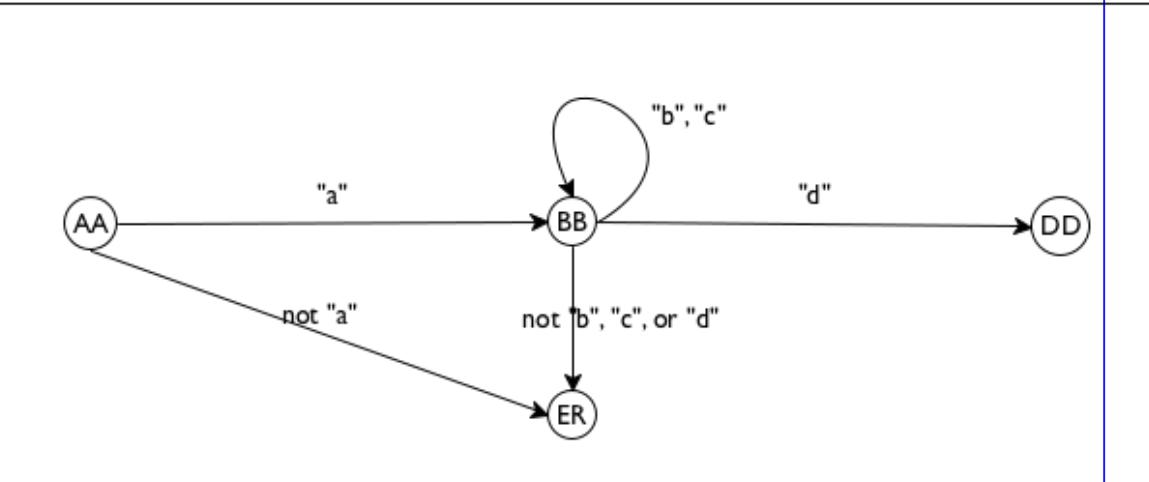


Figure 37: A finite state machine as a diagram

**Hint** Your solution implements a finite state machine (FSM), an idea introduced in [Finite State Worlds](#) as one design principle behind world programs. As the name says, an 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 reconsider 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*.

For a sequence-recognition problem, states typically represent the letters that the machine expects to see next; see [figure 36](#) for a data definition. Take a look at the last state, which says an illegal input has been encountered. [Figure 37](#) 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.

The data definition on the right uses the naming technique introduced in [exercise 61](#).

**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 means a followed by 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 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's demonstrate this point with a simple program, a function for computing the area of a disk:

```
; Number -> Number
; computes the area of a disk with radius r
(define (area-of-disk r)
 (* 3.14 (* r r)))
```

It is a form of self-delusion to expect that we always respect our own function signatures. Calling a function on the wrong kind of data happens to the best of us. While many languages are like BSL and expect programmers to check signatures on their own, others do so automatically at the cost of some additional complexity.

Our friends may wish to use this function for 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 your own choice.

```
; Any BSL value is one of:
; - Number
; - Boolean
; - String
; - Image
; - (make-posn Any Any)
; ...
; - (make-tank Any Any)
; ...
```

Figure 38: The universe of BSL data

Specifically, we can define checked versions of our functions, when we wish to hand them to our friends. Because our friends may not know much BSL, we must expect that they apply this *checked function* to arbitrary BSL values: numbers, 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. Figure 38 displays this shape of this data definition. As discussed in [The Universe of Data](#), the data definition for `Any` is open-ended because every structure type definition adds new instances. These instances may contain `Any` values again, which implies that the data definition of `Any` must refer to itself—a scary thought at first.

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) ...)]
 ...
 ; which selectors are needed in the next clause?
 [(tank? v) ...]
 ...))
```

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 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 MESSAGE "area-of-disk: number expected")

(define (checked-area-of-disk v)
 (cond
 [(number? v) (area-of-disk v)]
 [(boolean? v) (error MESSAGE)]
 [(string? v) (error MESSAGE)]
 [(image? v) (error MESSAGE)]
 [(posn? v) (error MESSAGE)]
 ...
 [(tank? v) (error MESSAGE)]
 ...))
```

The use of `else` helps us finish this 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, let's experiment:

```
> (checked-area-of-disk "my-disk")
area-of-disk:number expected
```

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 proper program 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 110.** 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 111.** 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. 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 assume that there is a data definition labeled `XYZ`, and that `(XYZ? a)` produces `#true` when `a` is an element of `XYZ` and `#false` otherwise.

For `area-of-disk`, which consumes `Numbers`, `number?` is clearly the appropriate predicate. 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 9 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? empty-image) #false)
```

Explain the expected answers!

Since predicates consume all possible BSL values, their templates are just like the templates for checked-f. Stop! Find the template and take a second look before you read on.

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)`, but `(false? v)` would also do:

```
(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 112.** Reformulate the predicate now using an `or` expression.

**Exercise 113.** Design predicates for the following data definitions from the preceding section: `SIGS`, `Coordinate` (exercise 105), and `VAnimal`.

To wrap up, let us mention `key-event?` and `mouse-event?` as two important predicates that you may wish to use in your world programs. They check the expected property, but you should check out their documentation to make sure you understand what they compute.

## 6.4 Checking the World

In a world program, many things can go wrong. Although 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 place that much trust in ourselves. When we design a world program that takes care of clock ticks, mouse clicks, keystrokes, 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. At the same time, `big-bang` accepts this piece of data and holds on to it, until the next event takes place. It is only when the following event handler receives this inappropriate piece of data that the program may fail. But it may get worse because even the second and third and fourth event-handling step may actually cope with inappropriate state values, 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 that 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 114.** Use the predicates from [exercise 113](#) to check the space invader world program, the virtual pet program ([exercise 106](#)), and the editor program ([A Graphical Editor](#)).

## 6.5 Equality Predicates

An *equality predicate* is a function that compares two elements of the same collection of data. Recall the definition of [TrafficLight](#), which is the collection of three strings: "red", "green", and "yellow". Here is one way to define the `light=?` function:

```
; 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 reveal conflicts with our intentions:

```
> (light=? "salad" "greens")
#false
> (light=? "beans" 10)
string=?: expects a string as 2nd argument, given 10
```

Compare these interactions with other, built-in equality predicates:

```
> (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. We call the predicate for [TrafficLight](#) `light?` for brevity:

The case of characters matters; "red" is different from "Red" or "RED".

```
; 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:

```

(define MESSAGE
 "traffic light expected, given some other value")

; 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 MESSAGE)))

```

**Exercise 115.** Revise `light=?` so that the error message specifies which of the two arguments isn't an element of `TrafficLight`.

While it is unlikely that your programs will use `light=?`, they 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 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, and 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. The following intermezzo explains this idea in detail.
5. Programming languages, including BSL, come with a rich set of libraries so that programmers don't have to reinvent the wheel all the time. A programmer should become comfortable with the functions that a library provides, especially their signatures and purpose 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**.

## Intermezzo 1: Beginning Student Language

**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 “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.

In many ways, the analogy of **Fixed-Size Data** is correct. A programming language does have a vocabulary and a grammar, though programmers use the word *syntax* for these elements. A sentence in BSL is an expression or a definition. The grammar of BSL dictates how to form these phrases. But 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*.

This intermezzo presents BSL as if it were an extension of the familiar language of arithmetic and algebra in middle school. After all, computation starts with this form of simple mathematics, and we should understand the connection between this mathematics and computing. The 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.

Programmers must eventually understand these principles of **computation**, but they are complementary to the principles of **design**.

## BSL Vocabulary

**Figure 39** 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: `" "`, `"he says \"hello world\" to you"`, `"doll"`, 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 39: BSL core vocabulary

Each of the explanations defines a set via a suggestive itemization of its elements. Although it is possible to specify these collections in their entirety, we consider this superfluous here and trust your intuition. Just keep in mind that each of these sets may come with some extra elements.

---

## BSL Grammar

Figure 40 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, though, is to discuss how to read such grammars. Each line with a `=` introduces a *syntactic category*; the best way to pronounce `=` is 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-expr* or a sequence of two of them, or three, four, five, or however many. Since this example is not particularly illuminating, let’s look at the second syntactic category. It says that *def* is either

Reading a grammar aloud makes it sound like a data definition. One could indeed use grammars to write down many of our data definitions.

```
(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-expr ...
def-expr = def
| expr
def = (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 40: 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 40 42 is a sentence of BSL:

- The first syntactic category says that a program is a *def-expr*. Expressions may refer to the definitions.
- The second one tells us that a *def-expr* is either a *def* 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 39 that 42 is a value, we have confirmation.

The interesting parts of the grammar show how to form compound sentences, those built from other sentences. For example, the *def* 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. Note how the leading keyword `define` distinguishes definitions from expressions.

Expressions (*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. Like `define`, the keyword `cond` distinguishes conditional expressions from 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, these parenthesized sentences are not legal 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. 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 carefully reading code examples in books, paying attention to how they are formatted.

Keep in mind that two kinds of readers study your BSL programs: people and DrRacket.

**Exercise 116.** Take a look at the following sentences:

1. `x`
2. `(= y z)`
3. `(= (= y z) 0)`

Explain why they are syntactically legal expressions

**Exercise 117.** Consider the following sentences:

1. `(3 + 4)`
2. `number?`
3. `(x)`

Explain why they are syntactically illegal.

**Exercise 118.** Take a look at the following sentences:

1. `(define (f x) x)`
2. `(define (f x) y)`
3. `(define (f x y) 3)`

Explain why they are syntactically legal definitions

**Exercise 119.** Consider the following sentences:

1. `(define (f "x") x)`
2. `(define (f x y z) (x))`

Explain why they are syntactically illegal.

**Exercise 120.** Discriminate the legal from the illegal sentences:

1. `(x)`
2. `(+ 1 (not x))`
3. `(+ 1 2 3)`

Explain why the sentences are legal or illegal. Determine whether the legal ones belong to the category *expr* or *def*.

**Note on Grammatical Terminology** The components of compound sentences have names. We have introduced some of these names on an informal basis. [Figure 41](#) 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 41: Syntactic naming conventions

In addition to the terminology of [figure 41](#), we say *function header* for the second component of a definition. Accordingly, the expression component is called a *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. **End**

---

## 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 39](#) defines grammatically what a value is—the set of values is just a subset of all expressions. The set includes [Booleans](#), [Strings](#), and [Images](#).

The rules of evaluation come in two categories. An infinite number of rules, like those of arithmetic, explain how to determine the value of an application of a primitive operation to values:

```
(+ 1 1) == 2
(- 2 1) == 1
...
```

Remember `==` says that two expressions are equal according to the laws of computation in BSL. 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
```

...

And, like in algebra, you can always replace equals with equals; see [figure 42](#) for a sample calculation.

```
(boolean? (= (string-length (string-append "h" "w"))
 (+ 1 3)))
==

(boolean? (= (string-length (string-append "h" "w")) 4))
==

(boolean? (= (string-length "hw") 4))
==

(boolean? (= 2 4))
==

(boolean? #false)
== #true
```

Figure 42: Replacing equals by equals

Second, we need a rule from algebra to understand the application of a function 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 formulated as generally as possible, so it is best to look at a concrete example. Say the definition is

See [Computing with lambda](#) for more on this rule.

```
(define (poly x y)
 (+ (expt 2 x) y))
```

and DrRacket is given the expression `(poly 3 5)`. Then the first step in an evaluation of the expression uses the beta rule:

```
(poly 3 5) == (+ (expt 2 3) 5) ... == (+ 8 5) == 13
```

In addition to beta, we need rules that determine the value of `cond` expressions. These rules are algebraic even if they are not explicitly taught as part of the standard curriculum. When the first condition is `#false`, the first `cond`-line disappears, leaving the rest of the lines intact:

```
(cond
 [#false ...]
 [condition2 answer2]
 ...)
== (cond
 ; first line removed
 [condition2 answer2]
 ...)
```

This rule has the name `condfalse`. Here is `condtrue`:

```
(cond == answer-1
 [#true answer-1]
 [condition2 answer2]
 ...)
```

The rule also applies when the first 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 the rules of plain arithmetic, the replacement of equals by equals, and both `cond` rules.

**Exercise 121.** Evaluate the following expressions step-by-step:

1. `(+ (* (/ 12 8) 2/3)  
 (- 20 (sqrt 4)))`
2. `(cond  
 [(= 0 0) #false]  
 [(> 0 1) (string=? "a" "a")]  
 [else (= (/ 1 0) 9)])`
3. `(cond  
 [(= 2 0) #false]  
 [(> 2 1) (string=? "a" "a")]  
 [else (= (/ 1 2) 9)])`

Use DrRacket's stepper to confirm your computations.

**Exercise 122.** 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

The stepper tool in DrRacket mimics a student in a pre-algebra course. Unlike you, the stepper is extremely good at applying the laws of arithmetic and algebra as spelled out here, and it is also extremely fast.

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.

A scientist calls the stepper a **model** of DrRacket's evaluation mechanism. [Refining Interpreters](#) presents another model, an **interpreter**.

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

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 40](#) and reasons along the lines explained above. Not all syntactically legal programs have meaning, however.

For a nearly full list of error messages, see the last section of this intermezzo.

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 syntactically legal expression  $(/ 1 0)$ , which, as you know from mathematics, has no 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 remember when they 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:

*Always choose the outermost and left-most nested expression that is ready for evaluation.*

While this guideline may look simplistic, 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")
```

Our current definition of BSL omits `or` and `and` 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. They are `not` 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 because we don't wish to divide by `0` accidentally. Now let's 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` simply does not evaluate the second expression when the first one is `#false`, meaning, `and` *short-circuits* evaluation.

It would be easy to formulate evaluation rules for `and` and `or`. Another way to explain their meaning is to translate them into other expressions:

To make sure `expr-2` evaluates to a Boolean value, these abbreviations should use `(if expr-2 #true #false)` instead of just `expr-2`. We gloss over this detail here.

`(and exp-1 exp-2)` is short for `(cond`  
    `[exp-1 exp-2]`  
    `[else #false])`

and

`(or exp-1 exp-2)` is short for `(cond`  
    `[exp-1 #true]`  
    `[else exp-2])`

So if you are ever in doubt about how to evaluate an `and` or `or` expression, use the above equivalences to calculate. But we trust that you understand these operations intuitively, and that is almost always enough.

**Exercise 123.** 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 of 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. 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.

As it turns out, DrRacket has another way of dealing with function definitions; see [Nameless Functions](#).

Next we must understand what a constant definition means. For a constant definition with a literal constant on the right-hand side, such as

```
(define RADIUS 5)
```

the variable is just a short-hand for the value. Wherever DrRacket encounters `RADIUS` during an evaluation, it replaces it with `5`.

For a 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 of whatever definitions precede this constant definition. Hence,

```
(define RADIUS 5)
(define DIAMETER (* 2 RADIUS))
```

is equivalent to

```
(define RADIUS 5)
(define DIAMETER 10)
```

This process 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 process 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 careful to use functions in constant definitions only when you know they are defined.

**Exercise 124.** 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 the most complex BSL construct. We have therefore saved its explanation for last. Here is the grammar:

```
definition = ...
| (define-struct name [name ...])
```

A structure type definition is a third form of definition. The keyword distinguishes it from both 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 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` adds 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:

```
(c-s-1 (make-c V-1 ... V-n)) == V-1
(c-s-n (make-c V-1 ... V-n)) == V-n
```

For our running example, we get the specific equations

```
(point-x (make-point V U W)) == V
(point-y (make-point V U W)) == U
(point-z (make-point V U W)) == W
```

When DrRacket sees `(point-y (make-point 3 4 5))`, it replaces the expression with `4` while `(point-x (make-point (make-point 1 2 3) 4 5))` evaluates 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:

```
(c? (make-c V-1 ... V-n)) == #true
(c? V) == #false
```

if `V` is a value not constructed with `make-c`. Again, the equations are best understood in terms of our example:

```
(point? (make-point U V W)) == #true
(point? X) == #false
```

if `X` is a value but not a point structure.

**Exercise 125.** Discriminate the legal from the illegal sentences:

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 126.** Identify the *values* among the following 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 127.** 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 the interactions area and with the stepper.

```
def-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 43: BSL, full grammar

---

## BSL Tests

Figure 43 presents all of BSL plus a number of testing forms.

The general meaning of testing expressions is easy to explain. When you click the *RUN* button, DrRacket collects all testing expressions and moves them to the end of the program, retaining the order in which they appear. It then evaluates the content of the definitions area. 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 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 remaining parts of the book re-introduce these test forms as needed.

**Exercise 128.** Copy the following tests into DrRacket's definitions area:

```
(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-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

A BSL program may signal many kinds of syntax errors. While we have developed BSL and its error reporting system specifically for novices who, by definition, make mistakes, error messages need some getting used to.

Below we sample the kinds of error messages that you may encounter. Each entry in one of the listings consists of three parts:

- the code fragments that signal the error message;
- the error message; and
- an explanation with a suggestion on how to fix the mistake.

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 BSL so much that it does not "see" the open parenthesis to the left of `<`. The fix is to use `(< n 0)` as the condition.

The 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.

So, when an error shows up and you need help, find the appropriate figure, search the entries for a match, and then study the complete entry.

### Error Messages about Function Applications in BSL

Assume that the definitions area contains the following and nothing else:

```
; Number Number -> Number
; finds the average of x and y
(define (average x y)
 (/ (+ x y)
 2))
```

Hit the *RUN* button. Now you may encounter the error messages below.

```
(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 must always be followed by a keyword or the name of a function, and `1` is neither. A function name is either defined by BSL, say `+`, or 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.

### Error Messages about Wrong Data in BSL

The error scenarios below again assume that the definitions area contains the following:

```
; Number Number -> Number
; find the average of x and y
(define (average x y) ...)
```

Remember that `posn` is a pre-defined structure type.

---

```
(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.

### Error Messages about Conditionals in BSL

This time we expect a constant definition in the definitions area:

```
; N in [0,1,...10)
(define 0-to-9 (random 10))
```

---

```
(cond
 [(>= 0-to-9 5)])
```

cond: expected a clause with a question and an answer, but found a clause with only one part

Every `cond` clause must consist of 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.

### Error Messages about Function Definitions in BSL

All of the following error scenarios assume that you have placed the code snippet into the definitions area and hit *RUN*.

---

```
(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)
```

define: expected a variable, but found  
a part

The function header contains (x), which is  
**not** a variable name.

---

```
(define (h 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.

### Error Messages about Structure Type Definitions in BSL

Now you need to place the structure type definitions into the definitions area and hit *RUN* to experiment with the following errors.

---

```
(define-struct [x])
(define-struct [x y])
```

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  
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.

---

## 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 with 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.

---

## 8 Lists

You have probably not encountered self-referential definitions before. Your English teachers certainly stay away from these, and many mathematics courses are vague when it comes to such definitions. Programmers cannot afford to be vague. Their work requires precision. While a definition may in general contain several references to itself, this chapter presents useful examples that need just one, starting with the one for lists.

The introduction of lists also spices up the kind of applications we can study. While this chapter carefully builds up your intuition with examples, it also 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.

---

### 8.1 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 an 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.

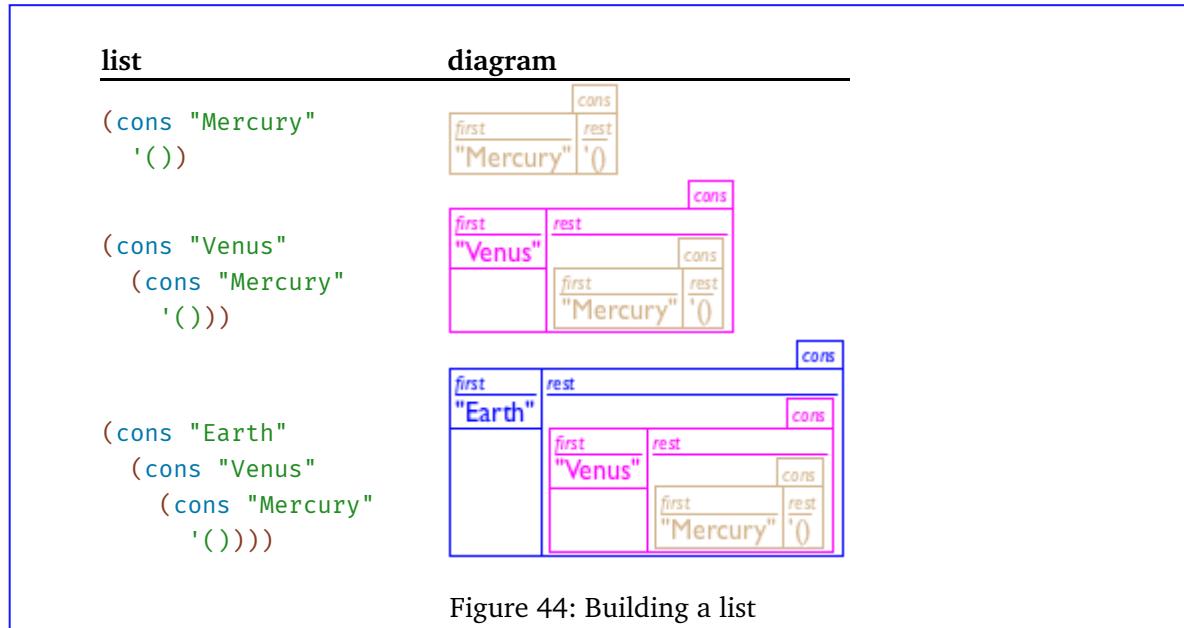
When we form a list, we always start out with the empty list. In BSL, we represent the *empty* list with

```
| '()
```

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 44 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 44 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 44 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, and so on. 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"
 '())))
```

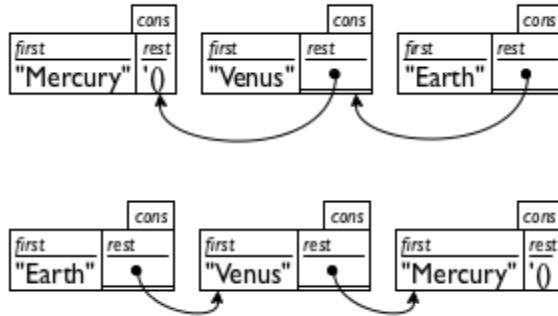


Figure 45: Drawing a list

Because even good artists would have problems with drawing deeply nested structures, computer scientists resort to box-and-arrow diagrams instead. Figure 45 illustrates how to rearrange the last row of figure 44. 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 45, 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 have produced when applied to the list, no matter how long the list is. For this reason, programmers prefer the second arrangement.

**Exercise 129.** 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 some box representations of these lists, similar to those in figures 44 and 45. Which of the sketches do you like better?

You can also make lists of numbers. 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 `cons`s.

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 a list of three numbers:
; (cons Number (cons Number (cons Number '())))
; interpretation a point in 3-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 invitees, by last name
```

Take a deep breath, read it again. This data definition is one of the most unusual definitions you have ever encountered—you have never before seen a definition that refers to itself. 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 '`()`' as one example, using the first clause in the itemization. Given '`()`' as an element of `List-of-names`, it is easy to make a second one:

```
(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 confirmed `List-of-names`.

**Exercise 130.** Create an element of `List-of-names` that contains five `Strings`. Sketch a box representation of the list similar to those found in [figure 44](#).

Explain why

```
(cons "1" (cons "2" '()))
```

is an element of `List-of-names` and why `(cons 2 '())` isn't.

**Exercise 131.** Provide a data definition for representing lists of `Boolean` values. The class contains all arbitrarily long lists of `Booleans`.

---

## 8.2 What Is '`()`', What Is `cons`

Let's 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 in 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 predicates, `empty?` can be applied to any value from the universe of BSL values. It produces `#true` precisely when it is applied to '`()`:

```

> (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 any list-like value. The following definitions translate this idea into BSL:

```

(define-struct pair [left right])
; A ConsPair is a structure:
; (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:
; - '()
; - (make-pair Any ConsOrEmpty)
; interpretation ConsOrEmpty is the class of all lists

; Any Any -> 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: second argument ...")]))

```

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 Definitions](#) demonstrates one way that 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 recognize instances of <code>cons</code>    |

Figure 46: List primitives

Figure 46 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**.

## 8.3 Programming with Lists

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 name is on the list. To make this idea concrete, let's state it as an exercise:

**Sample Problem** You are working on the contact list for some new cell phone. The phone's owner updates and consults this list 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."

Here we use the word "friend" in the sense of social networks, not the real world.

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" is on a-list-of-names
(define (contains-flatt? a-list-of-names)
 #false)
```

While `a-list-of-names` is a good name for the list of names that the function consumes, it is a mouthful and we therefore shorten it to `alon`.

Following the general design recipe, we next make up some examples that illustrate the purpose of the function. First, we determine the output for the simplest input: `'()`. Since this list does not contain any strings, it certainly does not contain "Flatt":

```
(check-expect (contains-flatt? '()) #false)
```

Then we consider lists with a single item. Here are two examples:

```
(check-expect (contains-flatt? (cons "Find" '())))
#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 single item is "Flatt", so the answer is `#true`. Finally, here is a more general example:

```
(check-expect
(contains-flatt?
(cons "A" (cons "Flatt" (cons "C" '()))))
#true)
```

Again, the answer case must be `#true` because the list contains "Flatt". Stop! Make a general example for which the answer must be `#false`.

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 determine which of the two kinds of lists the function received:

```
(define (contains-flatt? alon)
(cond
[(empty? alon) ...]
[(cons? alon) ...]))
```

Instead of `(cons? alon)`, 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 alon)` and `(rest alon)` to the template:

```
(define (contains-flatt? alon)
(cond
```

```
[(empty? alon) ...]
[(cons? alon)
 (... (first alon) ... (rest alon) ...))]
```

Now it is time to switch to the programming task proper, the fifth step of our design recipe. It starts from a template and deals with each `cond` clause separately. If `(empty? alon)` is true, the input is the empty list, in which case the function must produce the result `#false`. In the second case, `(cons? alon)` is true. The annotations in the template remind us that there is a first string and then the rest of the list. So let's consider an example that falls into this category:

```
(cons "A"
 (cons ...
 ... '()))
```

The function, like a person, must compare the first item with `"Flatt"`. In this example, the first one 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? alon)
 (cond
 [(empty? alon) #false]
 [(cons? alon)
 (... (string=? (first alon) "Flatt")
 ... (rest alon) ...))])
```

Furthermore, if the comparison yields `#true`, the function must produce `#true`. If the comparison yields `#false`, we are left with another list of strings: `(rest alon)`. Clearly, the function can't know the final answer in this case, because the answer depends on what “...” represents. 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 `contains-flatt?` and it fits the bill. According to its purpose statement, it determines whether a list contains `"Flatt"`. The statement implies that `(contains-flatt? l)` tells us whether the list of strings `l` contains `"Flatt"`. And, in the same vein, `(contains-flatt? (rest alon))` determines whether `"Flatt"` is a member of `(rest alon)`, which is precisely what we need to know.

In short, the last line should be `(contains-flatt? (rest alon))`:

```
; List-of-names -> Boolean
(define (contains-flatt? alon)
 (cond
 [(empty? alon) #false]
 [(cons? alon)
```

```

(... (string=? (first alon) "Flatt") ...
... (contains-flatt? (rest alon)) ...)))

```

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? alon)` 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 alon
(check-expect
 (contains-flatt? (cons "X" (cons "Y" (cons "Z" '()))))
 #false)
(check-expect
 (contains-flatt? (cons "A" (cons "Flatt" (cons "C" '())))))
 #true)
(define (contains-flatt? alon)
 (cond
 [(empty? alon) #false]
 [(cons? alon)
 (or (string=? (first alon) "Flatt")
 (contains-flatt? (rest alon))))]))

```

Figure 47: Searching a list

Figure 47 then shows the complete definition. 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 the self-reference in the function shouldn't be too surprising.

**Exercise 132.** 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 133.** Here is another way of formulating the second `cond` clause in `contains-flatt?`:

```

... (cond
 [(string=? (first alon) "Flatt") #true]
 [else (contains-flatt? (rest alon))]) ...

```

Explain why this expression produces the same answers as the `or` expression in the version of figure 47. Which version is better? Explain.

**Exercise 134.** Develop the `contains?` function, which determines whether some given string occurs on a given list of strings.

**Note** BSL actually comes with `member?`, a function that consumes two values and checks whether the first occurs in the second, a list:

```
| > (member? "Flatt" (cons "b" (cons "Flatt" '())))
| #true
```

Don't use `member?` to define the `contains?` function.

```
(contains-flatt? (cons "Flatt" (cons "C" '())))
==
(cond
 [(empty? (cons "Flatt" (cons "C" '()))) #false]
 [(cons? (cons "Flatt" (cons "C" '())))
 (or
 (string=? (first (cons "Flatt" (cons "C" '())))) "Flatt")
 (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))])
```

Figure 48: Computing with lists, step 1

## 8.4 Computing with Lists

Since we are still using BSL, the rules of algebra—see intermezzo 1—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.

Figure 48 displays the first step, which uses the usual substitution rule 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.

```
...
 ==
(cond
 [#false #false]
 [(cons? (cons "Flatt" (cons "C" '())))
 (or (string=? (first (cons "Flatt" (cons "C" '())))) "Flatt")
 (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))]
 ==
(cond
 [(cons? (cons "Flatt" (cons "C" '())))
 (or (string=? (first (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" '())))))

```

Figure 49: Computing with lists, step 2

The calculation is continued in figure 49. 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.

```

...
==
(or (string=? "Flatt" "Flatt")
 (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))
== (or #true (contains-flatt? ...))
== #true

```

Figure 50: Computing with lists, step 3

From here, it is just three more steps of arithmetic to get the final result. Figure 50 displays the three steps. The first evaluates `(first (cons "Flatt" ...))` to "Flatt" due to the laws for `first`. The second discovers that "Flatt" is a string and equal to "Flatt". The third says `(or #true X)` is `#true` regardless of what X is.

**Exercise 135.** 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 136.** Validate with DrRacket's stepper

```
(our-first (our-cons "a" '())) == "a"
(our-rest (our-cons "a" '())) == '()
```

See [What Is '\(\)](#), [What Is cons](#) for the definitions of these functions.

```

;; A List-of-strings is one of
;; -- '()
;; -- (cons String List-of-strings)
(define (fun-for-lots-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 51: Arrows for self-references in data definitions and templates

## 9 Designing with Self-Referential Data Definitions

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 is about information of arbitrary size, you need a self-referential data definition to represent it. At this point, you have seen only one such class, [List-of-names](#). The left side of [figure 51](#) shows how to define *List-of-strings* in the same way. Other lists of atomic data work the same way.

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 51](#) for an example of such an annotation.

You must check the validity of self-referential data definitions with the creation of data 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 51](#), you thus get lists like the following three:

```
'() by the first clause
(cons "a" '()) by the second clause, previous example
(cons "b" by the second clause, previous example
 (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 increasingly larger examples, the definition may not live up to its interpretation.

2. 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
; counts how many strings alos 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.

- 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 sub-classes of data?                                                     | Your template needs as many <code>cond</code> clauses as sub-classes that the data definition distinguishes.                                                                |
| How do the sub-classes 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.                                                                    |
| <b>If the data definition refers to some other data definition, where is this cross-reference to another data definition?</b> | Specialize the template for the other data definition. Refer to this template. See <a href="#">Designing with Itemizations, Again</a> , steps 4 and 5 of the design recipe. |

Figure 52: How to translate a data definition into a template

Figure 52 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 51](#) illustrates this idea with the template for functions that consume [List-of-strings](#) shown side by side with the data definition. Both contain one 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-application 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*.

5. For the function body we start with those `cond` lines without recursive function calls, known as *base cases*. The corresponding answers are typically easy to formulate or already given as 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 use of the induction hypothesis for the inductive step of such a proof. Logic proves the validity of this proof technique with an Induction Theorem.

**The rest is then a matter of combining the various values.**

| Question             | Answer                                                                                                     |
|----------------------|------------------------------------------------------------------------------------------------------------|
| What are the answers | The examples should tell you which values you need here. If not, formulate appropriate examples and tests. |

for the non-  
recursive  
`cond`  
clauses?

What do the  
selector  
expressions  
in the  
recursive  
clauses  
compute?

What do the  
natural  
recursions  
compute?

How can the  
function  
combine  
these values  
to get the  
desired  
answer?

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.

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.

Find a function in BSL that combines the values. Or, if that doesn't work, make a wish for a helper function. 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 a *combinator*, slightly abusing existing terminology.

Figure 53: How to turn a template into a function definition

| Question                                                                                 | Answer                                                                                                                                                                                                                                                                                                                       |
|------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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 54: Turning a template into a function, the table method

Figure 53 formulates the first four 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`, the function just needs to add `1` to the value of the latter expression:

```
| (define (how-many alos)
| (cond
| [(empty? alos) 0]
| [else (+ (how-many (rest alos)) 1)])))
```

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 [figure 54](#) 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 55](#) shows what this table looks like for our `how-many` example. The left-most column lists the sample inputs, while the right-most 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 some selector expressions in the template are possibly irrelevant for the actual 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.

- 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                                           | (first<br>alos) | (rest<br>alos)                  | (how-many<br>(rest alos)) | (how-many<br>alos) |
|------------------------------------------------|-----------------|---------------------------------|---------------------------|--------------------|
| (cons "a"<br>'())                              | "a"             | '()                             | 0                         | 1                  |
| (cons "b"<br>'(cons "a"<br>'()))               | "b"             | (cons "a"<br>'())               | 1                         | 2                  |
| (cons "x"<br>(cons "b"<br>(cons "a"<br>'())))) | "x"             | (cons "b"<br>(cons "a"<br>'())) | 2                         | 3                  |

Figure 55: Tabulating arguments, intermediate values, and results

Figure 56 summarizes the design recipe of this section in a tabular format. The first column names the steps of the design recipe, and 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 56 onto one side of an index card and write down your favorite versions of the questions and answers for this design recipe onto the back of it. Then carry it with you for future reference.

| steps            | outcome                                    | activity                                                                                                                                                                                     |
|------------------|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| problem analysis | data definition                            | Develop a data representation for the information; create examples for specific items of information and interpret data as information; identify self-references.                            |
| header           | signature;<br>purpose;<br>dummy definition | Write down a signature using defined names; 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 data definition.                                                                                                               |
| template         | function template                          | Translate the data definition into a template: one <code>cond</code> clause per data clause; selectors where the condition identifies a structure; one natural recursion per self-reference. |
| definition       | full-fledged definition                    | Find a function that combines the values of the expressions in the <code>cond</code> clauses into the expected answer.                                                                       |
| test             | validated tests                            | Turn them into <code>check-expect</code> tests and run them.                                                                                                                                 |

Figure 56: Designing a function for self-referential data

## 9.1 Finger Exercises: Lists

**Exercise 137.** Compare the template for `contains-flatt?` with the one for `how-many`. Ignoring the function name, they are the same. Explain the similarity.

**Exercise 138.** Here is a data definition for representing sequences of amounts of money:

```
; A List-of-amounts is one of:
; - ()
; - (cons PositiveNumber List-of-amounts)
```

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 139.** 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 138](#) 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 140.** Design the function `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`.

Now design `one-true`, a function that consumes a list of Boolean values and determines whether at least one item on the list is `#true`.

Employ the table-based approach to coding. It may help with 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 141.** 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
```

```

; concatenates 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)) ...)]))

```

| <code>l</code>              | <code>(first l)</code> | <code>(rest l)</code> | <code>(cat (rest l))</code> | <code>(cat l)</code> |
|-----------------------------|------------------------|-----------------------|-----------------------------|----------------------|
| <code>(cons "a"</code>      | ???                    | ???                   | ???                         | "ab"                 |
| <code>  (cons "b"</code>    |                        |                       |                             |                      |
| <code>    '())</code>       |                        |                       |                             |                      |
| <code>(cons</code>          | ???                    | ???                   | ???                         | "abcdef"             |
| <code>  "ab"</code>         |                        |                       |                             |                      |
| <code>  (cons "cd"</code>   |                        |                       |                             |                      |
| <code>    (cons "ef"</code> |                        |                       |                             |                      |
| <code>      '())))</code>   |                        |                       |                             |                      |

Figure 57: A table for `cat`

Fill in the table in figure 57. Guess a function that can create the desired result from the values computed by the sub-expressions.

Use DrRacket's stepper to evaluate `(cat (cons "a" '()))`.

**Exercise 142.** Design the `ill-sized?` function, which 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.

## 9.2 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's start with the simple-looking problem of computing the average of a list of temperatures. To simplify, we provide the data definitions:

```

; A List-of-temperatures is one of:
; - '()
; - (cons CTemperature List-of-temperatures)

; A CTemperature is a Number greater than -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 tells 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 mirror the two clauses of the data definition; the questions distinguish empty lists from non-empty lists; and the natural recursion is needed because of the self-reference in the data definition.

It is way too difficult, however, to turn this template into a 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. Similarly, the second clause demands a function that combines a temperature and an average for the remaining temperatures into a new average. Although possible, computing the average in this way is highly unnatural.

When we compute the average of temperatures, we divide their sum by their number. We said so when we formulated our trivial little example. This sentence suggests that `average` is a function of three tasks: summing, counting, and dividing. 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 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 run the tests for `average`.

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 143.** Determine how `average` behaves in DrRacket when applied to the empty list. Then design `checked-average`, a function that produces an informative error message when it is applied to `'()`.

In mathematics, we would say [exercise 143](#) shows that `average` is a *partial function* because it raises an error for `'()`.

An alternative solution is to inform future readers through the signature that `average` doesn't work for empty lists. For that, we need a data representation for lists that excludes `'()`, something like this:

```

; An 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:

```

; An NEList-of-temperatures is one of:
; - (cons CTemperature '())
; - (cons CTemperature NEList-of-temperatures)

```

```
; interpretation non-empty lists of Celsius 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 c '())
```

where `c` stands for a `CTemperature`, like thus: `(cons -273 '())`. Also, it is clear that all non-empty elements of `List-of-temperatures` are also elements of the new class of data: `(cons 1 (cons 2 (cons 3 '())))` fits the bill if `(cons 2 (cons 3 '()))` does, 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's 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:

This alternative development explains that, in this case, we can narrow down the domain of `average` and create a *total function*.

```
; NEList-of-temperatures -> Number
; computes the average temperature

(check-expect (average (cons 1 (cons 2 (cons 3 '()))))
 2)

(define (average ne-l)
 (/ (sum ne-l)
 (how-many ne-l)))
```

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 144.** Will `sum` and `how-many` work for `NEList-of-temperatures` even though they are 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 ne-l) 0)
```

The example is adapted from the example for average; the dummy definition produces a number, but the wrong one for the given test.

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, that is, `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 `cons` instead. Hence the proper condition to distinguish the first kind of data from the second extracts the `rest` field and then uses `empty? :`:

```
; NEList-of-temperatures -> Number
(define (sum ne-l)
 (cond
 [(empty? (rest ne-l)) ...]
 [else ...]))
```

Here `else` replaces `(cons? (rest ne-l))`.

Next you should inspect both clauses and determine whether one or both of them deal with `ne-l` as if it were a structure. This is of course the case, which the unconditional use of `rest` on `ne-l` demonstrates. Put differently, add appropriate selector expressions to the two clauses:

```
(define (sum ne-l)
 (cond
 [(empty? (rest ne-l)) (... (first ne-l) ...)]
 [else (... (first ne-l) ... (rest ne-l) ...)]))
```

Before you read on, explain why the first clause does not contain the selector expression `(rest ne-l)`.

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 ne-l)
 (cond
 [(empty? (rest ne-l)) (... (first ne-l) ...)]
 [else
 (... (first ne-l) ... (sum (rest ne-l)) ...))]))
```

Specifically, `sum` is called on `(rest ne-l)` in the second clause because the data definition is self-referential at the analogous point.

For the fifth design step, let's 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 ne-l)`. Clearly, this one temperature is the sum of all temperatures on the given list:

```
(define (sum ne-l)
 (cond
 [(empty? (rest ne-l)) (first ne-l)]
```

```
[else
 (... (first ne-l) ... (sum (rest ne-l)) ...))]
```

The second clause says that the list consists of a temperature and at least one more; `(first ne-l)` extracts the first position and `(rest ne-l)` the remaining ones. Furthermore, the template suggests to use the result of `(sum (rest ne-l))`. But `sum` is the function that you are defining, and you can't possibly know **how** it uses `(rest ne-l)`. All you do know is what the purpose statement says, namely, that `sum` adds all the temperatures on the given list, which is `(rest ne-l)`. If this statement is true, then `(sum (rest ne-l))` adds up all but one of the numbers of `ne-l`. To get the total, the function just has to add the first temperature:

```
(define (sum ne-l)
 (cond
 [(empty? (rest ne-l)) (first ne-l)]
 [else (+ (first ne-l) (sum (rest ne-l))))]))
```

If you now run the test for this function, you will see that the leap of faith is justified. Indeed, for reasons beyond the scope of this book, this leap is **always** justified, which is why it is an inherent part of the design recipe.

**Exercise 145.** Design the `sorted>?` predicate, which consumes a *NEList-of-temperatures* and 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** This problem is another one where the table-based method for guessing the combinator works well. Here is a partial table for a number of examples in [figure 58](#). Fill in the rest of the table. Then try to create an expression that computes the result from the pieces.

| <code>l</code>                                                       | <code>(first l)</code> | <code>(rest l)</code>                            | <code>(sorted&gt;? (rest l))</code> | <code>(sorted&gt;? l)</code> |
|----------------------------------------------------------------------|------------------------|--------------------------------------------------|-------------------------------------|------------------------------|
| <code>(cons 1<br/>  (cons 2<br/>    '()))</code>                     | 1                      | ???                                              | <code>#true</code>                  | <code>#false</code>          |
| <code>(cons 3<br/>  (cons 2<br/>    '()))</code>                     | 3                      | <code>(cons 2 '())</code>                        | ???                                 | <code>#true</code>           |
| <code>(cons 0<br/>  (cons 3<br/>    (cons 2<br/>      '()))))</code> | 0                      | <code>(cons 3<br/>  (cons 2<br/>    '()))</code> | ???                                 | ???                          |

Figure 58: A table for `sorted>?`

**Exercise 146.** Design `how-many` for *NEList-of-temperatures*. Doing so completes `average`, so ensure that `average` passes all of its tests, too.

**Exercise 147.** Develop a data definition for *NEList-of-Booleans*, a representation of non-empty lists of Boolean values. Then redesign the functions `all-true` and `one-true` from [exercise 140](#).

**Exercise 148.** 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?

## 9.3 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")
(cons "hello" (cons "hello" '()))
> (make-list 3 #true)
(cons #true (cons #true (cons #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”, that is, 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:

```
; An N is one of:
; - 0
; - (add1 N)
; interpretation represents the counting numbers
```

The first clause says that `0` is a natural number; it is 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 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 on natural numbers, such as `make-list`, yourself. The data definition is already available, so let's add the header material:

```
; N String -> List-of-strings
; creates a list of n copies of s

(check-expect (copier 0 "hello") '())
```

```

(check-expect (copier 2 "hello")
 (cons "hello" (cons "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 corresponding 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 copies of s

(check-expect (copier 0 "hello") '())
(check-expect (copier 2 "hello")
 (cons "hello" (cons "hello" '())))

(define (copier n s)
 (cond
 [(zero? n) '()]
 [(positive? n) (cons s (copier (sub1 n) s))]))

```

Figure 59: Creating a list of copies

Figure 59 contains a complete definition of the copier function, as obtained from its template. Let's 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 its expressions 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)` produces a list of `(sub1 n)` strings `s` according to the purpose statement of copier.

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 149.** 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 [Abstraction](#) for an answer.

An alternative definition of copier might use `else`:

```
(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 `"x"`? Explain. Use DrRacket's stepper to confirm your explanation.

**Exercise 150.** Design the function `add-to-pi`. It consumes a natural number `n` and adds it to `pi` without using the primitive `+` operation. 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 151.** Design the function `multiply`. It consumes a natural number `n` and multiplies it with a 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 152.** 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`.

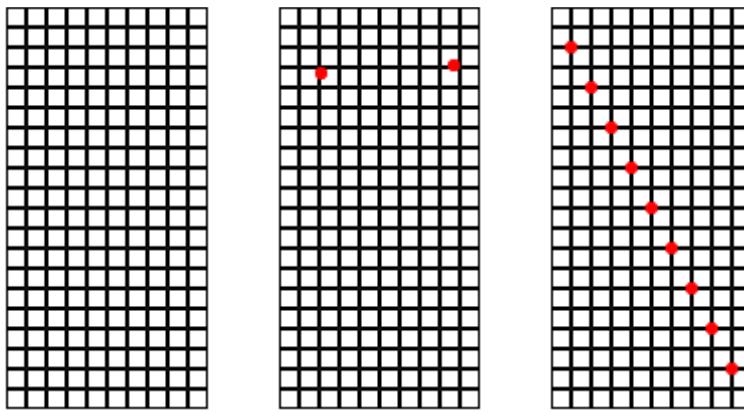


Figure 60: Random attacks

**Exercise 153.** 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 program displays how the balloons color the seats in the lecture hall.

Use the two functions from [exercise 152](#) 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-balloons`. The function consumes a list of [Posn](#)s 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](#).

[Figure 60](#) shows the output of **our** solution when given some list of [Posns](#). The left-most 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?

## 9.4 Russian Dolls

Wikipedia defines a Russian doll as “a set of dolls of decreasing sizes placed one inside the other” and illustrates it with this picture:



In this picture, the dolls are taken apart so that the viewer can see them all.

The problem may strike you as abstract or even absurd; it isn’t clear why you would want to represent Russian dolls or what you would do with such a representation. Just play along for now.

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 (short for 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:



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 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 of 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's 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 data examples, let's 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 to use the `string?` and `layer?` predicates. 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
- `(depth (layer-doll an-rd))` determines how many dolls are part of `(layer-doll an-rd)`, according to the purpose statement of `depth`.

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]
 [else (+ (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.

Let's finally translate the examples into tests:

```

(check-expect (depth "red") 1)
(check-expect
 (depth
 (make-layer "yellow" (make-layer "green" "red"))))
 3)

```

If you run these in DrRacket, you will see that their evaluation touches all pieces of the definition of `depth`.

**Exercise 154.** Design the function `colors`. It consumes a Russian doll and produces a string of all colors, separated by a comma and a space. Thus our example should produce

```
"yellow, green, red"
```

**Exercise 155.** 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`.

## 9.5 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's 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, reread [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:

```

(define HEIGHT 80) ; distances in terms of pixels
(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

```

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 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.

You can also use a data definition like this to represent this world:

```

; A ShotWorld is List-of-numbers.
; interpretation each number on such a list
; 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 the image of a shot for 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 on w 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 player presses the space bar
(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 61](#) includes this one function that is not designed but defined as a modification of standard schema.

Let's 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 9 '())`. 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 9 '()))
 (place-image SHOT XSHOTS 9 BACKGROUND))
```

Before you read on, work through an example that applies `to-image` to a list of two `Shots`. Doing so helps understand **how** the function works.

The fourth step is about translating 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 52](#) 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 53](#). 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

- `(to-image (rest w))` adds each shot on `(rest w)` to the background image, according to the purpose statement of `to-image`.

In other words, `(to-image (rest w))` renders the rest of the list as an image and thus performs almost all the work. What is missing 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.

[Figure 61](#) 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 nontrivial data definitions. On the one hand, the `ShotWorld` is a 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.

```
; 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 is hit
(define (keyh w ke)
 (if (key=? ke " ") (cons HEIGHT w) w))

; ShotWorld -> Image
; adds each shot y on w at {XSHOTS,y} to BACKGROUND
(define (to-image w)
 (cond
 [(empty? w) BACKGROUND]
 [else (place-image SHOT XSHOTS (first w)
 (to-image (rest w))))]))
```

Figure 61: A list-based world program

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 156.** Equip the program in [figure 61](#) with tests and make sure it passes those. Explain what `main` does. Then run the program via `main`.

**Exercise 157.** Experiment to determine whether the arbitrary decisions concerning constants are 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 or to change its color to black.

**Exercise 158.** If you run `main`, press the space bar (fire a shot), and wait for a goodly 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 that doesn’t just move shots one pixel per clock tick but also eliminates those whose coordinates place them above the canvas. **Hint** You may wish to consider the design of an auxiliary function for the recursive `cond` clause.

**Exercise 159.** Turn the solution of [exercise 153](#) 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 added to lob
```

(2) A **big-bang** expression is really just an expression. It is legitimate to nest it within another expression.

(3) Recall that **random** creates random numbers.

---

## 9.6 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 represent 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
; determines how often s occurs in los
(define (count los s)
 0)
```

Stop! Finish the design of this function.

Let's proceed in a 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.

```
; A Son.L is one of: ; A Son.R is one of:
; - empty ; - empty
; - (cons Number Son.L) ; - (cons Number Son.R)
;
; Son is used when it ;
; applies to Son.L and Son.R ; Constraint If s is a Son.R,
; ; no number occurs twice in s
```

Figure 62: Two data representations for sets

Figure 62 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 -> Boolean
; 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 ; Number Son.R -> Son.R
```

```

; removes x from s ; removes x from s
(define s1.L (define s1.R
 (cons 1 (cons 1 '())))) (cons 1 '()))

(check-expect (check-expect
 (set-.L 1 s1.L) es) (set-.R 1 s1.R) es)

(define (set-.L x s) (define (set-.R x s)
 (remove-all x s)) (remove x s))

```

Figure 63: Functions for the two data representations of sets

The difference between the two data definitions shows up when we design functions. 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
; subtracts 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 63](#) 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 63](#) 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 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, `(set- 1 set123)` evaluates to one of these two lists:

```

(define set23-version1
 (cons 2 (cons 3 '())))

(define set23-version2
 (cons 3 (cons 2 '())))

```

```
(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 precisely this idea.

[Intermezzo 1: Beginning Student Language](#) 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 | 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 299](#) introduces a completely different representation of sets, a

representation that can cope with infinite sets, too. The question of how actual programming languages represent sets is beyond the scope of this book, however.

**Exercise 160.** 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.

---

## 10 More on Lists

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 ideas from this world, including functions that create lists, data representations that call for structures inside of lists, and representing text files as lists.

---

### 10.1 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 one that computes the wages for all 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's 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 the weekly hours
(define (wage* whrs)
 '())
```

Next you need some examples of inputs and the corresponding outputs. So you make up some short lists of numbers that represent weekly hours:

| given                 | expected                 |
|-----------------------|--------------------------|
| '()                   | '()                      |
| (cons 28 '())         | (cons 336 '())           |
| (cons 4 (cons 2 '())) | (cons 48 (cons 24 '()))) |

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 '`1`'. Make sure you understand why the second and third expected outputs are 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 this template:

```
(define (wage* whrs)
 (cond
 [(empty? whrs) ...]
 [else (... (first whrs) ...
 ... (wage* (rest whrs)) ...)]))
```

In case you want to practice the development of templates, use the questions from [figure 52](#).

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? whrs)` 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 whrs)` yields the first number on `whrs`, which is the first number of hours worked;
- `(rest whrs)` is the rest of the given list; and
- `(wage* (rest whrs))` 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 if the function is not yet defined.

Since we already have the list of wages for all but the first item of `whrs`, the function must perform two computations to produce the expected output for the **entire** `whrs`: compute the weekly wage for `(first whrs)` and construct the list that represents all weekly wages for `whrs`. For the first part, we reuse `wage`. For the second, we `cons` the two pieces of information together into one list:

```
(cons (wage (first whrs)) (wage* (rest whrs)))
```

And with that, the definition is complete: see [figure 64](#).

```
; List-of-numbers -> List-of-numbers
; computes the weekly wages for all given weekly hours
(define (wage* whrs)
 (cond
 [(empty? whrs) '()]
 [else (cons (wage (first whrs)) (wage* (rest whrs))))]

; Number -> Number
; computes the wage for h hours of work
(define (wage h)
```

(\* 12 h))

Figure 64: Computing the wages of all employees

**Exercise 161.** Translate the examples into tests and make sure they all succeed. Then change the function in [figure 64](#) 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 162.** 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 64](#) if we want to perform this basic reality check?

**Exercise 163.** Design `convertFC`. The function converts a list of measurements in Fahrenheit to a list of Celsius measurements.

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.

**Exercise 164.** 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.

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 165.** 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 `substitute`. The latter 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`.

## 10.2 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 represent employees plus their work hours and pay rates:

```
(define-struct work [employee rate hours])
; A (piece of) Work is a structure:
; (make-work String Number Number)
; interpretation (make-work n r h) combines the name
; with the pay rate r and the number of hours h
```

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 (short for list of works) is one of:
; - '()
; - (cons Work Low)
; interpretation an instance of Low represents the
; hours worked for 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.

When you work on real-world projects, you won't use such suffixes; instead you will use a tool for managing different versions of code.

Now that you know that the definition of `Low` makes sense, it is time to redesign the function `wage*` so that it consumes elements of `Low`, not just lists of numbers:

```

; Low -> List-of-numbers
; computes the weekly wages for the given 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.

The third step of the design recipe is to work through an example. Let's 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, that is, `(* 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. Stop! 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 template lists **all** potentially interesting data.

We use a different strategy here. Specifically, we suggest that you **create and 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 processing elements of Work
(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 grows 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. `(wage*.v2 (rest an-low))` determines the list of wages for all the work records other than the first one, according to the purpose statement of the function.

If you are stuck here, use the table method from [figure 54](#).

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's 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)
```

The design of this kind of function is extensively covered in [Fixed-Size Data](#) and thus doesn't need any additional explanation here. [Figure 65](#) shows the final result of developing `wage` and `wage*.v2`.

```
; Low -> List-of-numbers
; computes the weekly wages for all 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 65: Computing the wages from work records

**Exercise 166.** 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 paychecks. Assume that a paycheck contains two distinctive pieces of information: the employee's name and an amount. Then design the function `wage*.v3`. It consumes a list of work records and computes a list of paychecks from it, one per record.

In reality, a paycheck 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 paychecks 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 paychecks.

**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 paychecks 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 167.** Design the function `sum`, which consumes a list of `Posns` and produces the sum of all of its x-coordinates.

**Exercise 168.** 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 169.** Design the function `legal`. Like `translate` from [exercise 168](#), 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 170.** 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 a Number between 100 and 999.
; A Four is a Number between 1000 and 9999.
```

Design the function `replace`. It consumes and produces a list of `Phones`. It replaces all occurrence of area code `713` with `281`.

---

## 10.3 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

Add `(require 2htdp/batch-io)` to your definitions area.

(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.

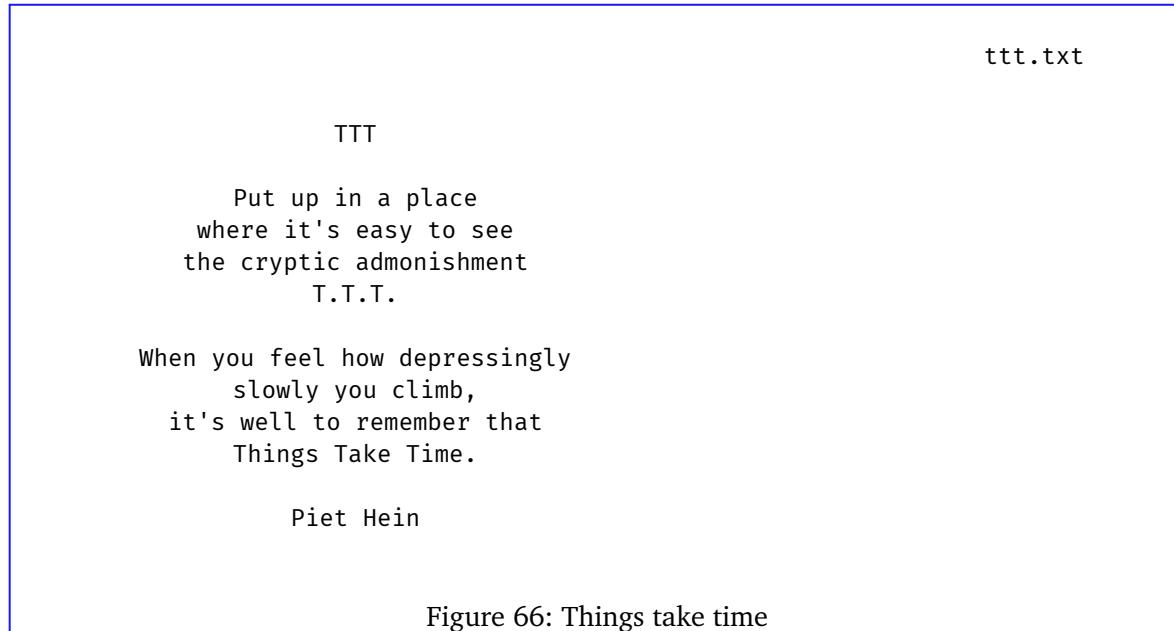


Figure 66: Things take time

For concreteness, take a look at the sample file in [figure 66](#). 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 dots 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, for example, `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 words, again each word represented as 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 relies on lists of lists of words:

```
(cons (cons "TTT" '())
 (cons '()
 (cons (cons "Put"
 (cons "up"
 (cons ... '())))
 (cons ...
 '())))))
```

This representation has an advantage over the second one in that it preserves the organization of the file, including the emptiness of the second line. The price is that all of a sudden lists contain lists.

While the idea of list-containing lists may sound frightening at first, you need not worry. The design recipe helps even with such complications.

```
; 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) ...)

; The above functions consume the name of a file as a String
; argument. If the specified file does not exist in the
; same folder as the program, they signal an error.
```

Figure 67: Reading files

Before we get started, take a look at [figure 67](#). It introduces a number of useful file reading functions. They are not comprehensive: 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.

[Figure 67](#) uses 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 171.** 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 instance 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 files. One counts the number of lines, another determines 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 function `words-on-line` because it is appropriate and captures the purpose statement in one phrase.

What is really needed, though, is a set of **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 examples. When you 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 '())))

```

By doing it at the end of the second step, 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 52](#), 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? lls) ...]
 [else
 (... (first ln) ; a string
 ... (line-processor (rest ln)) ...)]))

```

The important differences are that `(first ln)` extracts a string from the list, and 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 53](#) 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 `'()`, that is, the empty list of numbers. The second case, concerning `cons`, 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; and

- `(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's call it `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 the 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 input 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 good to reuse existing functions, you may define `words#` as

```
(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.

```
; An LLS is one of:
; - '()
; - (cons Los LLS)
; interpretation a list of lines, each 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 68: Counting the words on a line

**Figure 68** 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 list of functions that come with BSL. Some may look obscure but may become useful in one of the upcoming problems. Using such functions saves your time, not ours.

With one small step, you can now design your first file utility:

```
; String -> List-of-numbers
; counts the words on each line in the given file
(define (file-statistic file-name)
 (words-on-line
 (read-words/line file-name)))
```

It merely composes a library function with `words-on-line`. 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 implemented with a function.

```
; 1String -> String
; converts the given 1String to a 3-letter numeric String

(check-expect (encode-letter "z") (code1 "z"))
(check-expect (encode-letter "\t")
 (string-append "00" (code1 "\t")))
(check-expect (encode-letter "a")
 (string-append "0" (code1 "a")))

(define (encode-letter s)
 (cond
 [(>= (string->int s) 100) (code1 s)]
 [(< (string->int s) 10)
 (string-append "00" (code1 s))]
 [(< (string->int s) 100)
```

```

(string-append "0" (code1 s)))))

; 1String -> String
; converts the given 1String into a String

(check-expect (code1 "z") "122")

(define (code1 c)
 (number->string (string->int c)))

```

Figure 69: Encoding strings

**Exercise 172.** 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, use the program like this:

```

(write-file "ttt.dat"
 (collapse (read-words/line "ttt.txt")))

```

To make sure the two files "ttt.dat" and "ttt.txt" are identical, remove all extraneous white spaces in your version of the T.T.T. poem.

**Exercise 173.** 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 174.** 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. [Figure 69](#) shows our encoding function for single letters. Before you start, explain these functions.

**Hints** (1) Use `read-words/line` to preserve the organization of the file into lines and words.  
(2) Read up on `explode` again.

**Exercise 175.** 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.

```

; Matrix -> Matrix
; transposes 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))))]))
```

Figure 70: Transpose a matrix

**Exercise 176.** Mathematics teachers may have introduced you to matrix calculations by now. 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

; A 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, and 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 function in [figure 70](#) 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.

Stop! Transpose `mat1` by hand, then read [figure 70](#). 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; and
- `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 `transpose` with some test cases.

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 a pair of strings or a structure that combines a string with an index to a current position (see [exercise 87](#)).

A third alternative is to use structures that combine two lists of [1Strings](#):

```
(define-struct editor [pre post])
; An Editor is a structure:
; (make-editor Lo1S Lo1S)
; An Lo1S is one of:
; - '()
; - (cons 1String Lo1S)
```

Before you wonder why, let's 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 reverse it.

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:

|          |
|----------|
| all good |
|----------|

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's start with `rev` because we clearly need this function to make sense out of the data definition. Its header material is straightforward:

```
; Lo1s -> Lo1s
; 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 (rest l))</code> | <code>(rev (rest l))</code>            | <code>(rev l)</code>                     |
|----------------------------------------------------|-------------------------------|----------------------------------------|------------------------------------------|
| <code>(cons "a" '())</code>                        | "a"                           | '()                                    | '()                                      |
| <code>(cons "a" (cons "b" (cons "c" '()))))</code> | "a"                           | <code>(cons "b" (cons "c" '()))</code> | <code>(cons "c" (cons "b" '()))))</code> |

Figure 71: Tabulating for `rev`

If these hints leave you stuck, remember to create a table from the examples. Figure 71 shows the table for two examples: `(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, because `(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 function definition, we proceed according to the recipe questions for step 5. Our first question is to formulate an answer for the “basic” case, that is, the first case here. If you worked 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 `1Strings`. And naturally, it is called `reverse`.

**Exercise 177.** 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—for example, the cursor. Here are ours:

```
(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

(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)(index "editor-kh")
```

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]))
```

Reread [exercise 177](#) to determine the initial editor for this program.

While it does not matter which wish you tackle next, we choose to design `editor-kh` first and `editor-render` second. Since we have the header material, let's 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, this design case calls for an improved 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 (“`left`”, “`\b`”, and so on), some need to be ignored because they are special (“`\t`” and “`\r`”), and some should be classified into one large group (ordinary keys).

**Exercise 178.** 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, 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's 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 examples for `editor-ins` from those 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 in terms of information and what the function call is supposed to achieve in those terms. In this particular case, it is best to draw the visual representation of the editor because it represents the information well.

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)))
```

Remember a template lists parameters because they are available, too.

From the template and the examples, it is relatively easy to conclude that `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 lists of [1String](#)s, there is no need to design auxiliary functions. To get the desired result, it suffices to use `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 as if this isn't enough, you may wish to compare this simple definition with the one from [exercise 84](#) and figure out why the other one needs an auxiliary function while our definition here doesn't.

**Exercise 179.** 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 a 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 the 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's 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))

```

This dummy definition produces an empty text image.

To demonstrate what `editor-text` is supposed to compute, we work through an example. The example input is

```
(create-editor "pre" "post")
```

which was also used to explain `editor-render` and 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 `1String`s, 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 `1String`s. Naturally, there is a function `implode` that performs the inverse computation, that is,

```

> (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 180.** Design `editor-text` without using `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 typeset backward. We forgot that this part of the editor’s state is represented in reverse. Fortunately, the unit tests for the two functions pinpoint which function is wrong and even tell us what is wrong with the function and suggest how to fix the problem:

```
(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))
```

This definition 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](#).

---

## 11 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 several scenarios that call for the design of programs that compose many functions. To support this kind of design, the chapter presents some informal guidelines on divvying up functions and composing them. Since these examples demand complex forms of lists, however, this chapter starts with a section on concise list notation.

---

### 11.1 The `list` Function

At this point, you should have tired of writing so many `cons`es 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.

You have graduated from BSL. It is time to use the “Language” menu and to select “Beginning Student with List Abbreviations” for your studies.

The key innovation is `list`, which consumes an arbitrary number of values and creates a list. The simplest way to understand `list` is to think of it as an abbreviation. Specifically, every expression of the shape

```
| (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 actually the rest of the list. Here is a table with three examples:

| short-hand                        | long-hand                                                                 |
|-----------------------------------|---------------------------------------------------------------------------|
| ( <code>list</code> "ABC")        | ( <code>cons</code> "ABC" '())                                            |
| ( <code>list</code> #false #true) | ( <code>cons</code> #false ( <code>cons</code> #true '())))               |
| ( <code>list</code> 1 2 3)        | ( <code>cons</code> 1 ( <code>cons</code> 2 ( <code>cons</code> 3 '())))) |

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))
(list 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 contain 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" 7 "a")
| (list "john" 8 "d")
| (list "karel" 9 "e")))
```

requires 11 uses of `list`, which sharply contrasts with 40 `cons` and 11 additional uses of '`()`'.

**Exercise 181.** Use `list` to construct the equivalent of these lists:

1. (`cons` "a" (`cons` "b" (`cons` "c" (`cons` "d" '())))))

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

```
3. (cons "a" (cons (cons 1 '()) (cons #false '()))))
```

```
4. (cons (cons "a" (cons 2 '())) (cons "hello" '())))
```

Also try your hand at this one:

```
(cons (cons 1 (cons 2 '())))
 (cons (cons 2 '())
 '()))
```

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 182.** Use `cons` and `'()` to form the equivalent of these lists:

```
1. (list 0 1 2 3 4 5)
```

```
2. (list (list "he" 0) (list "it" 1) (list "lui" 14))
```

```
3. (list 1 (list 1 2) (list 1 2 3))
```

Use `check-expect` to express your answers.

**Exercise 183.** On some occasions lists are formed with `cons` and `list`.

```
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 '()))))
```

Reformulate each of the following expressions using only `cons` or only `list`. Use `check-expect` to check your answers.

**Exercise 184.** Determine the values of the following expressions:

```
1. (list (string=? "a" "b") #false)
```

```
2. (list (+ 10 20) (* 10 20) (/ 10 20))
```

```
3. (list "dana" "jane" "mary" "laura")
```

Use `check-expect` to express your answers.

**Exercise 185.** 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` and `fourth` exist.

## 11.2 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.

*Design one function per task.*

*Formulate auxiliary function definitions for every dependency between quantities in the problem.*

This part of the book introduces another guideline on auxiliary functions:

*Design one template per data definition. Formulate auxiliary function definitions when one data definition points to a second 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, depends on a number being positive, zero, or negative—use a `cond` expression. If the `cond` looks complex, design an auxiliary function whose arguments are the template's expressions and whose body is the `cond` expression.
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 counterintuitive, 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 continue, though, remember that the key to managing the design of programs is to maintain the often-mentioned

### Wish List

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.

---

### 11.3 Auxiliary Functions that Recur

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 criterion.

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 reals.

The exercises below clarify how to adapt this function to other data.

Since the problem statement does not mention any other task and since sorting does not seem to suggest other tasks, we just follow the design recipe. Sorting means rearranging a bunch of numbers. This restatement 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 '()))))`, that is, a list with the numbers arranged in descending order; and

- `(cons -5 (cons 12 (cons 20 '())))`, that is, 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
; rearranges alon in descending order

(check-expect (sort> '()) '())
(check-expect (sort> (list 3 2 1)) (list 3 2 1))
(check-expect (sort> (list 1 2 3)) (list 3 2 1))
(check-expect (sort> (list 12 20 -5))
 (list 20 12 -5))

(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 already 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; and
- `(sort> (rest alon))` rearranges `(rest alon)` in descending order, according to the purpose statement of the function.

To clarify these abstract answers, let's 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` along) in its proper place into the result of (`sort> (rest along)`). 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 item 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 along
(define (insert n along) along)
```

That is, `insert` consumes a number and a list sorted in descending order and produces a sorted list by inserting the former into the latter.

With `insert`, it is easy to complete the definition of `sort>`:

```
(define (sort> along)
 (cond
 [(empty? along) '()]
 [else
 (insert (first along) (sort> (rest along))))]))
```

In order to produce the final result, `sort>` extracts the first item of a non-empty list, computes the sorted version of the rest, and uses `insert` to produce the completely sorted list from the two pieces.

Stop! Test the program as is. Some test cases pass, and some fail. That's progress. 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))
```

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's 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 53](#):

1. `(first alon)` is the first number on `alon`;
2. `(rest alon)` is the rest of `alon` and, like `alon`, it is sorted in descending order; and
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's 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 72** contains the complete sort program. Copy it into the definitions 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 l
(define (sort> l)
 (cond
 [(empty? l) '()]
 [(cons? l) (insert (first l) (sort> (rest l))))]

; Number List-of-numbers -> List-of-numbers
; inserts n into the sorted list of numbers l
(define (insert n l)
 (cond
 [(empty? l) (cons n '())]
 [else (if (>= n (first l))
 (cons n l)
 (cons (first l) (insert n (rest l))))]))
```

Figure 72: Sorting lists of numbers

**Exercise 186.** Take a second look at [Intermezzo 1: Beginning Student Language](#), 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 145](#) to reformulate 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)
 (list 9 8 7 6 5 4 3 2 1 0))
```

Can you formulate a test case that shows that `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 specify `sort>` completely.

**Exercise 187.** 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
```

**Hint** Formulate a function that compares two elements of `GamePlayer`.

**Exercise 188.** Design a program that sorts lists of emails by date:

```
(define-struct email [from date message])
; An 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 189.** Here is the function `search`:

```
; Number List-of-numbers -> Boolean
(define (search nalon)
 (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 190.** Design the `prefixes` function, which consumes a list of `1Strings` and produces the list of all prefixes. A list `p` is a *prefix* of `l` if `p` and `l` are the same up through all items in `p`. For example, `(list "a" "b" "c")` is a prefix of itself and `(list "a" "b" "c" "d")`.

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 "b" "c" "d")` is a suffix of itself and `(list "a" "b" "c" "d")`.

---

## 11.4 Auxiliary Functions that Generalize

On occasion an auxiliary function is not just a small helper function but a solution to a more general problem. Such auxiliaries are needed when a problem statement is too narrow. As programmers work through the steps of the design recipe, they may discover that the “natural” solution is wrong. An analysis of this broken solution may suggest a slightly different, but more general, problem statement, as well as a simple way of using the solution to the general problem for the original one.

We illustrate this idea with a solution to the following problem:

**Sample Problem** Design a function  
that adds a polygon to a given  
scene.

Paul C. Fisher suggested this problem.

Just in case you don’t recall your basic geometry (domain) knowledge, we add a (simplistic) definition of polygon:

A *polygon* is a planar figure with at least three points (not on a straight line) connected by three straight sides.

One natural data representation for a polygon is thus a list of `Posns`. For example, the following two definitions

```
(define triangle-p (define square-p
 (list (list
 (make-posn 20 10) (make-posn 10 10)
 (make-posn 20 20) (make-posn 20 10)
 (make-posn 30 20))) (make-posn 20 20)
 (make-posn 10 20)))
```

introduce a triangle and a square, just as the names say. Now you may wonder how to interpret `'()` or `(list (make-posn 30 40))` as polygons, and the answer is that they do **not** describe polygons. Because a polygon consists of at least three points, a good data representation of polygons is the collection of lists with at least three `Posns`.

Following the development of the data definition for non-empty lists of temperatures ([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` a `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 is that a naively chosen data representation—plain lists of `Posns`—may not properly represent the intended information. Revising the data definition during an initial exploration is normal; 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 design.

The second step calls for the signature, purpose statement, and header of the function. Since the problem statement mentions just one task and no other task is implied, we start with one function:

```
; a plain background image
(define MT (empty-scene 50 50))

; Image Polygon -> Image
; renders the given polygon p into img
(define (render-poly img p)
 img)
```

The additional definition of `MT` is called for because it simplifies the formulation of 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 MT triangle-p)
 (scene+line
 (scene+line
 (scene+line MT 20 10 20 20 "red")
 20 20 30 20 "red")
 30 20 20 10 "red")))
```

The innermost `scene+line` renders 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 interactions area to get this expression right.

Given that the first and smallest polygon is a triangle, then a rectangle or a square suggests itself as the second example. We use `square-p`:

```
(check-expect
 (render-poly MT square-p)
 (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"))
```

A square is just one more point than a triangle, and it is easy to render. You may also wish to draw these shapes on a piece of graph paper.

The construction of the template poses a challenge. Specifically, the first and the second questions of [figure 52](#) 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`, while 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, that is,

```
(cons Posn (cons Posn (cons Posn '()))))
```

suggests a second way to formulate the first 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 52](#) have direct answers, and thus we get this template:

It is truly better to formulate conditions in terms of built-in predicates and selectors than your own (recursive) functions. See [Intermezzo 5: The Cost of Computation](#) for an explanation.

```
(define (render-poly img p)
 (cond
 [(empty? (rest (rest (rest p))))
 (... (first p) ... img ...
 ... (second p) ...
 ... (third p) ...)]
 [else (... (first p) ...
 ... (render-poly img (rest p)) ...)])))
```

Because `p` describes a triangle in the first clause, it must consist of exactly three `Posns`, which are extracted via `first`, `second`, and `third`. In the second clause, `p` 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 it; we must 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 suggests a straightforward answer. Specifically, there are three `Posns` and `render-poly` should connect the three in an empty scene of 50 by 50 pixels. Given that `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` is 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 `Posn`. In the template, we find two expressions, and, following [figure 53](#), 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 img (rest p))` renders `(rest p)`, which is what the purpose statement of the function says.

The question is how to use these pieces to render the 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 MT (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 analysis suggests a rather natural, complete function definition:

```
(define (render-poly img p)
 (cond
 [(empty? (rest (rest (rest p))))]
 (render-line
 (render-line
 (render-line
 (render-line MT (first p) (second p))
 (second p) (third p)))
```

```

 (third p) (first p))]
[else
 (render-line (render-poly img (rest p))
 (first p)
 (second p)))]

```

Designing `render-line` is the kind of problem that you solved in the first part of the book. Hence we just provide the final definition so that you can test the above function:

```

; Image Posn Posn -> Image
; renders a line from p to q into img
(define (render-line img p q)
 (scene+line
 img
 (posn-x p) (posn-y p) (posn-x q) (posn-y q)
 "red"))

```

Stop! Develop a test for `render-line`.

Lastly, we must test the functions. The tests for `render-poly` fail. On the one hand, the test failure 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 natural choices, and yet the function doesn't work.

Stop! Why do you think the tests fail? Draw an image of the pieces in the template of `render-poly`. Then draw the line that combines them. Alternatively, experiment in DrRacket's interactions area:

```
> (render-poly MT square-p)
```



The image shows that `render-polygon` connects the three dots of `(rest p)` and then connects `(first p)` to the first point of `(rest p)`, that is, `(second p)`. You can easily validate this claim with an interaction that uses `(rest square-p)` directly as input for `render-poly`:

```
> (render-poly MT (rest square-p))
```



In addition, you may wonder what `render-poly` would draw if we added another point, say, `(make-posn 40 30)`, to the original square:

```
> (render-poly
 MT
 (cons (make-posn 40 30) square-p))
```



Instead of the desired pentagon, `render-polygon` always draws the triangle at the end of the given `Polygon` and otherwise connects the `Posns` that precede the triangle.

While the experiments confirm the problems of our design, they also suggest that the function is “almost correct.” It connects the successive dots specified by a list of `Posns`, and then it draws a line from the first to the last `Posn` of the trailing triangle. If it skipped this last step, the function would just “connect the dots” and thus draw an “open” polygon. By connecting the first and the last point, it could then complete its task.

Put differently, the analysis of our failure suggests a two-step solution:

1. Solve a **more general** problem.
2. Use the solution to this general problem to solve the original one.

We start with the statement for the general problem:

**Sample Problem** Design a function that draws connections between a given bunch of dots and into a given scene.

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 a couple of dots. To keep things simple, we go with at least one dot:

```
; An NELoP is one of:
; - (cons Posn '())
; - (cons Posn NELoP)
```

Second, we formulate a signature, a purpose statement, and a header for a “connect the dots” function:

```
; Image NELoP -> Image
; connects the dots in p by rendering lines in img
(define (connect-dots img 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 MT triangle-p)
 (scene+line
 (scene+line MT 20 0 10 10 "red")
 10 10 30 10 "red"))
```

The expected value is an image with two lines: one from the first `Posn` to the second one, and another one from the second to the third `Posn`.

**Exercise 191.** Adapt the second example for the `render-poly` function to `connect-dots`.

Fourth, we use the template for functions that process non-empty lists:

```
(define (connect-dots img p)
 (cond
 [(empty? (rest p)) (... (first p) ...)]
 [else (... (first p) ...
 ... (connect-dots img (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 img (rest p))` connects the dots in `(rest p)` by rendering lines in `img`.

From our first attempt to design `render-poly`, we know `connect-dots` needs to add one line to the result of `(connect-dots img (rest p))`, 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 img p)
 (cond
 [(empty? (rest p)) img]
 [else
 (render-line
 (connect-dots img (rest p))
 (first p)
 (second p))))
```

This definition looks simpler than the faulty version of `render-poly`, even though it copes with two more lists of `Posns` than `render-poly`.

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 an `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

This argument is **informal**. If you ever need a **formal** argument for such claims about the relationship between sets or functions, you will need to study *logic*. Indeed, this book's design process is deeply informed by logic, and a course on logic in computation is a natural complement. In general, logic is to computing what analysis is to engineering.

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.

As spelled out above, `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:

```
; Image Polygon -> Image
; adds an image of p to img
(define (render-polygon img p)
 (render-line (connect-dots img 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 last?

**Exercise 192.** Argue why it is acceptable to use last on Polygons. Also argue why you may adapt the template for connect-dots to 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 73 works on your examples.

```
; Image Polygon -> Image
; adds an image of p to MT
(define (render-polygon img p)
 (render-line (connect-dots img p) (first p) (last p)))

; Image NELoP -> Image
; connects the Posns in p in an image
(define (connect-dots img p)
 (cond
 [(empty? (rest p)) MT]
 [else (render-line (connect-dots img (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 73: 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 193.** 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 `Polygon`s.

Use both ideas to define `render-poly`; make sure both definitions pass the test cases.

**Exercise 194.** Modify `connect-dots` so that it consumes an additional `Posn` to which the last `Posn` is connected. Then modify `render-poly` to use this new version of `connect-dots`.

Naturally, functions such as `last` are available in a full-fledged programming language, and something like `render-poly` is available in the `2htdp/image` library. If you are wondering why we just designed these functions, consider the titles of both the book and this section. The goal is **not** (just) to design useful functions but to study how code is designed systematically. Specifically, this section is about the idea of generalization in the design process; for more on this idea see [Abstraction](#) and [Accumulators](#).

---

## 12 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

This chapter relies on the `2htdp/batch-io` library.

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.

## 12.1 Real-World Data: Dictionaries

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. 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.

For performance concerns, see [Generative Recursion](#). From here to there, the focus is on designing programs systematically so that you can then explore performance problems properly.

```
; On OS X:
(define LOCATION "/usr/share/dict/words")
;
; On LINUX: /usr/share/dict/words or /var/lib/dict/words
;
; On WINDOWS: borrow the word file from your Linux friend

;
; A Dictionary is a List-of-strings.
(define AS-LIST (read-lines LOCATION))
```

Figure 74: Reading a dictionary

Figure 74 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, July 25, 2017.

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 74:

```
; A Letter is one of the following 1Strings:
; - "a"
; - ...
; - "z"
;
; or, equivalently, a member? of this list:
(define LETTERS
 (explode "abcdefghijklmnopqrstuvwxyz"))
```

**Hint** Use `list` to formulate examples and tests for the exercises.

**Exercise 195.** 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 196.** Design `count-by-letter`. The function consumes a `Dictionary` and counts how often each letter is used as the first one of a word in the given dictionary. Its result is a list of `Letter-Counts`, a piece of data that combines letters and counts.

Once your function is designed, determine how many words appear for all letters in your computer's dictionary.

**Note on Design Choices** An alternative is to design an auxiliary function that consumes a list of letters and a dictionary and produces a list of [Letter-Counts](#) that report how often the given letters occur as first ones in the dictionary. You may of course reuse your solution of [exercise 195](#). **Hint** If you design this variant, notice that the function consumes two lists, requiring a design problem that is covered in [Simultaneous Processing](#) in detail. Think of [Dictionary](#) as an atomic piece of data that is along for the ride and is handed over to `starts-with#` as needed.

**Exercise 197.** Design `most-frequent`. The function consumes a [Dictionary](#). It produces the [Letter-Count](#) for the letter that occurs most often as the first one in 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 from a list of [Letter-Counts](#). There are two ways to design this latter function:

- Design a function that picks the pair with the maximum count.
- Design a function that selects the first from a sorted list of pairs.

Consider designing both. Which one do you prefer? Why?

**Exercise 198.** Design `words-by-first-letter`. The function consumes a [Dictionary](#) and produces a list of [Dictionaries](#), one per [Letter](#).

Redesign `most-frequent` from [exercise 197](#) using this new function. 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:

```
(check-expect
 (most-frequent AS-LIST)
 (most-frequent.v2 AS-LIST))
```

**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 of the word-counting function 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 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.

## 12.2 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 [Project: 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 kinds of information for each music track, occasionally a bit less:

- *Track ID*, a unique identifier for the track with respect to your library, example: 442
- *Name*, the title of the track, `Wild Child`
- *Artist*, the producing artists, Enya
- *Album*, the title of the album to which it belongs, `A Day Without`
- *Genre*, the music genre to which the track is assigned, `New Age`
- *Kind*, the encoding of the music, `MPEG audio file`
- *Size*, the size of the file, 4562044
- *Total Time*, the length of the track in milliseconds, 227996
- *Track Number*, the position of the track within the album, 2
- *Track Count*, the number of tracks on the album, 11
- *Year*, the year of release, 2000
- *Date Added*, when the track was added, 2002-7-17 3:55:14
- *Play Count*, how many times it was played, 20
- *Play Date*, when the track was last played, 3388484113 Unix seconds
- *Play Date UTC*, when it was last played, 2011-5-17 17:35:13

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.

In addition to the `2htdp/batch-io` library, this section relies on the `2htdp/itunes` library.

```

; the 2htdp/itunes library documentation, part 1:

; An LTracks is one of:
; - '()
; - (cons Track LTracks)

(define-struct track
 [name artist album time track# added play# played])
; A Track is a structure:
; (make-track String String String N N Date N)
; 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 within the
; album, the date it was added, how often it has been
; played, and the date when it was last played

(define-struct date [year month day hour minute second])
; A Date is a structure:
; (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).

```

Figure 75: Representing iTunes tracks as structures (the structures)

```

; Any Any Any Any Any Any Any -> Track or #false
; creates an instance of Track for legitimate inputs
; otherwise it produces #false
(define (create-track name artist album time
 track# added play# played)
 ...)

; Any Any Any Any Any -> Date or #false
; creates an instance of Date for legitimate inputs
; otherwise it produces #false
(define (create-date y mo day h m s)
 ...)

; String -> LTracks
; creates a list-of-tracks representation from the
; text in file-name (an XML export from iTunes)
(define (read-itunes-as-tracks file-name)
 ...)

```

Figure 76: Representing iTunes tracks as structures (the functions)

Figures 75 and 76 introduce 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 some iTunes app, you can run the following code snippet to retrieve all the records:

```
; modify the following to use your chosen name
(define ITUNES-LOCATION "itunes.xml")

; LTracks
(define itunes-tracks
 (read-itunes-as-tracks ITUNES-LOCATION))
```

Save the snippet in the same folder as your iTunes XML export. Remember not 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 199.** 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 200.** Design the function `total-time`, which consumes an element of [LTracks](#) and produces the total amount of play time. Once the program is done, compute the total play time of your iTunes collection.

**Exercise 201.** 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 202.** 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 203.** 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 204.** Design `select-albums`. The function consumes an element of [LTracks](#). It produces a list of [LTracks](#), one per album. Each album is uniquely identified by its title and shows up in the result only once. **Hints** (1) You want to use some of the solutions of the preceding exercises. (2) 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 196](#).

**Terminology** The functions whose names starts with `select-` are so-called *database queries*. See [Project: Database](#) for more details. **End**

```
; the 2htdp/itunes library documentation, part 2:

; An LLists is one of:
; - '()
; - (cons LAssoc LLists)

; An LAssoc is one of:
; - '()
; - (cons Association LAssoc)
;
; An Association is a list of two items:
; (cons String (cons BSDN '()))

; A BSDN is one of:
; - Boolean
; - Number
; - String
; - Date

; String -> LLists
; creates a list of lists representation for all tracks in
; file-name, which must be an XML export from iTunes
(define (read-itunes-as-lists file-name)
 ...)
```

Figure 77: Representing iTunes tracks as lists

Figure 77 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, you get access to all track information if you add the following definitions to your program:

```
; modify the following to use your chosen name
(define ITUNES-LOCATION "itunes.xml")

; LLists
(define list-tracks
 (read-itunes-as-lists ITUNES-LOCATION))
```

Then save it in the same folder where the iTunes library is stored.

**Exercise 205.** Develop examples of `LAssoc` and `LLists`, that is, the list representation of tracks and lists of such tracks.

**Exercise 206.** 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.

**Note** Read up on `assoc` after you have designed this function.

**Exercise 207.** Design `total-time/list`, which consumes an [LLists](#) and produces the total amount of play time. **Hint** Solve [exercise 206](#) 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 200](#) computes. Why is there a difference?

**Exercise 208.** Design `boolean-attributes`. The function consumes an [LLists](#) and produces the [Strings](#) that are associated with a [Boolean](#) attribute. **Hint** Use `create-set` from [exercise 201](#).

Once you are done, 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**

---

## 12.3 Word Games, Composition Illustrated

Some of you solve word puzzles in newspapers and magazines. Try this:

**Sample Problem** Given a word, find all words that are made up from the same 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.”

**Note** If a word contains the same letter twice, the collection of all re-arrangements may contain several copies of the same string. For our purposes, this is acceptable. For a realistic program, you may wish to avoid duplicate entries by using sets instead of lists. See [A Note on Lists and Sets](#). **End**

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 search function, leaving the solution of the other problem to the next section. By separating the two, this first section can focus on the high-level ideas of systematic program design.

See [Real-World Data: Dictionaries](#) for dealing with real-world dictionaries.

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 signature and purpose statement:

```
; String -> List-of-strings
; finds all words that use the same letters as s
(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.

Assume we wish to formulate a test for `alternative-words` and “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.

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](#):

See [Intermezzo 1: Beginning Student Language](#).

```
; List-of-strings -> Boolean
(define (all-words-from-rat? w)
 (and (member? "rat" w)
```

```
(member? "art" w)
(member? "tar" w)))
```

With this function, it is easy to formulate a test for alternative-words:

```
(check-satisfied (alternative-words "rat")
 all-words-from-rat?)
```

**Note on Data versus Design** What this discussion suggests is that the alternative-words function 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 represent 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 the subject of the second book. **End**

```
; List-of-strings -> Boolean
(define (all-words-from-rat? w)
 (and
 (member? "rat" w) (member? "art" w) (member? "tar" w)))

; String -> List-of-strings
; finds all words that the letters of some given word spell

(check-member-of (alternative-words "cat")
 (list "act" "cat")
 (list "cat" "act"))

(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
; turns all Words in low into Strings
(define (words->strings low) '())

; List-of-strings -> List-of-strings
; picks out all those Strings that occur in the dictionary
(define (in-dictionary los) '())(index "in-dictionary")
```

Figure 78: 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 rearranging 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 rearrangements:

```
; A Word is ...
; A List-of-words is ...
```

```

; Word -> List-of-words
; finds all rearrangements of word
(define (arrangements word)
 (list word))

```

**Exercise 209.** 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 wish-list entries:

```

; String -> Word
; converts s to the chosen word representation
(define (string->word s) ...)

; Word -> String
; converts 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 rearrangements of the letters and `in-dictionary` to select those rearrangements that also occur in a dictionary.

Stop! Look up the signatures for the two functions to make sure the composition works out. What exactly do you need to check?

What this expression fails to capture is the fourth point, the decision not to use plain strings to rearrange the letters. Before we hand `s` to `arrangements`, we need to convert it into a word. Fortunately, [exercise 209](#) 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 209](#) asks for a function that converts a single word, here we need a function that deals with lists of them. 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 78 collects all the pieces. The following exercises ask you to design the remaining functions.

**Exercise 210.** Complete the design of the `words->strings` function specified in figure 78.

**Hint** Use your solution to [exercise 209](#).

**Exercise 211.** Complete the design of `in-dictionary`, specified in figure 78. Hint See [Real-World Data: Dictionaries](#) for how to read a dictionary.

## 12.4 Word Games, the Heart of the Problem

The goal is to design `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, `Strings` could serve as a representation of words, but a `String` is atomic and the very fact that `arrangements` needs to rearrange its letters calls for a different representation. 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 one of:
; - '()
; - (cons 1String Word)
; interpretation a Word is a list of 1Strings (letters)
```

**Exercise 212.** Write down the data definition for *List-of-words*. Make up examples of `Words` and `List-of-words`. Finally, formulate the functional example from above with `check-expect`. Instead of the full example, consider working 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 all rearrangements of the letters in w
(define (arrangements w)
 (cond
 [(empty? w) ...]
 [else (... (first w) ...
 ... (arrangements (rest w)) ...)]))
```

In preparation of the fifth step, let's look at the template's `cond` lines:

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. Also, 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's call this function `insert-everywhere/in-all-words` and let's 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 213.** 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, and the like. 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:

```
| (insert-everywhere/in-all-words "d"
| (cons (list "e" "r")
| (cons (list "r" "e")
| '()))))
```

(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"))
| (list "a" "b" "c" "d" "e")
```

The development of functions like `append` is the subject of [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 214.** 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.

## 12.5 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 reach 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 long enough, it can run into itself and, if it does, the game is over, too.

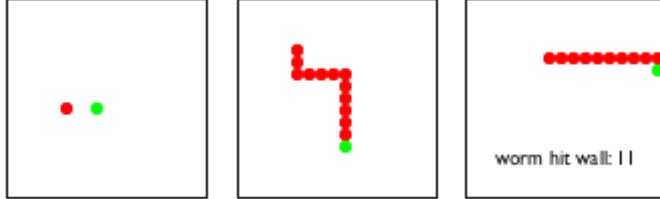


Figure 79: Playing Worm

Figure 79 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 nontrivial 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 215.** Design a world 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) Reread [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 216.** Modify your program from [exercise 215](#) so that it stops if the worm has reached 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.

**Exercise 217.** 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. To keep things simple, treat all segments—head and tail segments—the same.

Now modify your program from [exercise 215](#) to accommodate a multi-segment worm. Keep things simple: (1) your program may render all worm segments as red disks and (2) ignore that the worm may run into the wall or itself. **Hint** One way to realize the worm's movement is to add a segment in the direction in which it is moving and to delete the last one.

```
; 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=-1-1? p)
 (not (and (= (posn-x p) 1) (= (posn-y p) 1))))
```

Figure 80: Random placement of food

**Exercise 218.** Redesign your program from [exercise 217](#) 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 216](#) 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 `member?` function.

**Exercise 219.** Equip your program from [exercise 218](#) 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 a 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 218](#)) 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 these functions in [figure 80](#). Before you use them, however, explain how these functions work—assuming MAX is greater than 1—and then formulate purpose statements.

For the workings of `random`, read the manual or [exercise 99](#).

**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, 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 have a complete worm game. Now modify your `worm-main` function so that it returns the length of the final worm. Then use `Create Executable` (under the Racket menu) in DrRacket to turn your program into something that anybody can launch, not just someone who knows about BSL+.

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 placing extra obstacles in the room, and a few other ideas. What can you think of?

---

## 12.6 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.

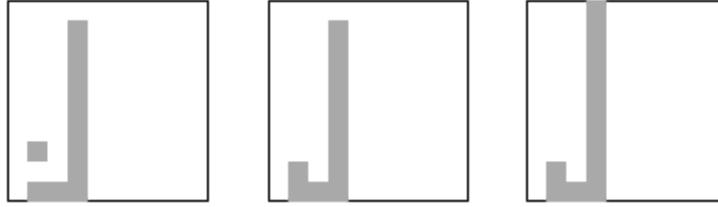


Figure 81: Simple Tetris

In our simplified version, the game starts with individual blocks dropping from the top of the scene. Once one of them 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; 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 81](#) for an illustration of the idea.

Given this description, we can turn to the design guidelines for interactive programs from [Designing World Programs](#). They call 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 simple Tetris game. 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:

```
(define WIDTH 10) ; # of blocks, horizontally
(define SIZE 10) ; blocks are squares
(define SCENE-SIZE (* WIDTH SIZE))

(define BLOCK ; red squares with black rims
 (overlay
 (square (- SIZE 1) "solid" "red")
 (square SIZE "outline" "black")))
```

Explain these definitions before you read on.

- The “landscapes” of blocks differ from game to game and from clock tick to clock tick. Let’s make this more precise. The appearance 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

See exercise 215 for a related design decision.

**physical** 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, and so forth.

In this section we choose the data representation for you:

```
(define-struct tetris [block landscape])
(define-struct block [x y])

; A Tetris is a structure:
; (make-tetris Block Landscape)
; A Landscape is one of:
; - '()
; - (cons Block Landscape)
; A Block is a structure:
; (make-block N N)

; interpretations
; (make-block x y) depicts a block whose left
; corner is (* x SIZE) pixels from the left and
; (* y SIZE) pixels from the top;
; (make-tetris b0 (list b1 b2 ...)) means b0 is the
; dropping block, while b1, b2, and ... are resting
```

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 220.** 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 interactions 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 221.** 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 blocks. **Hint** Read up on the `member?` primitive.

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 219](#) for inspiration.

**Exercise 222.** Modify the program from [exercise 221](#) so that a player can control the horizontal 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 “`right`”, the dropping block should move one column to the right if possible.

**Exercise 223.** Equip the program from [exercise 222](#) with a `stop-when` clause. The game ends when one of the columns contains enough blocks to “touch” the top of the canvas.

Once you have solved [exercise 223](#), 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 display 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.

---

## 12.7 Full Space War

[Itemizations and Structures](#) alludes to 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 exercises that help you complete this game.

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 224.** 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 guidelines for auxiliary functions. If you like the game, add other features: show a running text; equip the UFO with charges that can 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, use the same ideas to produce a similar, civilized game.

**Exercise 225.** 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 at random places on the ground. You may wish to limit the number of fires, making them a function of how many fires are currently burning or other factors. The purpose of the game is to extinguish all fires in a limited amount of time. **Hint** Use an iterative design approach as illustrated in this chapter to create this game.

---

## 12.8 Finite State Machines

Finite state machines (FSMs) and regular expressions are ubiquitous elements of programming. As [Finite State Worlds](#) explains, state machines are one way to think about world programs. Conversely, [exercise 109](#) shows how to design world programs that implement an FSM and check whether a player presses a specific series of keystrokes.

As you may also recall, a finite state machine is equivalent to a regular expression. Hence, computer scientists tend to say that an FSM accepts the keystrokes that match a particular regular expression, like this one

$$a (b|c)^* d$$

from [exercise 109](#). If you wanted a program that recognizes a different pattern, say,

$$a (b|c)^* a$$

you would just modify the existing 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 whole bunch of similar-looking programs.

A natural idea is to look for a general solution, that is, a world program that consumes a **data representation of an FSM** and 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 the matching ignores the actual keystrokes; instead the transition from one state to another takes place 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:

**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 an 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 476. End](#)

```
; An FSM is one of:
; - '()
; - (cons Transition FSM)

(define-struct transition [current next])
; A Transition is a structure:
; (make-transition FSM-State FSM-State)

; FSM-State is a Color.

; interpretation An FSM represents the transitions that a
; finite state machine can take from one state to another
; in reaction to keystrokes
```

Figure 82: 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 82](#) collects this information. It starts with a data definition for **FSMs**. As you can see, an **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 arbitrarily 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 226.** 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 227.** The BW Machine is an 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 with the given FSM
(define (simulate an-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 player's keystrokes.

Let's follow the design recipe for world programs anyway to see how far it takes us. It 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 an FSM-State.
```

According to the design recipe for world programs, this data definition completes the main function:

```
(define (simulate.v1 fsm0)
 (big-bang initial-state
 [to-draw render-state.v1]
 [on-key find-next-state.v1]))
```

The **empty-image** constant represents an “invisible” image. It is a good default value for writing down the headers of rendering functions.

and implies a wish list with two entries:

```
; SimulationState.v1 -> Image
; renders a world state as an image
(define (render-state.v1 s)
 empty-image)

; SimulationState.v1 KeyEvent -> SimulationState.v1
; finds the next state from ke and 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 grey 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 a keystroke?

This question rings especially true because, according to the simplified problem statement, the exact nature of the keystroke is 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` itself:

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 KeyEvent -> SimulationState.v2
; finds the next state from ke and 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 with the given FSM
(define (simulate.v2 an-fsm s0)
 (big-bang (make-fs an-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 machine and "red":

```
(simulate.v2 fsm-traffic "red")
```

Stop! Why do you think "red" is good for traffic lights?

**Note on Expressive Power** Given the work-around, we can now explain the limitation of BSL. Even though the given

Engineers call "red" the safe state.

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 [Local Definitions 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 an-fsm)
 (square 100 "solid" (fs-current an-fsm)))
```

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 ke and cs
(define (find-next-state an-fsm current)
 an-fsm)
```

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"))
```

The examples say that when the current state combines the fsm-traffic machine and its "red" state, the result combines the same FSM with "green", regardless of whether the player hit "n" or "a" on the keyboard. Here is one more example:

```
(check-expect
 (find-next-state (make-fs fsm-traffic "green") "q")
```

```
(make-fs fsm-traffic "yellow"))
```

Interpret the example before reading on. Can you think of another one?

Since the function consumes a structure, we write down a template for structures processing:

```
(define (find-next-state an-fsm ke)
 (... (fs-fsm an-fsm) ... (fs-current an-fsm) ...))
```

Furthermore, because the desired result is a [SimulationState.v2](#), we can refine the template with the addition of an appropriate constructor:

```
(define (find-next-state an-fsm ke)
 (make-fs
 ... (fs-fsm an-fsm) ... (fs-current an-fsm) ...))
```

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 an-fsm)`—and finish the definition:

```
(define (find-next-state an-fsm ke)
 (make-fs
 (fs-fsm an-fsm)
 (find (fs-fsm an-fsm) (fs-current an-fsm))))
```

Here is the formulation of the new wish:

```
; FSM FSM-State -> FSM-State
; finds the state representing current in transitions
; and retrieves the next field
(check-expect (find fsm-traffic "red") "green")
(check-expect (find fsm-traffic "green") "yellow")
(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 some additional examples, then tackle the exercises.

**Exercise 228.** Complete the design of `find`.

Once the auxiliary functions are tested, use `simulate` to play with `fsm-traffic` and the BW Machine from [exercise 227](#).

Our simulation program is intentionally quite restrictive. In particular, you cannot use it to represent **FSMs** that transition from one state to another depending on which key a player presses. Given the systematic design, though, you can extend the program with such capabilities.

**Exercise 229.** 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 109](#) using lists of [Transition.v2](#)s; ignore errors and final states.

Modify the design of `simulate` so that it deals with keystrokes in the appropriate manner now. Follow the design recipe, starting with the adaptation of the data examples.

Use the revised program to simulate a run of the FSM from [exercise 109](#) on the following sequence of keystrokes: "a", "b", "b", "c", and "d".

Finite state machines do come with initial and final states. When a program that “runs” an [FSM](#) reaches a final state, it should stop. The final exercise revises the data representation of [FSMs](#) one more time to introduce these ideas.

**Exercise 230.** Consider the following data representation for [FSMs](#):

```
(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 KeyEvent FSM-State)
```

Represent the [FSM](#) from [exercise 109](#) in this context.

Design the function `fsm-simulate`, which accepts an [FSM.v2](#) and runs it on a player’s keystrokes. If the sequence of keystrokes forces the [FSM.v2](#) to reach a final state, `fsm-simulate` stops. **Hint** The function uses the `initial` field of the given `fsm` structure to track the current state.

**Note on Iterative Refinement** These last two projects 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 [Iterative Refinement](#). **End**

## 13 Summary

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 prespecified 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** to 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 ensure that the composition works after you have spent time on the separate programs, you need to formulate your “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 had 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, and the like.

[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.

## Intermezzo 2: Quote, Unquote

Lists play an important role in our book as well as in Racket, the basis of our teaching languages. For the design of programs, it is critical to understand how lists are constructed from first principles; it informs the creation of our programs. Routine work with lists calls for a compact notation, however, like the `list` function introduced in [The list Function](#).

Be sure to set your language level to BSL+ or up.

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 form of data that is intimately tied to quotation. While this introduction is informal and uses simplistic examples, the rest of the book illustrates the power of the idea with near-realistic variants. Come back to this intermezzo if any of these examples cause you 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 1: Beginning Student Language](#) 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` expression 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)
(list 1 2 3)
> '("a" "b" "c")
(list "a" "b" "c")
> '#(true "hello world" 42)
(list #true "hello world" 42)
```

As you can see, the use of `'` creates the promised lists. In case you forgot what `(list 1 2 3)` means, reread [The list Function](#); it explains that this list is short for `(cons 1 (cons 2 (cons 3 '())))`.

So far `quote` looks like a small improvement over `list`, but look:

```
> '((("a" 1)
 ("b" 2)
 ("d" 4))
 (list (list "a" 1) (list "b" 2) (list "d" 4)))
```

With `'` we can construct lists as well as nested lists.

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:

`(list '("a" 1) '3)` is short for `(list (list '"a" '1) 3)`

We leave it to you to wrap up this example.

**Exercise 231.** Eliminate `quote` in favor of `list` from these expressions:

- `'(1 "a" 2 #false 3 "c")`
- `'()`
- and this table-like shape:

```
'(("alan" 1000)
 ("barb" 2000)
 ("carl" 1500))
```

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 didn't do so 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's 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)
(list 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 name 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 `(+ 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](#), which, like [quote](#), is just a keyword for a compound sentence: `(quasiquote (1 2 3))`. And, like [quote](#), [quasiquote](#) comes with a short-hand, 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)
(list 1 2 3)
> `("a" "b" "c")
(list "a" "b" "c")
> `(#true "hello world" 42)
(list #true "hello world" 42)
```

The best part about ``` is that you can also use it to *unquote*, that is, you can demand an escape back to the programming language proper inside of a *quasiquoted* list. Let's illustrate the idea with the above examples:

```
> `(40 41 ,x 43 44)
(list 40 41 42 43 44)
> `(1 ,(+ 1 1) 3)
(list 1 2 3)
```

As above, the first interaction assumes a definitions area that contains `(define x 42)`. The best way to understand this syntax is to see it with actual keywords instead of ``` and `,` short-hands:

```
(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 there is a simplistic example in [figure 83](#). As you can immediately see, this function consumes two strings and produces a deeply nested list—a data representation of a web page.

```
; String String -> ... deeply nested list ...
; produces 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.")))))))
```

Figure 83: A simplistic HTML generator

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.

| 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>&lt;html&gt;   &lt;head&gt;     &lt;title&gt;       Hello World     &lt;/title&gt;     &lt;meta       http-equiv="content-type"       content="text-html" /&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;h1&gt;       Hello World     &lt;/h1&gt;     &lt;p&gt;       I,       Matthias,       made this page.     &lt;/p&gt;   &lt;/body&gt; &lt;/html&gt;</pre> |

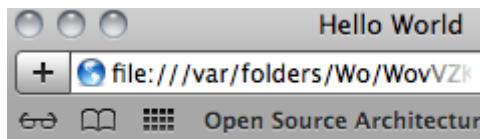
Figure 84: A data representation based on nested lists

To understand how the function works, we experiment in DrRacket's interactions 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: see the left column in figure 84. 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 `show-in-browser` from the `web-io.rkt` library to display the result in a web browser.



## Hello World

I, Matthias, made this page.

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 232.** 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 is short for (list 'tr
 ,(make-row (make-row
 '(3 4 5))) (list 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. If `make-row` translates the given list of numbers into a list of strings, then the result is

```
(list 'tr (list "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 "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 '(3 4 5)))
```

Convince yourself that the result is `(list 'tr "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 '(3 4 5)))
```

translates into

```
| (cons 'tr (make-row '(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 with four cells each:

```
'(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 l
(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 interactions area:

```
> (make-cell 2)
(list 'td "2")
> (make-row '(1 2))
(list (list 'td "1") (list 'td "2"))
```

These interactions show the creation of lists that represent a cell and a row.

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))
(list 'table (list (list 'border "1")) '....)

```

This application of `make-table` suggests another reason why people write programs to create web pages rather than make them by hand.

The dots are **not** part of the output.

**Exercise 233.** Develop alternatives to the following expressions that use only `list` and produce the same values:

- `(0 ,@(1 2 3) 4)
- this table-like shape:

```

`((("alan" ,(* 2 500))
 ("barb" 2000)
 (,@'("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.

Use `check-expect` to check your work.

**Exercise 234.** 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 resulting web page in a browser, you see something like the screen shot in [figure 85](#).

The screenshot shows a web browser window with a title bar "Top 3 Songs from the 80s, 90s, 00s". The address bar shows the URL "file:///var/folders/Wo/WovVZK". Below the address bar are standard browser controls: back, forward, search, and tabs labeled "Open Source Architecture" and "Learn Has". The main content area displays a table with three rows:

|   |                                      |
|---|--------------------------------------|
| 1 | Asia: Heat of the Moment             |
| 2 | U2: One                              |
| 3 | The White Stripes: Seven Nation Army |

Figure 85: A web page generated with BSL+

**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.

### 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.

---

## 14 Similarities Everywhere

If you solved (some of) the exercises in [Arbitrarily Large Data](#), you know that many 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 the next. 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.

In DrRacket, choose “Intermediate Student” from the “How to Design Programs” submenu in the “Language” menu.

Regardless, these mechanisms share the basic characteristics spelled out in this chapter, and thus the design ideas explained here apply in other contexts, too.

## 14.1 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 ; Los -> Boolean
; does l contain "dog" ; does l contain "cat"
(define (contains-dog? l) (define (contains-cat? l)
 (cond (cond
 [(empty? l) #false] [(empty? l) #false]
 [else [else
 (or (or
 (string=? (first l) (string=? (first l)
 "dog") "cat")
 (contains-dog? (contains-cat?
 (rest l))))]))]) (rest l))))))
```

Figure 86: Two similar functions

Consider the two functions in figure 86, 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 87 does just that, and you should briefly compare it with figure 86 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 ; Los -> Boolean
; does l contain "dog" ; does l contain "cat"
(define (contains-dog? l) (define (contains-cat? l)
 (contains? "dog" l)) (contains? "cat" l))
```

Figure 87: Two similar functions, revisited

What you have just witnessed is called *abstraction* or, more precisely, *functional abstraction*. Abstracting different versions of functions is one way to eliminate similarities from programs, and as you will see, removing similarities simplifies keeping a program intact over a long period.

Computer scientists borrow the term “abstract” from mathematics. There, “6” is an abstract concept because it represents all ways of enumerating six things. In contrast, “6 inches” or “6 eggs” are concrete uses.

**Exercise 235.** Use the `contains?` function to define functions that search for “`atom`”, “`basic`”, and “`zoo`”, respectively.

**Exercise 236.** Create test suites for the following two functions:

```
; Lon -> Lon ; Lon -> Lon
; adds 1 to each item on l ; adds 5 to each item on l
(define (add1* l) (define (plus5 l)
 (cond (cond
 [(empty? l) '()]
 [else
 (cons
 (add1 (first l))
 (add1* (rest l))))])) (cond
 [(empty? l) '()]
 [else
 (cons
 (+ (first l) 5)
 (plus5 (rest l))))]))
```

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.

## 14.2 Different Similarities

Abstraction looks easy in the case of the `contains-dog?` and `contains-cat?` functions. 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 88](#). 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.

```

; Lon Number -> Lon ; Lon Number -> Lon
; select those numbers on l ; select those numbers on l
; that are below t ; that are above t
(define (small l t)
 (cond
 [(empty? l) '()]
 [else
 (cond
 [(< (first l) t)
 (cons (first l)
 (small
 (rest l) t)))]
 [else
 (small
 (rest l) t))])))

(define (large l t)
 (cond
 [(empty? l) '()]
 [else
 (cond
 [(> (first l) t)
 (cons (first l)
 (large
 (rest l) t)))]
 [else
 (large
 (rest l) t))]))))

```

Figure 88: 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 `<`, and the right one `>`. Other than that, the two functions look identical, not counting the function name.

Let's 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 fuss, 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 have encountered the differential operator and the indefinite integral. Both of those are functions that consume and produce functions. But we do not assume that you have taken a calculus course.

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 result as `(large l t)`, which means that you can define the two original functions like this:

```
; Lon Number -> Lon ; Lon Number -> Lon
(define (small-1 l t) (define (large-1 l t)
 (extract < 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 pre-defined 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 237.** Evaluate `(squared>? 3 10)` and `(squared>? 4 10)` in DrRacket. How about `(squared>? 5 10)`?

So far you have seen that abstracted function definitions can be more useful than the original functions. 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

These benefits of abstraction are available at all levels of programming: word documents, spreadsheets, small apps, and large industrial projects. Creating abstractions for the latter drives research on programming languages and software engineering.

any extra effort. The following exercises indicate how these single-point-of-control improvements work.

```
; Nelon -> Number ; Nelon -> Number
; determines the smallest ; determines the largest
; number on l ; number on l
(define (inf l) (define (sup l)
 (cond (cond
 [(empty? (rest l)) [(empty? (rest l))]
 (first l)] (first l)]
 [else [else
 (if (< (first l) (if (> (first l)
 (inf (rest l)))) (sup (rest l)))
 (first l) (first l))
 (inf (rest l))))]) (sup (rest l))))])
```

Figure 89: Finding the inf and sup in a list of numbers

**Exercise 238.** Abstract the two functions in figure 89 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, and 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 another answer to these questions, see [Local Definitions](#).

## 14.3 Similarities in Data Definitions

Now take a close look at the following two data definitions:

```
; An Lon (List-of-numbers) ; An Los (List-of-String)
; is one of: ; 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:

- When we write `[List-of Number]`, we are saying that `ITEM` represents all numbers so it is just another name for `List-of-numbers`;
- Similarly, `[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 a structure:
; (make-ir String Number)
```

then `[List-of IR]` would be a name for the lists of inventory records.

By convention, we use names with all capital letters for parameters of data definitions, while the arguments are spelled as needed.

Our way to validate that these short-hands really mean what we say they mean is to substitute the actual name of a data definition, for example, `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's 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 use this structure type:

```
; A Pair-boolean-string is a structure:
; (make-point Boolean String)
```

```
; A Pair-number-image is a structure:
; (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 239.** A list of two items is another frequently used form of data in ISL programming. Here is a data definition with two parameters:

```
; A [List X Y] is a structure:
; (cons X (cons Y '()))
```

Instantiate this definition to describe the following classes of data:

- 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 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. The 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]
```

This inner part combines `Boolean` and `Image` in a `point`. By implication,

```
; [List-of [CP Boolean Image]]
```

is a list of points 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`.

**Exercise 240.** Here are two strange but similar data definitions:

```

; An LStr is one of: ; An LNum is one of:
; - String ; - Number
; - (make-layer LStr) ; - (make-layer LNum)

```

Both data definitions rely on 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 241.** Compare the definitions for `NEList-of-temperatures` and `NEList-of-Booleans`. Then formulate an abstract data definition *NEList-of*.

**Exercise 242.** Here is one more parametric data definition:

```

; A [Maybe X] is one of:
; - #false
; - X

```

Interpret these data definitions: `[Maybe String]`, `[Maybe [List-of String]]`, and `[List-of [Maybe String]]`.

What does the following function signature mean:

```

; String [List-of String] -> [Maybe [List-of String]]
; returns the remainder of los starting with s
; #false otherwise
(check-expect (occurs "a" (list "b" "a" "d" "e"))
 (list "d" "e"))
(check-expect (occurs "a" (list "b" "c" "d")) #f)
(define (occurs s los)
 los)

```

Work through the remaining steps of the design recipe.

## 14.4 Functions Are Values

The functions in this part stretch our understanding of program evaluation. It is easy to understand how functions consume more than numbers, say strings or images. Structures and lists are a bit of a stretch, but they are finite “things” in the end. Function-consuming functions, however, are strange. Indeed, the very idea violates the first intermezzo in two ways: (1) the names of primitives and functions are used as arguments in applications, and (2) parameters are used in the function 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.

The changes to the grammar seem to demand changes to the evaluation rules, but all that changes is the set of values. Specifically, to accommodate functions as arguments of functions, the simplest change is to say that functions and primitive operations **are** values.

**Exercise 243.** 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 (not) values.

**Exercise 244.** Argue why the following sentences are now legal:

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 245.** Develop the function=? function. Given two functions from numbers to numbers, the function determines whether the two 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? 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.

---

## 14.5 Computing with Functions

The switch from BSL+ to ISL allows the use of functions as arguments and the use of names in the first position of an application. DrRacket deals with names in these positions like anywhere else, but naturally, it expects a function as a result. Surprisingly, a simple adaptation of the laws of algebra suffices to evaluate programs in ISL.

Let's see how this works for extract from [Different Similarities](#). Obviously,

```
| (extract < '() 5) == '()
```

holds. We can use the law of substitution from [Intermezzo 1: Beginning Student Language](#) 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 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)])])
== '()
```

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 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 from above, meaning we can apply the law of substituting equals for equals.

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. It deals with the case that `extract` eliminates the first item on the list if it is not below the threshold.

**Exercise 246.** Check step 1 of the last calculation

```

(extract < (cons 6 (cons 4 '()) 5)
==

(extract < (cons 4 '()) 5)
```

using DrRacket's stepper.

**Exercise 247.** Evaluate `(extract < (cons 8 (cons 4 '()) 5))` with DrRacket's stepper.

**Exercise 248.** Evaluate `(squared? 3 10)` and `(squared? 4 10)` in DrRacket's stepper.

Consider this interaction:

```
> (extract squared? (list 3 4 5) 10)
(list 4 5)
```

Here are some steps as the stepper would show them:

```
(extract squared? (list 3 4 5) 10) (1)
== (2)
(cond
 [(empty? (list 3 4 5)) '()]
 [else
 (cond
 [(squared? (first (list 3 4 5)) 10)
 (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)
 (cons (first (list 3 4 5))
 (extract squared?
 (rest (list 3 4 5))
 10))]

 [else (extract squared?
 (rest (list 3 4 5))
 10))])]
```

Use the stepper to confirm the step from lines (1) to (2). Continue the stepping to fill in the gaps between steps (2) and (3). Explain each step as the use of a law.

**Exercise 249.** 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) '())))
```

The stepper displays functions as `lambda` expressions; see [Nameless Functions](#).

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 encounter other programming languages, you will see that their abstraction mechanisms also require the introduction of parameters, though they may not be function parameters.

## 15.1 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 given speed and 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
; temperatures to Fahrenheit ; toys from an inventory
(define (cf* l)
 (cond
 [(empty? l) '()]
 [else
 (cons
 (C2F (first l))
 (cf* (rest l))))])) (define (names i)
 (cond
 [(empty? i) '()]
 [else
 (cons
 (IR-name (first i))
 (names (rest i))))])) (define-struct IR
 [name price])
; An IR is a structure:
; (make-IR String Number)
; An Inventory is one of:
; - '()
; - (cons IR Inventory)

; Number -> Number
; converts one Celsius
; temperature to Fahrenheit
(define (C2F c)
 (+ (* 9/5 c) 32))
```

Figure 90: A pair of similar functions

```
(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))))]) (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))))])
```

```
(map1 (rest k) g))))
```

```
(map1 (rest k) g))))
```

Figure 91: The same two similar functions, abstracted

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 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 *analogous* places, compare them and mark the differences. If the two definitions differ in more than one place, connect the corresponding differences with a line.

The recipe requires a substantial modification for abstracting over non-values.

[Figure 90](#) 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 functions 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*; see [figure 91](#). 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; see bottom of [figure 91](#). 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 with 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, `(f-from-abstract v)` should produce the same answer as `(f-original v)` for every proper value `v`. In particular, it must hold for all values that your tests for `f-original` use. So reformulate and rerun those tests for `f-from-abstract` and make sure they succeed.

Let's return to our running example:

```
| ; List-of-numbers -> List-of-numbers
| (define (cf*-from-map1 l) (map1 l C2F))

| ; Inventory -> List-of-strings
| (define (names-from-map1 i) (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. A new abstraction needs **a signature**, because, as [Using Abstractions](#) explains, the reuse of abstractions starts with their signatures. Finding useful signatures is a serious problem. For now we use the running example to illustrate the difficulty; [Similarities in Signatures](#) resolves the issue.

Consider the problem of `map1`'s signature. On the one hand, if you view `map1` as an abstraction of `cf*`, you might think it is

```
| ; List-of-numbers [Number -> Number] -> List-of-numbers
```

that is, the original signature extended with one part for functions:

```
| ; [Number -> Number]
```

Since the additional parameter for `map1` is a function, the use of a function signature to describe it should not 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](#).

```
; Number -> [List-of Number] ; Number -> [List-of Number]
; tabulates sin between n ; tabulates sqrt between n
; and 0 (incl.) in a list ; and 0 (incl.) in a list
(define (tab-sin n) (define (tab-sqrt n)
 (cond (cond
 [(= n 0) (list (sin 0))] [= n 0) (list (sqrt 0))]
 [else [else
 (cons (cons
 (sin n) (sqrt n)
 (tab-sin (sub1 n))))]) (tab-sqrt (sub1 n))))])
```

Figure 92: The similar functions for exercise 250

```
; [List-of Number] -> Number ; [List-of Number] -> Number
; computes the sum of ; computes the product of
; the numbers on l ; the numbers on l
(define (sum l) (define (product l)
 (cond (cond
 [(empty? l) 0] [(empty? l) 1]
 [else
```

```
(+ (first l) (* (first l)
 (sum (rest l))))) (product (rest l))))
```

Figure 93: The similar functions for exercise 251

**Exercise 250.** Design `tabulate`, which is the abstraction of the two functions in figure 92. When `tabulate` is properly designed, use it to define a tabulation function for `sqr` and `tan`.

**Exercise 251.** Design `fold1`, which is the abstraction of the two functions in figure 93.

**Exercise 252.** Design `fold2`, which is the abstraction of the two functions in figure 94. Compare this exercise with exercise 251. 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.

```
; [List-of Number] -> Number ; [List-of Posn] -> Image
(define (product l) (define (image* l)
 (cond
 [(empty? l) 1]
 [else
 (* (first l)
 (product
 (rest 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"))
```

Figure 94: The similar functions for exercise 252

Lastly, when you are dealing with data definitions, the abstraction process proceeds in an analogous manner. The extra parameters to data definitions stand 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.

## 15.2 Similarities in Signatures

As it turns out, a function's signature is key to its reuse. Hence, you must learn to formulate signatures that describe 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

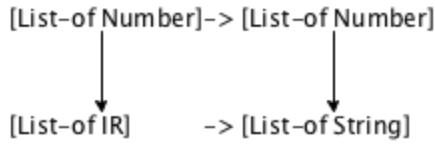
```
; Number Boolean -> String
(define (f n b) "hello world")
```

your first line describes an entire class of data, and your second one states that `f` belongs to that class. To be precise, the signature describes the class of **all functions** that consume a `Number` and a `Boolean` and yield 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 signatures 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's start with the signatures of `cf*` 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*:`

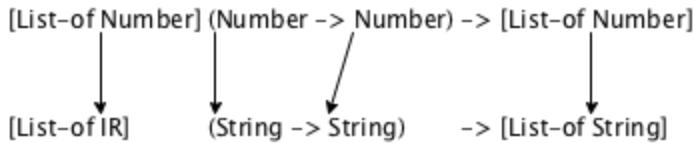
```
; [List-of Number] -> [List-of Number]
```

If you choose **IR** and **String**, 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 markups 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**:

; [X Y] [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** is lists that contain items from **Y**.

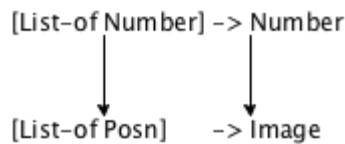
Abstracting over signatures takes practice. Here is a second pair:

; [List-of Number] -> Number  
; [List-of Posn] -> Image

They are the signatures for **product** and **image\*** in [exercise 252](#). 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; and
- both argument descriptions employ the **List-of** construction.

In contrast to the first example, here one signature refers to **Number** twice while the second one refers to **Posns** and **Images** in analogous positions. A structural comparison shows that the first occurrence of **Number** corresponds to **Posn** and the second one to **Image**:



To make progress on a signature for the abstraction of the two functions in [exercise 252](#), let's take the first two steps of the design recipe:

```

(define (pr* l bs jn) (define (im* l bs jn)
 (cond (cond
 [(empty? l) bs] [(empty? l) bs]
 [else [else
 (jn (first l) (jn (first l)
 (pr* (rest l) (im* (rest l)
 bs 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 because the function for the second `cond` line is `+`. Similarly, the signature for `im*` is

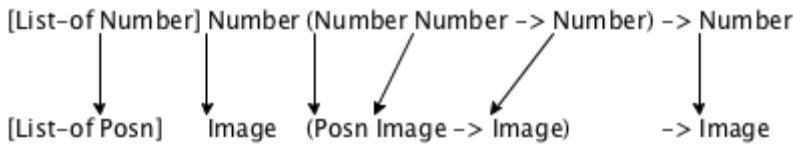
```

; [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`, and
2. 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 from [exercise 252](#):

```
| ; [X Y] [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 a 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. 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` must 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 253.** Each of these signatures describes a class 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 254.** 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 255.** 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 `map1` function above.

## 15.3 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 only one of the functions would run faster.

We therefore formulate this guideline:

*Form an abstraction instead of copying and modifying any code.*

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.

## 15.4 Abstractions from Templates

The first two chapters of this part present many functions based on 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, and you should do so automatically; 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:

```
; [X Y] [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:

```
; [List-of Number] -> Number ; [List-of Number] -> Number
(define (sum lon) (define (product lon)
 (reduce lon 0 +)) (reduce lon 1 *))
```

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`.

---

## 16 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 reusing 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 reusing abstract functions.

```
; [X] 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] [X -> Boolean] [List-of X] -> [List-of X]
; produces a list from those items on lx for which p holds
(define (filter p lx) ...)

; [X] [List-of X] [X X -> Boolean] -> [List-of X]
; produces a version of lx that is sorted according to cmp
(define (sort lx cmp) ...)

; [X Y] [X -> Y] [List-of X] -> [List-of Y]
; constructs a list by applying f to each item on lx
; (map f (list x-1 ... x-n)) == (list (f x-1) ... (f x-n))
(define (map f lx) ...)

; [X] [X -> Boolean] [List-of X] -> Boolean
; determines whether p holds for every item on lx
; (andmap p (list x-1 ... x-n)) == (and (p x-1) ... (p x-n))
(define (andmap p lx) ...)

; [X] [X -> Boolean] [List-of X] -> Boolean
; determines whether p holds for at least one item on lx
; (ormap p (list x-1 ... x-n)) == (or (p x-1) ... (p x-n))
(define (ormap p lx) ...)
```

Figure 95: ISL's abstract functions for list processing (1)

## 16.1 Existing Abstractions

ISL provides a number of abstract functions for processing natural numbers and lists.

Figure 95 collects the header material for the most important ones. The first one processes natural numbers and builds lists:

```
> (build-list 3 add1)
(list 1 2 3)
```

The next three process lists and produce lists:

```
> (filter odd? (list 1 2 3 4 5))
(list 1 3 5)
> (sort (list 3 2 1 4 5) >)
(list 5 4 3 2 1)
> (map add1 (list 1 2 2 3 3 3))
(list 2 3 3 4 4 4)
```

In contrast, `andmap` and `ormap` reduce lists to a Boolean:

```
> (andmap odd? (list 1 2 3 4 5))
#false
> (ormap odd? (list 1 2 3 4 5))
#true
```

Hence, this kind of computation is called a *reduction*.

```
; [X Y] [X Y -> Y] Y [List-of X] -> Y
; applies f from right to left to each item in lx and b
; (foldr f b (list x-1 ... x-n)) == (f x-1 ... (f x-n b))
(define (foldr f b lx) ...)

(foldr + 0 '(1 2 3 4 5))
== (+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0))))))
== (+ 1 (+ 2 (+ 3 (+ 4 5))))
== (+ 1 (+ 2 (+ 3 9)))
== (+ 1 (+ 2 12))
== (+ 1 14)

; [X Y] [X Y -> Y] Y [List-of X] -> Y
; applies f from left to right to each item in lx and b
; (foldl f b (list x-1 ... x-n)) == (f x-n ... (f x-1 b))
(define (foldl f b lx) ...)

(foldl + 0 '(1 2 3 4 5))
== (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))
== (+ 5 (+ 4 (+ 3 (+ 2 1))))
== (+ 5 (+ 4 (+ 3 3)))
== (+ 5 (+ 4 6))
== (+ 5 10)
```

Figure 96: ISL's abstract functions for list processing (2)

The two functions in figure 96, `foldr` and `foldl`, are extremely powerful. Both `reduce` lists to values. The sample computations explain the abstract examples in the headers of `foldr` and `foldl` via an application of the functions to `+`, `0`, and a short list. As you can see, `foldr` processes the list values from right to left and `foldl` from left to right. While for some functions the direction makes no difference, this isn't true in general.

Mathematics calls functions *associative* if the order makes no difference. ISL's `=` is associative on integers but not on inexact numbers. See below.

**Exercise 256.** Explain the following abstract function:

```
; [X] [X -> Number] [NELList-of X] -> X
; finds the (first) item in lx that maximizes f
; if (argmax f (list x-1 ... x-n)) == x-i,
; then (>= (f x-i) (f x-1)), (>= (f x-i) (f x-2)), ...
(define (argmax f lx) ...)
```

Use it on concrete examples in ISL. Can you articulate an analogous purpose statement for `argmin`?

```
(define-struct address [first-name last-name street])
; An Addr is a structure:
; (make-address String String String)
; interpretation associates an address with a person's name

; [List-of Addr] -> String
; creates a string from 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
; appends two strings, prefixes with " "
(define (string-append-with-space s t)
 (string-append " " s t))

(define ex0
 (list (make-address "Robert" "Findler" "South")
 (make-address "Matthew" "Flatt" "Canyon")
 (make-address "Shriram" "Krishna" "Yellow")))

(check-expect (listing ex0) " Matthew Robert Shriram ")
```

Figure 97: Creating a program with abstractions

Figure 97 illustrates the power of composing the functions from figures 95 and 96. Its main function is `listing`. The purpose is to create a string from a list of addresses. Its purpose statement suggests three tasks and thus the design of three functions:

1. one that extracts the first names from the given list of `Addr`;
2. one that sorts these names in alphabetical order; and
3. one 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 is possible to design a single function that achieves the same goal. Take a close look at the innermost expression of `listing` in figure 97:

```
| (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 how the combination of `foldr` and `string-append-with-space` works.

**Exercise 257.** 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] [X Y -> Y] Y [List-of X] -> Y
; f*oldl works just like foldl
(check-expect (f*oldl cons '() '(a b c))
 (foldl cons '() '(a b c)))
(check-expect (f*oldl / 1 '(6 3 2))
 (foldl / 1 '(6 3 2)))
(define (f*oldl f e l)
 (foldr f e (reverse l)))
```

Design `build-l*st`, which works just like `build-list`. **Hint** Recall the `add-at-end` function from [exercise 193](#). **Note on Design Accumulators** covers the concepts needed to design these functions from scratch.

---

## 16.2 Local Definitions

Let's take a second look at [figure 97](#). The `string-append-with-space` function clearly plays a subordinate role and has no use outside of this narrow context. Furthermore, the organization of the function body does not reflect the three tasks identified above.

Almost all programming languages support some way for stating these kinds of relationships as a part of a program. The idea is called a *local definition*, also called a *private definition*. In ISL, `local` expressions introduce locally defined functions, variables, and structure types.

This section introduces the mechanics of `local`. In general, a `local` expression has this shape:

```
(local (def ...)
 ; - IN -
 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 and it becomes the result of the `local` expression. It is often helpful to separate the `local` *defs* from the body-expression with a comment; as indicated, we may use – IN – because the word suggests that the definitions are available in a certain expression.

```
; [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 (; 1. extract names
 (define names (map address-first-name l))
 ; 2. sort the names
 (define sorted (sort names string<?)))
 ; 3. append them, add spaces
 ; String String -> String
 ; appends two strings, prefix with " "
 (define (helper s t)
 (string-append " " s t))
 (define concat+spaces
 (foldr helper " " sorted)))
 concat+spaces))
```

Figure 98: Organizing a function with `local`

Figure 98 shows a revision of figure 97 using `local`. 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. The sequence of local definitions looks exactly like a sequence in DrRacket’s definitions area.

In this example, the sequence of definitions consists of four pieces: three constant definitions and a single function definition. Each constant definition represents one of the three planning tasks. The function definition is a renamed version of `string-append-with-space`; it is used with `foldr` to implement the third task.

The body of `local` is just the name of the third task.

Since the names are visible only within the `local` expression, shortening the name is fine.

The visually most appealing difference concerns the overall organization. It clearly brings across that the function achieves three tasks and in which order. As a matter of fact, this example demonstrates a general principle of readability:

*Use `local` to reformulate deeply nested expressions. Use well-chosen names to express what the expressions compute.*

Future readers appreciate it because they can comprehend the code by looking at just the names and the body of the `local` expression.

**Note on Organization** A `local` expression is really just an expression. It may show up wherever a regular expression shows up. Hence it is possible to indicate precisely where an auxiliary function is needed. Consider this reorganization of the `local` expression of listing.v2:

```
... (local ((define names ...)
 (define sorted ...))
 (define concat+spaces
 (local (; String String -> String
 (define (helper s t)
 (string-append " " s t)))
 (foldr helper " " sorted))))
 concat+spaces) ...
```

It consists of exactly three definitions, suggesting it takes three computation steps. The third definition consists of a `local` expression on the right-hand side, which expresses that `helper` is really just needed for the third step.

Whether you want to express relationships among the pieces of a program with such precision depends on two constraints: the programming language and how long the code is expected to live. Some languages cannot even express the idea that `helper` is useful for the third step only. Then again, you need to balance the time it takes to create the program and the expectation that you or someone needs to revisit it and comprehend the code again. The preference of the Racket team is to err on the side of future developers because the team members know that no program is ever finished and all programs will need fixing. **End**

```
; [List-of Number] [Number Number -> Boolean]
; -> [List-of Number]
; produces a version of alon, sorted according to cmp
(define (sort-cmp alon0 cmp)
 (local (; [List-of Number] -> [List-of Number]
 ; produces the sorted version of alon
 (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 99: Organizing interconnected function definitions with `local`

Figure 99 presents a second example. The organization of this function definition informs the reader that `sort-cmp` calls on two auxiliary functions: `isort` and `insert`. By locality, it

becomes obvious that the adjective “sorted” in the purpose statement of `insert` refers to `isort`. In other words, `insert` is useful in this context only; a programmer should not try to use it elsewhere, out of context. While this constraint is already important in the original definition of the `sort-cmp` function, a `local` expression expresses it as part of the program.

Another important aspect of this reorganization of `sort-cmp`’s definition concerns the visibility of `cmp`, the second function parameter. The locally defined functions can refer to `cmp` because it is defined in the context of the definitions. By **not** passing around `cmp` from `isort` to `insert` (or back), the reader can immediately infer that `cmp` remains the same throughout the sorting process.

**Exercise 258.** Use a `local` expression to organize the functions for drawing a polygon in [figure 73](#). If a globally defined function is widely useful, do not make it local.

**Exercise 259.** 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.v2 l)
 (cond
 [(empty? (rest l)) (first l)]
 [else
 (local ((define smallest-in-rest (inf.v2 (rest l))))
 (if (< (first l) smallest-in-rest)
 (first l)
 smallest-in-rest))]))
```

Figure 100: Using `local` may improve performance

Our final example of `local`’s usefulness concerns performance. Consider the definition of `inf` in [figure 89](#). It contains two copies of

```
(inf (rest l))
```

which is the natural recursion in the second `cond` line. Depending on the outcome of the question, the expression is evaluated twice. Using `local` to name this expression yields an improvement to the function’s readability as well as to its performance.

[Figure 100](#) displays the revised version. Here the `local` expression shows up in the middle of a `cond` expression. It defines a constant whose value is the result of a natural recursion. Now recall that the evaluation of a `local` expression evaluates the definitions once before proceeding to the body, meaning `(inf (rest l))` is evaluated once while the body of the `local` expression refers to the result twice. Thus, `local` saves the re-evaluation of `(inf (rest l))` at each stage in the computation.

**Exercise 260.** Confirm the insight about the performance of `inf.v2` by repeating the performance experiment of [exercise 238](#).

```
; Inventory -> Inventory
; creates an Inventory from an-inv for all
; those items that cost less than a dollar
```

```

(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))]))])

```

Figure 101: A function on inventories, see [exercise 261](#)

**Exercise 261.** Consider the function definition in [figure 101](#). Both clauses in the nested `cond` expression extract the first item from `an-inv` and both compute `(extract1 (rest an-inv))`. Use `local` to name this expression. Does this help increase the speed at which the function computes its result? Significantly? A little bit? Not at all?

## 16.3 Local Definitions 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 keystrokes. While the data analysis leads in a natural manner to the data definitions in [figure 82](#), 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
; matches 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
 (define (find-next-state s key-event)
 (find fsm s)))
 (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 current state in 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)))]))

```

```
(find (rest transitions) current))))])
```

Figure 102: Power from local function definitions

Figure 102 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 keystroke. 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 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, as well as a future reader's ability to recognize the design insight of the original creator.

**Exercise 262.** Design the function `identityM`, which creates diagonal squares of `0`s and `1`s:

Linear algebra calls these squares *identity* matrices.

```
> (identityM 1)
(list (list 1))
> (identityM 3)
(list (list 1 0 0) (list 0 1 0) (list 0 0 1))
```

Use the structural design recipe and exploit the power of `local`.

## 16.4 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 s)))
(big-bang A-STATE
 [to-draw state-as-colored-square]
 [on-key find-next-state]))
```

This is the body of `simulate` with all occurrences of `fsm` and `s` replaced by the argument values AN-FSM and A-STATE, respectively.

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 the top level and evaluate the body of the `local` expression next.

Stop! Don't think. Accept the two steps for now.

Let's apply these two steps to our running example, one at a time:

```
==
(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. So here is the result of step 2:

```
==
(define (find-next-state-1 s key-event)
 (find an-fsm a-state))
```

⊕

We use  $\oplus$  to indicate that the step produces two pieces.

```
(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 100](#). 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 `(list 2 1 3)` for `l`.

Since the list 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)
 (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]))
⊕
(define smallest-in-rest-2
 [(< (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).

**Exercise 263.** Use DrRacket's stepper to study the steps of this calculation in detail.

**Exercise 264.** Use DrRacket's stepper to calculate out how it evaluates

```
(sup (list 2 1 3))
```

where `sup` is the function from [figure 89](#) equipped with `local`.

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 265.** Use DrRacket's stepper to fill in any gaps above.

**Exercise 266.** Use DrRacket's stepper to find out how ISL evaluates

```
((local ((define (f x) (+ x 3))
 (define (g x) (* x 4)))
 (if (odd? (f (g 1)))
 f
 g))
 2)
```

to 5.

## 16.5 Using Abstractions, by Example

Now that you understand `local`, you can easily use the abstractions from [figures 95](#) and [96](#). Let's look at examples, starting with this one:

**Sample Problem** Design `add-3-to-all`. The function consumes a list of `Posns` and adds 3 to the x-coordinates of each.

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 3 1) (make-posn 0 0)))
 (list (make-posn 6 1) (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 a list-processing function. The question is whether it is like any of the functions in [figures 95](#) and [96](#). The signature tells us that `add-3-to-all` is a list-processing function that consumes and produces a list. In the two figures, 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 signature again:

```
; [X Y] [X -> Y] [List-of X] -> [List-of Y]
```

It tells us that `map` consumes a function from `X` to `Y` and a list of `Xs`. 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` can do the job of add-3-to-all when given the right function from `Posns` to `Posns`. With `local`, we can express this idea as a template for add-3-to-all:

```
(define (add-3-to-all lop)
 (local (; Posn -> Posn
 ; ...
 (define (fp p)
 ... p ...))
 (map fp 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 3 1) (make-posn 0 0)))
==
(map fp (list (make-posn 3 1) (make-posn 0 0)))
==
(list (fp (make-posn 3 1)) (fp (make-posn 0 0))))
```

And that shows how `fp` 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:

```
(define (add-3-to-all lop)
 (local (; Posn -> Posn
 ; adds 3 to the x-coordinate of p
 (define (add-3-to-1 p)
 (make-posn (+ (posn-x p) 3) (posn-y p))))
 (map add-3-to-1 lop)))
```

We chose `add-3-to-1` 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`.

You may now think 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's take a look at a second example a bit more quickly:

**Sample Problem** Design a function that eliminates all `Posns` with y-coordinates larger than 100 from some given list.

The first two steps of the design recipe yield this:

```
; [List-of Posn] -> [List-of Posn]
; eliminates Posns whose y-coordinate is > 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:

```
(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:

```
(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’s 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 the other 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`. It 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 signature differentiates this example from the preceding ones.

The `Boolean` range also gives away a clue with respect to [figures 95](#) and [96](#). 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's apply our standard “trick” of adding a `local` whose body uses the chosen abstraction on some locally defined function and the given list:

```
(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] [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:

```
(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) ; in terms of pixels
```

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`.

---

## 16.6 Designing with Abstractions

The three sample problems from the preceding section suffice for formulating a design recipe:

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  $200 \times 200$  canvas for a given list of `Posns`. The first three steps of the design recipe yields 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 31 MT-SCENE))

(define (dots lop)
 MT-SCENE)
```

Add definitions for the constants so DrRacket can run the code.

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] [X Y -> Y] Y [List-of X] -> Y
; foldl : [X Y] [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 the `foldr` function:

```
(define (dots lop)
 (local (; Posn Image -> Image
```

```
(define (add-one-dot p scene) ...))
(foldr add-one-dot MT-SCENE 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 auxiliary function inside `local`. In most cases, this function consumes relatively simple kinds of data, like those encountered in [Fixed-Size Data](#). You know how to design those in principle. The difference is that now you use not only 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 MT-SCENE 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 that’s the point of the next two sections.

---

## 16.7 Finger Exercises: Abstraction

**Exercise 267.** 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 US\$1.06 per € (on April 13, 2017).

Also use `map` to define `convertFC`, which converts a list of Fahrenheit measurements to a list of Celsius measurements.

Finally, try your hand at `translate`, a function that translates a list of `Posns` into a list of lists of pairs of numbers.

**Exercise 268.** 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 269.** Define `eliminate-expensive`. The function 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 270.** Use `build-list` to define a function 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 1/2 ... 1/n)` for any natural number `n`;
4. creates the list of the first `n` even numbers; and
5. creates a diagonal square of `0`s and `1`s; see [exercise 262](#).

Finally, define `tabulate` from [exercise 250](#) using `build-list`.

**Exercise 271.** 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 that start with the letter `"a"`.

Should you use `ormap` or `andmap` to define a function that ensures that no name on some list exceeds a given width?

**Exercise 272.** 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 273.** The fold functions are so powerful that you can define almost any list processing functions with them. Use `fold` to define `map`.

**Exercise 274.** Use existing abstractions to define the `prefixes` and `suffixes` functions from [exercise 190](#). Ensure that they pass the same tests as the original function.

## 16.8 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 [figures 95](#) and [96](#). 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.

**Exercise 275.** [Real-World Data: Dictionaries](#) deals with relatively simple tasks relating to English dictionaries. The design of two of them just call out for the use of existing abstractions:

You may wish to tackle these exercises again after studying [Nameless Functions](#).

- 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 [Dictionarys](#), 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 74](#).

**Exercise 276.** [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 from the given album that have been played after the date.
- Design `select-albums`. The function consumes an [LTracks](#). It produces 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 77](#) for the services provided by the `2htdp/itunes` library.

**Exercise 277.** [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 278.** [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 that you aren't introducing mistakes.

---

## 17 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`);
- (`foldr cons another-list a-list`) concatenates the items in `a-list` and `another-list` into a single list; and
- (`foldr above empty-image images`) stacks the given images.

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] Number -> Boolean
(define (find l th)
 (local (; IR -> Boolean
 (define (acceptable? ir)
 (<= (ir-price ir) th)))
 (filter acceptable? l)))
```

It finds all items on an inventory list whose price is below `th`. 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 a new concept, nameless functions. 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 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.

```
; [List-of IR] Number -> Boolean
(define (find l th)
 (filter (lambda (ir) (<= (ir-price ir) th)) l))
```

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.

## 17.1 Functions from `lambda`

The syntax of `lambda` 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) (expt 10 x))`, which assumes that the argument is a number and computes the exponent of 10 to the number;
2. `(lambda (n) (string-append "To " n ","))`, which uses a given string to synthesize an address with `string-append`; and
3. `(lambda (ir) (<= (ir-price ir) th))`, which is a function on an IR structure that extracts the price and compares it with `th`.

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

This way of thinking about `lambda` shows one more time why the rule for computing with `local` is complicated.

```
(local ((define some-name (lambda (x) (* 10 x))))
 some-name)
```

This “trick” works, in general, as long as `some-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.

To use a function created from a `lambda` expression, you apply it to the correct number of arguments. It works as expected:

```
> ((lambda (x) (expt 10 x)) 2)
100
> ((lambda (name rst) (string-append name ", " rst))
 "Robby"
 "etc.")
"Robby, etc."
> ((lambda (ir) (<= (ir-price ir) th))
 (make-ir "bear" 10))
#true
```

Note how the second sample function requires two arguments and that the last example assumes a definition for `th` in the definitions window such as this one:

```
| (define th 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**.

The important point is that these nameless functions can be used wherever a function is required, including with the abstractions from [figure 95](#):

```
> (map (lambda (x) (expt 10 x))
 '(1 2 3))
(list 10 100 1000)
> (foldl (lambda (name rst)
 (string-append name ", " rst)))
"etc."
'("Matthew" "Robby")
"Robby, Matthew, etc."
> (filter (lambda (ir) (<= (ir-price ir) th))
 (list (make-ir "bear" 10)
 (make-ir "doll" 33)))
(list (ir ...))
```

Once again, the last example assumes a definition for **th**.

The dots are **not** part of the output.

**Exercise 279.** 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 of DrRacket.

**Exercise 280.** 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 281.** Write down a `lambda` expression that

1. consumes a number and decides whether it is less than `10`;
2. multiplies two given numbers and turns the result into a string;
3. consumes a natural number and returns `0` for evens and `1` for odds;
4. consumes two inventory records and compares them by price; and
5. adds a red dot at a given `Posn` to a given `Image`.

Demonstrate how to use these functions in the interactions area.

## 17.2 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.

It's best to look at some concrete examples:

Alonzo Church, who invented `lambda` in the late 1920s, hoped to create a unifying theory of functions. From his work we know that from a 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](#).

|                                          |                                                                                     |
|------------------------------------------|-------------------------------------------------------------------------------------|
| <code>(define (f x))</code> is short for | <code>(define f</code><br><code>  (lambda (x)</code><br><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 give names to functions for two distinct reasons. On the 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 282.** 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-plain` and the right-hand one to `f-lambda`. Then run

```
(compare (random 100000))
```

a few times to make sure the two functions agree on all kinds of inputs.

If function definitions are just abbreviations for constant definitions, we can replace the function name by its `lambda` expression:

```
(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 expression whose function position is a `lambda` expression.

The point is that ISL+'s grammar differs from ISL's 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 1: Beginning Student Language](#) 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:

```
((Lambda (x-1 ... x-n) f-body) v-1 ... v-n) == f-body
```

Church stated the *beta axiom*  
roughly like this.

```
; 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 example in this chapter:

```
((lambda (x) (* 10 x)) 2)
==
(* 10 2)
 ==
20
```

The second one proceeds in an analogous manner:

```
((lambda (name rst) (string-append name ", " rst))
 "Robby" "etc.")
 ==
(string-append "Robby" ", " "etc.")
 ==
"Robby, etc."
```

Stop! Use your intuition to calculate the third example:

```
((lambda (ir) (<= (ir-price ir) th))
 (make-ir "bear" 10))
```

Assume `th` is larger than or equal to `10`.

**Exercise 283.** Confirm that DrRacket's stepper can deal with `lambda`. Use it to step through the third example and also to determine how DrRacket evaluates the following 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) th))
 (list (make-ir "bear" 10)
 (make-ir "doll" 33)))
```

**Exercise 284.** Step through the evaluation of this expression:

```
((lambda (x) x) (lambda (x) x))
```

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*.

## 17.3 Abstracting with `lambda`

Although it may take you a while to get used to `lambda` notation, you'll 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 (a-posn scene) ...) BACKGROUND lop))
```

From here, you should be able to complete the definition as well as you did from the original template:

```
(define (dots lop)
 (foldr (lambda (a-posn scene)
 (place-image DOT
 (posn-x a-posn)
 (posn-y a-posn)
 scene)))
 BACKGROUND lop))
```

Let's illustrate `lambda` with some more 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]
(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 clearly want `(lambda (p) ...)`. The function then deconstructs `p`, adds 3 to the x-coordinate, and repackages the data into a `Posn`.

- the second one eliminates `Posns` whose y-coordinate is above 100:

```
; [List-of Posn] -> [List-of Posn]
(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
(define (close? lop pt)
 (ormap (lambda (p) (close-to p pt CLOSENESS))
 lop))
```

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 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 285.** 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 US\$1.06 per €.

Also use `map` to define `convertFC`, which converts a list of Fahrenheit measurements to a list of Celsius measurements.

Finally, try your hand at `translate`, a function that translates a list of `Posns` into a list of lists of pairs of numbers.

**Exercise 286.** 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 287.** 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 288.** Use `build-list` and `lambda` to define a function 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 1/2 ... 1/n`) for any natural number  $n$ ;

4. creates the list of the first  $n$  even numbers; and

5. creates a diagonal square of `0s` and `1s`; see [exercise 262](#).

Also define `tabulate` with `lambda`.

**Exercise 289.** 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 that 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 290.** 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 291.** 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`.

---

## 17.4 Specifying with `lambda`

[Figure 99](#) shows a generalized sorting function that consumes a list of values and a comparison function for such values. For convenience, [figure 103](#) reproduces the essence of the definition. 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 on numbers.

```
; [X] [List-of X] [X X -> Boolean] -> [List-of X]
; sorts alon0 according to cmp

(check-expect (sort-cmp '("c" "b") string<?) '("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 X] -> [List-of X]
 ; produces a variant of alon sorted by cmp
 (define (isort alon) ...)

 ; X [List-of X] -> [List-of X]
 ; inserts n into the sorted list of numbers alon
 (define (insert n alon) ...))
(isort alon0)))

```

Figure 103: A general sorting function

Now take a quick look at [exercise 186](#). It asks you to formulate `check-satisfied` tests for `sort>` using the `sorted>?` predicate. 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` analogously.

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 '("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 via `<`.

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))

```

Stop! This is your first function-producing function. Read the definition again. Can you explain this definition in your own words?

Next we need examples. According to our above analysis, `sorted` consumes predicates such as `string<?` and `<`, but clearly, `>`, `<=`, and your own comparison functions should be acceptable, too. At first glance, this suggests test cases of the shape

```
(check-expect (sorted string<?)) ...)
(check-expect (sorted <)) ...)
```

But, `(sorted ...)` produces a function, and, according to [exercise 245](#), it impossible to compare functions.

Hence, to formulate reasonable test cases, we need to apply the result of `(sorted ...)` to appropriate lists. And, based on this insight, the test cases almost formulate themselves; indeed, they can easily be derived from those for `sort-cmp` in [figure 103](#):

```
(check-expect [(sorted string<?)] '("b" "c")] #true)
(check-expect [(sorted <)] '(1 2 3 4 5 6)] #true)
```

**Note** Using square instead of parentheses highlights that the first expression produces a function, which is then applied to arguments.

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) ...))
 ...)))
```

Note how `sorted/l` is defined locally yet refers to `cmp`.

**Exercise 292.** Design the function `sorted?`, which comes with the following signature and purpose statement:

```
; [X X -> Boolean] [NEList-of X] -> Boolean
; determines 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 104](#) 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 redefine sorted to use sorted? from exercise 292? 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 104: A curried predicate for checking the ordering of a list

The sorted function in figure 104 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 186 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.

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 '(3 2) <) '(2 3)]
```

```

 #true)
(check-expect [(sorted-variant-of '(3 2) <) '(3)]
 #false)

```

Like `sorted`, `sorted-variant-of` consumes arguments and produces a function. For the first case, `sorted-variant-of` produces `#true` because the `'(2 3)` is sorted and it contains all numbers in `'(3 2)`. In contrast, the function produces `#false` in the second case because `'(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 292](#), 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:

```

; [List-of X] [List-of X] -> Boolean
; are all items in list k members of list l

(check-expect (contains? '(1 2 3) '(1 4 3)) #false)
(check-expect (contains? '(1 2 3 4) '(1 3)) #true)

(define (contains? l k)
 (andmap (lambda (in-k) (member? in-k l)) k))

```

On the 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 (sort-cmp l <)))
 (cons (- (first sorted) 1) sorted)))

```

It is again easy to expose a flaw in this function with a `check-expect` test that it ought to pass but clearly fails:

```

(check-expect (sort-cmp/worse '(1 2 3)) '(1 2 3))

```

Surprisingly, a `check-satisfied` test based on `sorted-variant-of` succeeds:

```
(check-satisfied (sort-cmp/worse '(1 2 3))
 (sorted-variant-of '(1 2 3) <))
```

Indeed, such a test succeeds for any list of numbers, not just '(1 2 3), because 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 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 reuse `contains?` but with its arguments flipped.

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 (build-list-of-random-numbers 500))

(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 293.** Develop `found?`, a specification for the `find` function:

```
; X [List-of X] -> [Maybe [List-of X]]
; returns 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 294.** Develop `is-index?`, a specification for `index`:

```
; X [List-of X] -> [Maybe N]
; determine the 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 295.** Develop `n-inside-playground?`, a specification of the `random-posns` function below. The function generates a predicate that ensures that the length of the given list is some given count and that all `Posns` in this list are within a `WIDTH` by `HEIGHT` rectangle:

```
; distances in terms of pixels
(define WIDTH 300)
(define HEIGHT 300)

; N -> [List-of Posn]
; generates n random Posns in [0,WIDTH) by [0,HEIGHT)
(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.

---

## 17.5 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 a circle, a rectangle, or a combination of two shapes. See the next part of the book 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:

```
; A Shape is a function:
; [Posn -> Boolean]
; interpretation if s is a shape and p a Posn, (s p)
; produces #true if p is in 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's 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 (mk-point x y)
```

We use “mk” because this function is not an ordinary constructor.

```

(lambda (p)
 (and (= (posn-x p) x) (= (posn-y p) y)))

(define a-sample-shape (mk-point 3 4))

```

Stop again! Convince yourself that the last line creates a data representation of (3,4). Consider using DrRacket's stepper.

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 representation for a point at (x,y)
```

or, more concisely and more appropriately,

```
; represents a point at (x,y)
```

For the examples we want to go with the interpretation of **Shape**. To illustrate, `(mk-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? (mk-point 3 4) (make-posn 3 4))
 #true)
(check-expect (inside? (mk-point 3 4) (make-posn 3 0))
 #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, say circles. 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 than 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 representation for a circle of radius r
; located at (center-x, center-y)
(define (mk-circle center-x center-y r)
 ...)

```

Like `mk-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? (mk-circle 3 4 5) (make-posn 0 0)) #true)
(check-expect
 (inside? (mk-circle 3 4 5) (make-posn 0 9)) #false)
(check-expect
 (inside? (mk-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 remaining examples.

**Exercise 296.** 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 (mk-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 297.** 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

$$\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

that is, 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
; represents a width by height rectangle whose
; upper-left corner is located at (ul-x, ul-y)

(check-expect (inside? (mk-rect 0 0 10 3)
 (make-posn 0 0))
 #true)
(check-expect (inside? (mk-rect 2 3 10 3)
 (make-posn 4 5))
 #true)
; Stop! Formulate a negative test case.

(define (mk-rect ul-x ul-y width height)
 (lambda (p)
 (and (<= ul-x (posn-x p) (+ ul-x width))
 (<= ul-y (posn-y p) (+ ul-y height)))))
```

Its constructor receives four numbers: the coordinates of the upper-left corner, its width, and height. The result is again a `lambda` expression. 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 (mk-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 (mk-circle 3 4 5))
(define rectangle1 (mk-rect 0 3 10 3))
(define union1 (mk-combination circle1 rectangle1))
```

Some points are inside and some outside of this combination:

```
(check-expect (inside? union1 (make-posn 0 0)) #true)
(check-expect (inside? union1 (make-posn 0 9)) #false)
(check-expect (inside? union1 (make-posn -1 3)) #true)
```

Since `(make-posn 0 0)` is inside both, there is no question that it 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 a revision of `mk-combination`:

```
; Shape Shape -> Shape
(define (mk-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 expressions 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 298.** 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 denotes a series of images
```

Here is a data example:

```
; ImageStream
(define (create-rocket-scene height)
```



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

You may recognize this as one of the first pieces of code in the Prologue.

The job of `(my-animate s n)` is to show 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 concepts than this book supplies.

**Exercise 299.** 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`, and so on.

Design the functions `add-element`, which adds an element to a set; `union`, which combines the elements of two sets; and `intersect`, which collects all elements common to two sets.

**Hint** Mathematicians deal with sets as functions that consume a potential element `ed` and produce `#true` only if `ed` belongs to the set.

---

## 18 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 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 abstraction and the abstractions that a language provides. A discerning programmer will learn to distinguish programming languages along these axes.

Beyond abstraction, this third part also introduces the idea that

**functions are values, and they can represent information.**

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.

## Intermezzo 3: 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: 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`, and the like; 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 also the creation of entirely new language constructs, meaning they are not something programmers can usually design and add to their vocabulary.

---

### 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*.

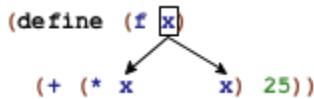
The definitions of `f` and `g` bind two more names: `f` and `g`. Their scope is called *top-level scope* because we think of scopes as nested (see below).

The term *free occurrence* applies to a variable without any binding occurrence. It is a name without definition, that is, 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 `x` at the prompt of the interactions would show that the first two are defined and the last one is not:

```
> f
f
```

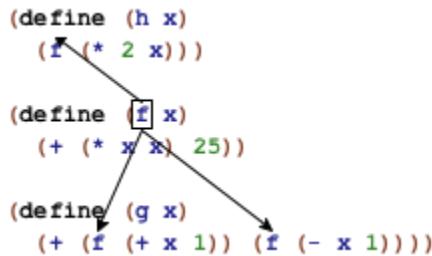
```
> g
g
> x
x:this variable is not defined
```

The description of lexical scope suggests a pictorial representation of f's definition:



DrRacket's "Check Syntax" functionality draws diagrams like these.

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 that

```
(define (f x) (+ (* x x) 25))
(define (g x) (+ (f (+ x 1)) (f (- x 1))))
```

changes only two occurrences of x:

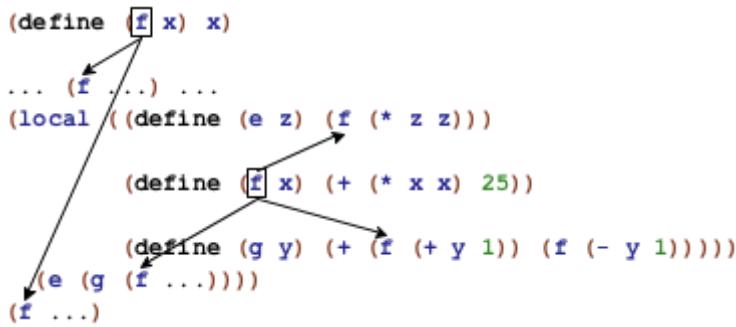
```
(define (f y) (+ (* y y) 25))
(define (g x) (+ (f (+ x 1)) (f (- x 1)))))
```

**Exercise 300.** 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))))
```

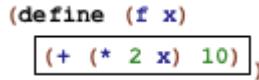
Draw arrows from p1's x parameter to all its bound occurrences. Draw arrows from p1 to all bound occurrences of p1. Check the results with DrRacket's *CHECK SYNTAX* functionality.

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 occurs outside:

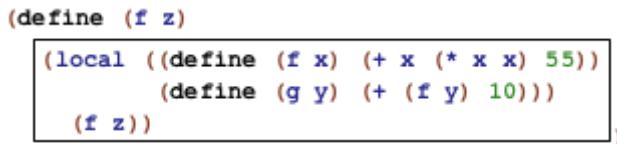


The two occurrences outside of `local` are not bound by the local definition of `f`. As always, the parameters of a function definition, local or not, are only bound in the function's body.

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:

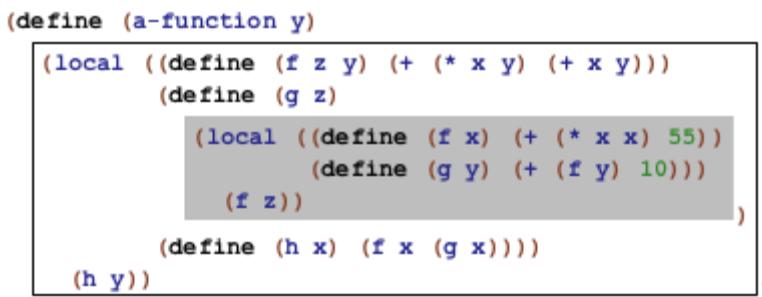


In the case of `local`, the box is drawn around the entire expression:



In this example, the box describes the scope of the definitions of `f` and `g`.

Drawing a box around a scope, we can also easily understand what it means to reuse the name of a function inside a `local` expression:



The gray box describes the scope of the inner definition of `f`; the white box is the scope of the outer definition of `f`. Accordingly, all occurrences of `f` in the gray box refer to the inner `local`; all those in the white box, minus the gray one, refer to the definition in the outer `local`. In other words, the gray box is a *hole* in the scope of the outer definition of `f`.

Holes can also occur in the scope of a parameter definition:

```

(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 thus a hole in the scope of the former.

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.

```

(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)))

```

Figure 105: Drawing lexical scope contours for [exercise 301](#)

**Exercise 301.** Draw a box around the scope of each binding occurrence of `sort` and `alon` in [figure 105](#). Then draw arrows from each occurrence of `sort` to the appropriate binding occurrence. Now repeat the exercise for the variant in [figure 106](#). Do the two functions differ other than in name?

```

(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)))
```

Figure 106: Drawing lexical scope contours for [exercise 301](#) (version 2)

**Exercise 302.** Recall that each occurrence of a variable receives its value from its 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 immediately. 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. That is, if `a-new-name` does not occur in `exp`,

```
(lambda (x-1 ... x-n) exp)
```

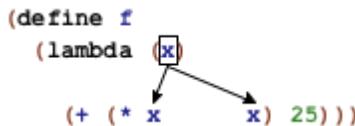
is short for

```
(local ((define (a-new-name x-1 ... x-n) exp))
 a-new-name)
```

The short-hand explanation suggests that

```
(lambda (x-1 ... x-n) exp)
```

introduces `x-1, ..., x-n` as binding occurrences and that the scope of parameters is `exp`, for example:



Of course, if `exp` contains further binding constructs (say, a nested `local` expression), then the scope of the variables may have a hole.

**Exercise 303.** 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.

---

## ISL 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.

Use the `2htdp/abstraction` library. Instructors who use it for the remainder of the book should explain how the principles of design apply to languages without `for` and `match`.

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 the functional loops of the preceding part. On the 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.

In this section, we introduce ISL+'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 107](#) spells out the grammar of our selected `for` loops as an extension of BSL's grammar from [Intermezzo 1: Beginning Student Language](#). 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)

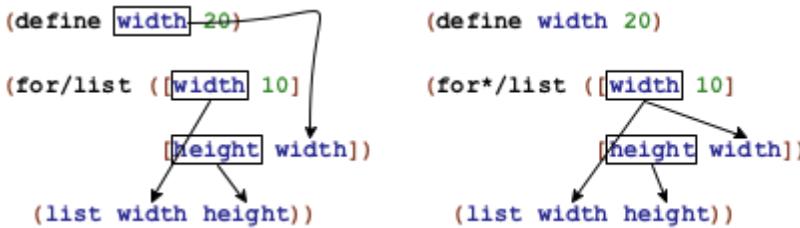
clause = [variable expr]

```

Figure 107: 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-identical code snippets illustrate the difference between these two scoping rules:

Racket's version of these loops comes with more functionality than those presented here, and the language has many more loops than this.



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 the 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 a clause expression evaluates to

- a list, its items make up the sequence values;
- a natural number `n`, the sequence consists of `0, 1, ..., (- n 1)`; and
- a string, its one-character strings are the sequence items.

Next, `for/list` evaluates the loop body with the loop variables successively bound to the values of the generated sequence(s). Finally, it collects the values of its body into a list. 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, we can easily generate the list from `0` to `9`:

```

> (for/list ([i 10])
 i)
(list 0 1 2 3 4 5 6 7 8 9)

```

This is the equivalent of a `build-list` loop:

```

> (build-list 10 (lambda (i) i))
(list 0 1 2 3 4 5 6 7 8 9)

```

The second example “zips” together two sequences:

```
> (for/list ([i 2] [j '(a b)])
 (list i j))
(list (list 0 'a) (list 1 'b))
```

For comparison again, here is the same expression using plain ISL+:

```
> (local ((define i-s (build-list 2 (lambda (i) i)))
 (define j-s '(a b)))
 (map list i-s j-s))
(list (list 0 'a) (list 1 'b))
```

The final example emphasizes designing with `for/list`:

**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]]
; pairs each item in lx with its index

(check-expect
 (enumerate '(a b c)) '((1 a) (2 b) (3 c)))

(define (enumerate lx)
 (for/list ([x lx] [ith (length lx)])
 (list (+ ith 1) x)))
```

The function’s body uses `for/list` to iterate over the given list and a list of numbers from `0` to `(length lx)` (minus `1`); the loop body combines the index (plus `1`) with the list item.

In semantic terms, `for*/list` iterates over the sequences in a nested fashion while `for/list` traverses them in parallel. That is, a `for*/list` expression basically unfolds into a nest of loops:

```
(for*/list ([i 2] [j '(a b)])
 ...)
```

is short for

```
(for/list ([i 2])
 (for/list ([j '(a b)])
 ...))
```

In addition, `for*/list` collects the nested lists into a **single** list by concatenating them with `foldl` and `append`.

**Exercise 304.** Evaluate

```
(for/list ([i 2] [j '(a b)]) (list i j))
```

and

```
(for*/list ([i 2] [j '(a b)]) (list i j))
```

in the interactions area of DrRacket.

Let's continue the exploration by turning the difference in scoping between `for/list` and `for*/list` into a semantic difference:

```
> (define width 2)
> (for/list ([width 3][height width])
 (list width height))
(list (list 0 0) (list 1 1))
> (for*/list ([width 3][height width])
 (list width height))
(list (list 1 0) (list 2 0) (list 2 1))
```

To understand the first interaction, remember that `for/list` traverses the two sequences in parallel and stops when the shorter one is exhausted. Here, the two sequences are

|               |   |                       |
|---------------|---|-----------------------|
| <i>width</i>  | = | 0, 1, 2               |
| <i>height</i> | = | 0, 1                  |
| <i>body</i>   | = | (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 and the absence of a pair containing 2.

Now contrast this situation with `for*/list`:

|               |   |                                     |
|---------------|---|-------------------------------------|
| <i>width</i>  | = | 0    1                2             |
| <i>height</i> | = | 0                0, 1               |
| <i>body</i>   | = | (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 recomputes `height` for a specific value of `width` 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 are collected into a single list of pairs.

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 these lists.

Stop! Take a moment to design the function, using existing abstractions.

As you design `cross`, you work through a table such as:

| cross | '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 to be generated.

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]]
; generates all 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 ([x1 l1][x2 l2])
 (list x1 x2)))
```

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.

```
; [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))]))

; [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 108: A compact definition of arrangements with `for*/list`

**Note** Figure 108 shows another in-context use of `for*/list`. It displays a compact solution of the extended design problem of creating all possible rearrangements of the letters in a given list.

While [Word Games, the Heart of the Problem](#) sketches the proper design of this complex program, figure 108 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 figure merely exhibits the power of these abstractions; for the underlying design, see especially [exercise 477](#). End

We thank Mark Engelberg for suggesting this exhibition of expressive power.

The .../list suffix clearly signals that the loop expression creates a list. In addition, the library comes with `for` and `for*x` loops that have equally suggestive suffixes:

- .../and collects the values of all iterations with `and`:

```
> (for/and ([i 10]) (> (- 9 i) 0))
#false
> (for/and ([i 10]) (if (>= i 0) i #false))
9
```

For pragmatics, the loop returns the last generated value or `#false`.

- .../or is like .../and but uses `or` instead of `and`:

```
> (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`.

- .../sum adds up the numbers that the iterations generate:

```
> (for/sum ([c "abc"]) (string->int c))
294
```

- .../product multiplies the numbers that the iterations generate

```
> (for/product ([c "abc"]) (+ (string->int c) 1))
970200
```

- .../string creates `String`s from the `1String` sequence:

```
> (define a (string->int "a"))
> (for/string ([j 10]) (int->string (+ a j)))
"abcdefg hij"
```

Stop! Imagine how a `for/fold` loop would work.

Stop again! It is an instructive exercise to reformulate 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 which return the appropriate non-`#false` values.

```
; N -> sequence?
; constructs the infinite sequence of natural numbers,
; starting from n
(define (in-naturals n) ...)

; N N N -> sequence?
; constructs the following finite sequence of natural numbers:
; start
; (+ start step)
; (+ start step step)
```

```

; ...

; until the number exceeds end

(define (in-range start end step) ...)

```

Figure 109: 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 nonsequential 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 109](#) 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 lx)
 (for/list ([item lx] [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.

With the second one, it is, for example, possible to step through the even numbers among the first `n`:

```

; N -> Number
; adds 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 305.** Use loops to define `convert-euro`. See [exercise 267](#).

**Exercise 306.** Use loops to define a function 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 1/2 ... 1/n)` for any natural number `n`;
4. creates the list of the first `n` even numbers; and
5. creates a diagonal square of `0`s and `1`s; see [exercise 262](#).

Finally, use loops to define `tabulate` from [exercise 250](#).

**Exercise 307.** 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 271](#).

## 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 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.

The interested instructor may wish to study the facilities of the `2htdp/abstraction` library to define algebraic data types.

Repetition calls 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 these 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 a simplification of Racket's pattern matcher. [figure 110](#), which displays its grammar; `match` is clearly a syntactically complex construct. While its outline resembles that of `cond`, it features patterns instead of conditions, and they come with their own rules.

```
expr = ...
| (match expr [pattern expr] ...)

pattern = variable
| literal-constant
| (cons pattern pattern)
| (structure-name pattern ...)
| (? predicate-name)
```

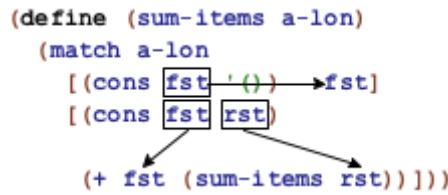
Figure 110: ISL+ match expressions

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:



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 is the case in the above code snippet—their bindings cannot interfere with each other.

Syntactically, a pattern resembles 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`.

- [\(cons pattern<sub>1</sub> pattern<sub>2</sub>\)](#), it matches only an instance of `cons`, assuming its first field matches `pattern1` and its rest 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 our own 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`.

Finally, matching also works across several layers of 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 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`:

**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
; retrieves 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 a Russian doll. Here is the data definition again:

```
(define-struct layer [color doll])
; An RD.v2 (short for Russian doll) is one of:
; - "doll"
; - (make-layer String RD.v2)
```

Here is a definition of `depth` using `match`:

```
; RD.v2 -> N
; how many dolls are a part of an-rd
(check-expect (depth (make-layer "red" "doll")) 1)
(define (depth a-doll)
 (match a-doll
 ["doll" 0]
 [(layer c inside) (+ (depth inside) 1)]))
```

While the pattern in the first `match` clause looks for `"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 makes the function definition concise.

The final problem is an excerpt from the generalized UFO game:

**Sample Problem** Design the `move-right` function. It consumes a list of `Posns`, which represent the positions of objects on a canvas, plus a number. The function adds the latter to each x-coordinate, which represents a rightward movement of these objects.

Here is our solution, using the full power of ISL+:

```
; [List-of Posn] -> [List-of Posn]
; moves each object right by delta-x pixels
```

```
(define input `(,(make-posn 1 1) ,(make-posn 10 14)))
(define expect `(,(make-posn 4 1) ,(make-posn 13 14)))

(define (move-right lop delta-x)
 (for/list ((p lop))
 (match p
 [(posn x y) (make-posn (+ x delta-x) y)])))
```

Stop! Did you notice that we use `define` to formulate the tests? If you give data examples good names with `define` and write down next to them what a function produces as the expected result, you can read the code later much more easily than if you had just written down the constants.

Stop! How does a solution with `cond` and selectors compare? Write it out and compare the two. Which one do you like better?

**Exercise 308.** Design the function `replace`, which substitutes the area code 713 with 281 in a list of phone records.

**Exercise 309.** Design the function `words-on-line`, which determines the number of `Strings` per item in a list of list of strings.

---

## 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 from [Projects: Lists](#) on a rigorous basis 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, reworking it for functions that process two complex arguments at the same time.

---

## 19 The Poetry of S-expressions

Programming resembles poetry. Like poets, programmers practice their skill on seemingly pointless ideas. They revise and edit all the time, as the preceding chapter explains. This chapter introduces increasingly complex forms of data—seemingly without a real-world purpose. Even when we provide a motivational background, the chosen kinds of data are pure to an extreme, and it is unlikely that you will ever encounter them 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 [Project: The Commerce of XML](#), which, in contrast to this chapter, comes much closer to real-world uses of complex forms of data.

---

### 19.1 Trees

All of us have a family tree. One way to draw a family tree is to add an element every time a child is born and to connect the elements of the father and mother. For those people whose parents are unknown, there is no connection to draw. The result is an *ancestor family tree* because, given any person, the tree points to all of the person's known ancestors.

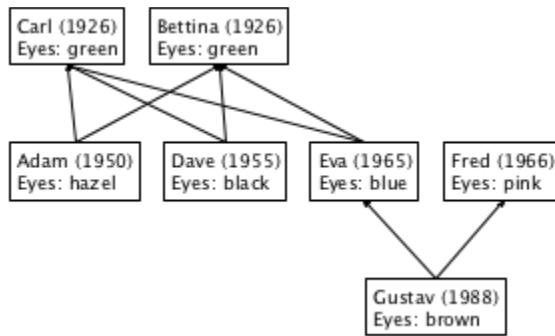


Figure 111: A family tree

[Figure 111](#) 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 point 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 have 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, and 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 "hazel"))
```

assuming `Carl` and `Bettina` stand for representations of Adam's parents.

Then again, a person's parents may be unknown, like Bettina's in the family tree of [figure 111](#). Yet, even then, we must fill the corresponding parent field(s) in the `child` representation. 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 the information is missing from a family tree. 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.

Our experimentation 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 variant describing unknown family trees and another one for known family trees:

```
(define-struct no-parent [])
(define-struct child [father mother name date eyes])
; An FT (short for 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 `NP` as a short-hand and revise the data definition a bit:

```
(define NP (make-no-parent))
; An FT is one of:
; - NP
; - (make-child FT FT String N String)
```

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 111](#) into our data representation. The information for Carl is easy to translate into data:

```
(make-child NP NP "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 NP NP "Carl" 1926 "green")
 (make-child NP NP "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 112](#) illustrates this approach for the complete data representation of the family tree from [figure 111](#). Take a close look; the tree serves as our running example for the following design exercise.

```

; Oldest Generation:
(define Carl (make-child NP NP "Carl" 1926 "green"))
(define Bettina (make-child NP NP "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 NP NP "Fred" 1966 "pink"))

; Youngest Generation:
(define Gustav (make-child Fred Eva "Gustav" 1988 "brown"))

```

Figure 112: A data representation of the sample family tree

Instead of designing a concrete function on family trees, let's first look at the generic organization of such a function. That is, let's work through the design recipe as much as possible without having a concrete task in mind. We start with the header material, that is, step 2 of the recipe:

```

; FT -> ???
; ...
(define (fun-FT an-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 means 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:

```

(define (fun-FT an-ftree)
 (cond
 [(no-parent? an-ftree) ...]
 [else ...]))

```

In case the argument to fun-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-FT an-ftree)
 (cond
 [(no-parent? an-ftree) ...]
 [else (... (child-father an-ftree) ...
 ... (child-mother an-ftree) ...
 ... (child-name an-ftree) ...
 ... (child-date an-ftree) ...
 ... (child-eyes an-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. The

definition for `FT` has two self-references, and the template therefore needs two such recursions:

```
; FT -> ???
(define (fun-FT an-ftree)
 (cond
 [(no-parent? an-ftree) ...]
 [else (.... (fun-FT (child-father an-ftree)) ...
 (fun-FT (child-mother an-ftree)) ...
 (child-name an-ftree) ...
 (child-date an-ftree) ...
 (child-eyes an-ftree) ...)]))
```

Specifically, `fun-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's now turn to a concrete example, the `blue-eyed-child?` function. Its purpose is to determine whether any `child` structure in a given family tree has blue eyes. You may copy, paste, and rename `fun-FT` to get its template; we replace “`???`” with `Boolean` and add a purpose statement:

```
; FT -> Boolean
; does an-ftree contain a child
; structure with "blue" in the eyes field
(define (blue-eyed-child? an-ftree)
 (cond
 [(no-parent? an-ftree) ...]
 [else (.... (blue-eyed-child?
 (child-father an-ftree)) ...
 (blue-eyed-child?
 (child-mother an-ftree)) ...
 (child-name an-ftree) ...
 (child-date an-ftree) ...
 (child-eyes an-ftree) ...)]))
```

When you work in this fashion, you must 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 `child` with a "blue" eye color. Specifically, the `child` representing `Carl` says the eye color is "green"; given that `Carl`'s ancestor trees are empty, they cannot possibly contain a `child` with "blue" eye color:

```
(check-expect (blue-eyed-child? Carl) #false)
```

In contrast, `Gustav` contains a `child` 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 and a child. 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` whatsoever, it cannot contain one with "blue" as the 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. according to the purpose statement for the function,

```
| (blue-eyed-child? (child-father an-ftree))
```

determines whether some `child` in the father's `FT` has "blue" eyes;

2. likewise, `(blue-eyed-child? (child-mother an-ftree))` determines whether someone in the mother's `FT` has blue eyes; and
3. the selector expressions `(child-name an-ftree)`, `(child-date an-ftree)`, and `(child-eyes an-ftree)` extract the name, birth date, and eye color from the given `child` structure, respectively.

Now we just need to figure out how to combine these expressions.

Clearly, if the `child` structure contains "blue" in the `eyes` field, the function's answer is `#true`. Next, the expressions concerning names and birth dates are useless, which leaves us with the recursive calls. As stated, `(blue-eyed-child? (child-father an-ftree))` traverses the tree on the father's side, while the mother's side of the family tree is processed with `(blue-eyed-child? (child-mother an-ftree))`. If either of these expressions returns `#true`, `an-ftree` contains a `child` with "blue" eyes.

Our analysis suggests that the result should be `#true` if one of the following three expressions is `#true`:

- `(string=? (child-eyes an-ftree) "blue")`
- `(blue-eyed-child? (child-father an-ftree))`
- `(blue-eyed-child? (child-mother an-ftree))`

which, in turn, means we need to combine these expressions with `or`:

```
| (or (string=? (child-eyes an-ftree) "blue")
| (blue-eyed-child? (child-father an-ftree))
| (blue-eyed-child? (child-mother an-ftree)))
```

Figure 113 pulls everything together in a single definition.

```
; FT -> Boolean
; does an-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? an-ftree)
```

```
(cond
 [(no-parent? an-ftree) #false]
 [else (or (string=? (child-eyes an-ftree) "blue")
 (blue-eyed-child? (child-father an-ftree))
 (blue-eyed-child? (child-mother an-ftree)))]))
```

Figure 113: 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 NP NP "Carl" 1926 "green"))
```

Let's act as if `NP` were a value and let's use `carl` as an abbreviation for the instance of `child`:

```
==
(cond
 [(no-parent?
 (make-child NP NP "Carl" 1926 "green"))
 #false]
 [else (or (string=? (child-eyes carl) "blue")
 (blue-eyed-child? (child-father carl))
 (blue-eyed-child? (child-mother carl)))]))
```

After dropping the first `cond` line, it's time to replace `carl` with its value and to perform the three auxiliary calculations in [figure 114](#). Using these to replace equals with equals, the rest of the computation is explained easily:

```
==
(or (string=? "green" "blue")
 (blue-eyed-child? (child-father carl))
 (blue-eyed-child? (child-mother carl)))
== (or #false #false #false)
== #false
```

While we trust that you have seen such auxiliary calculations in your mathematics courses, you also need to understand that the stepper would **not** perform such calculations; instead it works out only those calculations that are absolutely needed.

```
; (1)
(child-eyes (make-child NP NP "Carl" 1926 "green"))
==
"green"

; (2)
(blue-eyed-child?
 (child-father
 (make-child NP NP "Carl" 1926 "green"))))
==
(blue-eyed-child? NP)
```

```

==

#false

;

(3)

(blue-eyed-child?

 (child-mother

 (make-child NP NP "Carl" 1926 "green")))

==

(blue-eyed-child? NP)

==

#false

```

Figure 114: Calculating with trees

**Exercise 310.** Develop `count-persons`. The function consumes a family tree and counts the child structures in the tree.

**Exercise 311.** Develop the function `average-age`. It consumes a family tree and the current year. It produces the average age of all child structures in the family tree.

**Exercise 312.** Develop the function `eye-colors`, which consumes a family tree 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 `append` to concatenate the lists resulting from the recursive calls.

**Exercise 313.** Suppose we need the function `blue-eyed-ancestor?`, which is like `blue-eyed-child?` but responds with `#true` only when a proper ancestor, not the given child itself, has blue eyes.

Although the goals clearly differ, the signatures are the same:

```

; FT -> Boolean
(define (blue-eyed-ancestor? an-ftree) ...)

```

Stop! Formulate a purpose statement for the function.

To appreciate the difference, we take a look at Eva:

```
(check-expect (blue-eyed-child? Eva) #true)
```

Eva is blue-eyed, but has no blue-eyed ancestor. Hence,

```
(check-expect (blue-eyed-ancestor? Eva) #false)
```

In contrast, Gustav is Eva's son and 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? an-ftree)
 (cond
 [(no-parent? an-ftree) #false]
 [else
 (or
 (blue-eyed-ancestor?

```

```

 (child-father an-ftree)
 (blue-eyed-ancestor?
 (child-mother an-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?

## 19.2 Forests

It is a short step from a family tree to a family forest:

```

; An FF (short for family forest) is one of:
; - '()
; - (cons FT FF)
; interpretation a family forest represents several
; families (say, a town) and their ancestor trees

```

Here are some trees excerpts from [figure 111](#) 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.

Now consider this 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.

```

; FF -> Boolean
; does the forest contain any child 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 115: Finding a blue-eyed child in a family forest

The straightforward solution is displayed in [figure 115](#). Study the signature, the purpose statement, and the examples on your own. We focus on the program organization. Concerning the template, the design may employ the list template because the function consumes a list. 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's step back and inspect how we explained [figure 115](#). 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. Early chapters gloss over this kind of relationship, but now the situation is sufficiently complicated and deserves attention.

**Exercise 314.** Reformulate the data definition for **FF** with the [List-of](#) abstraction. Now do the same for the `blue-eyed-child-in-forest?` function. Finally, define `blue-eyed-child-in-forest?` using one of the list abstractions from the preceding chapter.

**Exercise 315.** Design the function `average-age`. It consumes a family forest and a year ([N](#)). From this data, it produces the average age of all `child` instances in the forest. **Note** If the trees in this forest overlap, the result isn't a true average because some people contribute more than others. For this exercise, act as if the trees don't overlap.

---

## 19.3 S-expressions

While [Intermezzo 2: 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 is one of: ; An Atom is one of:
; - Atom ; - Number
; - SL ; - String
; - Symbols ; - Symbol
;
; An SL is one of:
; - '()
; - (cons S-expr SL)
```

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 2: 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 316.** 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's work through a specific example:

**Sample Problem** Design the function `count`, which determines how many times some symbol occurs in some S-expression.

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 a list as well as an [S-expr](#):

```
'()
(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 2: Quote, Unquote](#) it is even easier to write down S-expressions. Here are the last three:

```
> '()
'()
> '(hello 20.12 "world")
(list 'hello #i20.12 "world")
> '((hello 20.12 "world"))
(list (list '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)
 (+ x 55))
(list 'define (list 'f 'x) (list '+ '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))
(list (list 6 'f) (list 5 'e) (list 4 'd)))
```

This piece of data looks like a table, associating letters with numbers. The last example is a piece of art:

```
> '(wing (wing body wing) wing)
 (list 'wing (list 'wing 'body '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` is disastrous.

Before we move on to the template step, we need to prepare you for the next generalization of the design recipe:

**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. **End**

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-exps**;
2. one for a function that counts occurrences of symbols in **SLs**; and
3. 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-atom at sy)
 (cond
 [(number? at) ...]
 [(string? at) ...]
 [(symbol? at) ...]))
```

```
(define (count-sl sl sy)
 (cond
 [(empty? sl) ...]
 [else ...]))
```

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 both `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 is to take apart compound data in the relevant clauses:

```
(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) ...])))
```

Why do we add just two selector expressions to `count-sl`?

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's 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`. To indicate the cross-reference from here to `Atom`, 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 extracts a different kind of value. Specifically, `(first sl)` is an element of `S-expr`, which means that we wrap it in `(count ...)`. After all, `count` is responsible for counting inside of arbitrary `S-exps`. Next, `(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.

|                                                                                                     |                                                                                                                        |
|-----------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|
| <code>(define (count sexp sy)   (cond     [(atom? sexp)      (count-atom sexp sy)]     [else</code> | <code>(define (count-atom at sy)   (cond     [(number? at) ...]     [(string? at) ...]     [(symbol? at) ...]))</code> |
|-----------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|

```

(count-sl sexp sy])))

(define (count-sl sl sy)
 (cond
 [(empty? sl) ...]
 [else
 (...
 (count (first sl) sy)
 ...
 (count-sl (rest sl) sy)
 ...)]))

```

Figure 116: A template for S-expressions

```

; 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)]))

```

Figure 117: A program for S-expressions

Figure 116 presents the three complete templates. Filling in the blanks in these templates is straightforward, as figure 117 shows. You ought to be able to explain any random line in the three definitions. For example:

[(atom? sexp) (count-atom sexp sy)]

determines whether sexp is an **Atom** and, if so, interprets the **S-expr** as an **Atom** via count-atom.

[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.

**Exercise 317.** A program that consists of three connected functions ought to express this relationship with a `local` expression.

Copy and reorganize the program from [figure 117](#) into a single function using `local`. Validate the revised code with the test suite for `count`.

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 to tell the reader that `sy` is a constant across the traversal process.

**Exercise 318.** 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 319.** Design `substitute`. 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 320.** 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. Redesign the `count` function for this data definition.

Now integrate the definition of `SL` into the one for `S-expr`. Simplify `count` again. Consider using `lambda`.

**Note** This kind of simplification is not always possible, but experienced programmers tend to recognize such opportunities.

**Exercise 321.** Abstract the data definitions for `S-expr` and `SL` so that they abstract over the kinds of `Atoms` that may appear.

---

## 19.4 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, draw arrows to connect references to definitions. Consider the left side of [figure 118](#). It displays the definition for `S-expr`, which contains references to `SL` and `Atom` that 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 of definitions also call for validation. At a minimum, you must be able to construct some examples for every individual definition. Start from clauses that do not refer to any of the other data definitions in the 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 52](#) to guide the template creation up to the last step. The revised last step 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 118](#) for the arrow-annotated version of the templates.

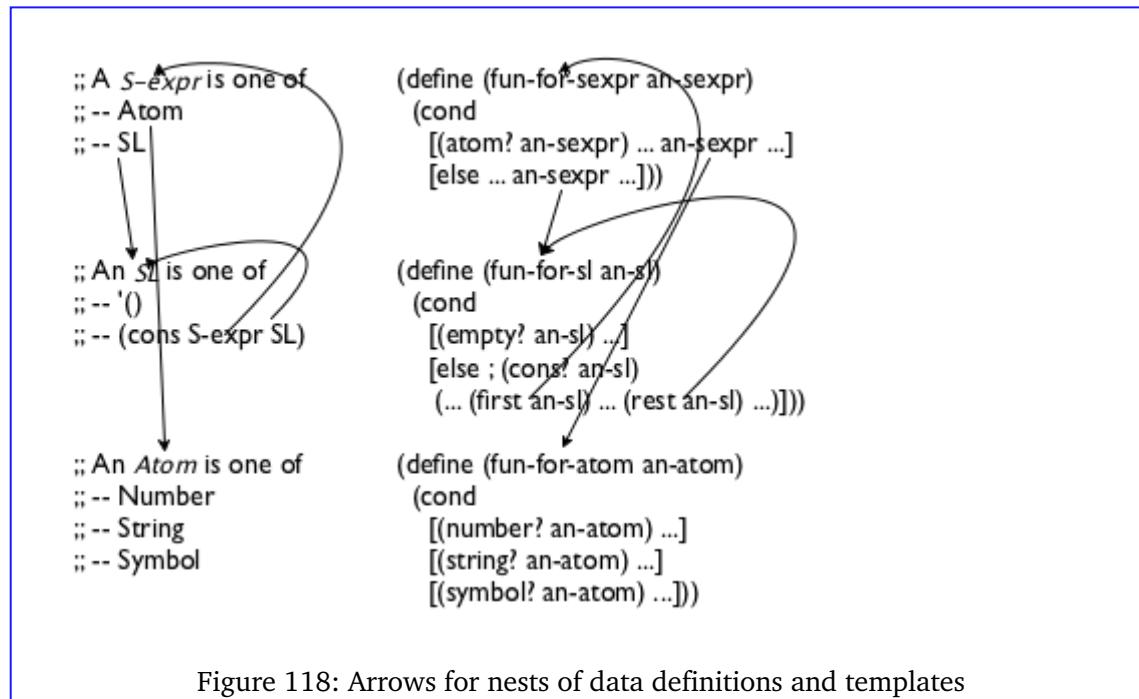


Figure 118: Arrows for nests of data definitions and templates

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** Observe 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 *symmetry*. 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 53](#) guide you.

6. Run the tests when all definitions are completed. If an auxiliary function is broken, you may get two error reports, 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 combination function. In the case of intertwined data, you may need not only a table per case but also a table per case and per function to work out the combination.

---

## 19.5 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 BT (short for BinaryTree) 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:

```
(make-node
 15
 'd
 NONE
 (make-node
 24
 'i
 NONE
 NONE))
 (make-node
 15
 'd
 (make-node
 87
 'h
 NONE
 NONE)
 NONE)
```

[Figure 119](#) shows how we should think about such trees as drawings. The trees are drawn upside down, 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 drawings omit `NONE`.

tree A

tree B

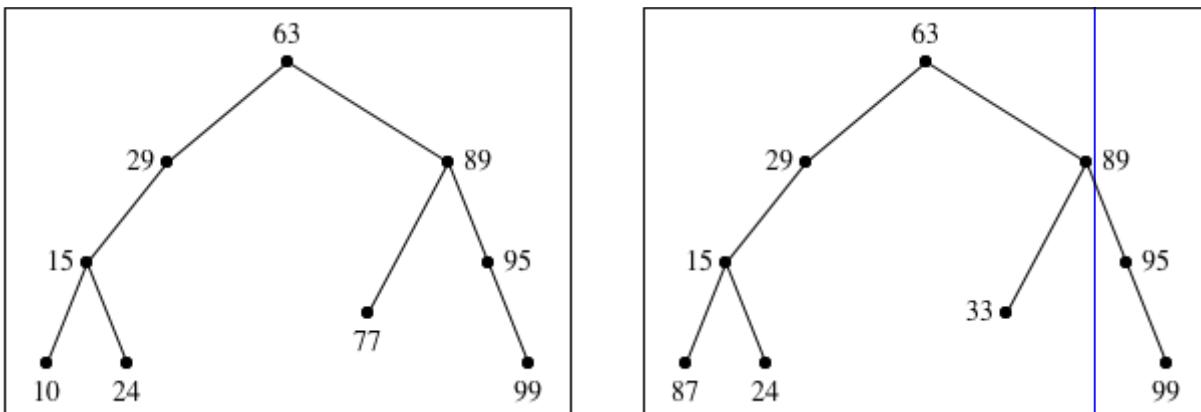


Figure 119: A binary search tree and a binary tree

**Exercise 322.** Draw the above two trees in the manner of [figure 119](#). Then design `contains-bt?`, which determines whether a given number occurs in some given BT.

**Exercise 323.** 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** Consider using `contains-bt?` to check the entire tree first or `boolean?` to check the result of the natural recursion at each stage.

If we read the numbers in the two trees in [figure 119](#) 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 of the first kind 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 **BST** (short for *binary search tree*) 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` are smaller than `ssn0`,
  - all `ssn` fields in `R` are larger than `ssn0`.

In other words, to check whether a BT also belongs to BST, we must inspect all numbers in all subtrees and ensure that they are smaller or larger than some given number. This places an additional burden on the construction of data, but, as the following exercises show, it is well worth it.

**Exercise 324.** 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 the idea with this sample `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 325.** 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.

See [exercise 189](#) for searching in sorted lists. Compare!

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 `BST` 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 326.** 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 the design is completed, use the function on tree A from [figure 119](#).

**Exercise 327.** Design the function `create-bst-from-list`. It consumes a list of numbers and names and produces a **BST** by repeatedly applying `create-bst`. Here is the signature:

```
; [List-of [List Number Symbol]] -> BST
```

Use the complete function to create a **BST** from this sample input:

```
'((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 119](#), 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?

---

## 19.6 Simplifying Functions

[Exercise 317](#) 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 319](#).

```
; S-expr Symbol Atom -> S-expr
; replaces all occurrences of old in sexp with new

(check-expect (substitute '(((world) bye) bye) 'bye '42)
 '(((world) 42) 42))

(define (substitute sexp old new)
 (local (; S-expr -> S-expr
 (define (for-sexp sexp)
 (cond
 [(atom? sexp) (for-atom sexp)]
 [else (for-sl sexp)])))
 ; SL -> S-expr
 (define (for-sl sl)
 (cond
 [(empty? sl) '()]
 [else (cons (for-sexp (first sl))
 (for-sl (rest sl))))]))
 ; Atom -> S-expr
 (define (for-atom at)
 (cond
 [(number? at) at]
 [(string? at) at]
 [(symbol? at) (if (equal? at old) new at)]))))
```

```
(for-sexp sexp)))
```

Figure 120: A program to be simplified

Figure 120 displays a complete definition of the `substitute` function. The definition uses `local` and three auxiliary functions as suggested by the data definition. The figure includes a test case so that you can retest the function after each edit suggested below. Stop! Develop additional test cases; one is almost never enough.

**Exercise 328.** Copy and paste [figure 120](#) into DrRacket; include your test suite. Validate the test suite. As you read along the remainder of this section, perform the edits and rerun the test suites to confirm the validity of our arguments.

```
(define (substitute sexp old new)
 (local (; S-expr -> S-expr
 (define (for-sexp sexp)
 (cond
 [(atom? sexp) (for-atom sexp)]
 [else (for-sl sexp)])))
 ; SL -> S-expr
 (define (for-sl sl)
 (map for-sexp sl))
 ; Atom -> S-expr
 (define (for-atom at)
 (cond
 [(number? at) at]
 [(string? at) at]
 [(symbol? at) (if (equal? at old) new at)])))
 (for-sexp sexp)))
```

Figure 121: Program simplification, step 1

Since we know that `SL` describes lists of `S-expr`, we can use `map` to simplify `for-sl`. See [figure 121](#) for the result. While the original program says that `for-sexp` is applied to every item on `sl`, its revised definition expresses the same idea more succinctly with `map`.

For the second simplification step, we need to remind you that `equal?` compares two arbitrary values. With this in mind, the third `local` function becomes a one-liner. [Figure 122](#) displays this second simplification.

```
(define (substitute sexp old new)
 (local (; S-expr -> S-expr
 (define (for-sexp sexp)
 (cond
 [(atom? sexp) (for-atom sexp)]
 [else (for-sl sexp)])))
 ; SL -> S-expr
 (define (for-sl sl) (map for-sexp sl))
 ; Atom -> S-expr
 (define (for-atom at)
 (if (equal? at old) new at)))
 (for-sexp sexp)))
```

```
(define (substitute.v3 sexp old new)
 (local (; S-expr -> S-expr
 (define (for-sexp sexp)
 (cond
 [(atom? sexp)
 (if (equal? sexp old) new sexp)]
 [else
 (map for-sexp sexp)])))
 (for-sexp sexp)))
```

Figure 122: Program simplification, steps 2 and 3

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 `for-sexp`. In-lining means replacing `(for-atom sexp)` with `(if (equal? sexp old) new sexp)`, that is, we replace the parameter at with the actual argument `sexp`. Similarly, for `(for-sl sexp)` we put in `(map for-sexp sexp)`; see the bottom half of [figure 121](#). 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 locally defined function can be used in lieu of the outer one.

While `sexp` is also a parameter, this substitution is really acceptable because it, too, stands in for an actual value.

Here is the result of translating this last thought into code:

```
(define (substitute sexp old new)
 (cond
 [(atom? sexp) (if (equal? 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.

## 20 Iterative 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, for example. 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 repeats—or as programmers say, *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.

## 20.1 Data Analysis

Before you turn off DrRacket, you want to make sure that all your work is safely 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 roughly a string.

A file is really a sequence of *bytes*, one after another.  
Try to define the class of files.

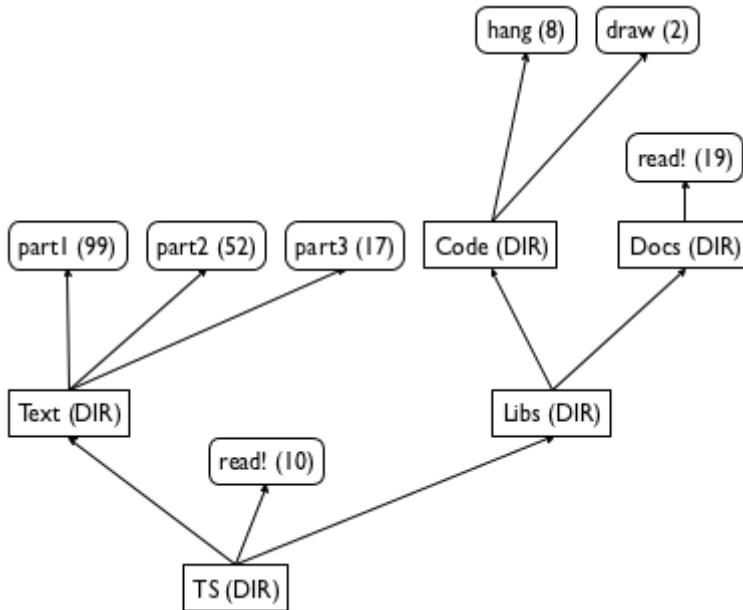


Figure 123: A sample directory tree

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 sub-directories and may contain yet more sub-directories and files. Because of the hierarchy, we speak of *directory trees*.

[Figure 123](#) contains a graphical sketch of a small directory tree, and the picture explains why computer scientists call them trees. Contrary to convention in computer science, the figure shows the tree growing upward, with a root directory named `TS`. The root directory contains one file, called `read!`, and two sub-directories, called `Text` and `Libs`, respectively. The first sub-directory, `Text`, contains only three files; the latter, `Libs`, contains only two sub-directories, 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 329.** 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 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?

---

## 20.2 Refining Data Definitions

[Exercise 329](#) 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 them. 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 123](#) 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.

**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 strings, that is, 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 String.
```

The names have a `.v1` suffix to distinguish them from future refinements.

**Exercise 330.** Translate the directory tree in [figure 123](#) into a data representation according to model 1.

**Exercise 331.** Design the function `how-many`, which determines how many files a given `Dir.v1` contains. Remember to follow the design recipe; [exercise 330](#) provides you with data examples.

**Model 2** If you solved [exercise 331](#), 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 sub-directories 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 String LOFD)

; An LOFD (short for list of files and directories) is one of:
; - ()
; - (cons File.v2 LOFD)
; - (cons Dir.v2 LOFD)

; A File.v2 is a String.
```

Note how the data definition for `Dir.v2` refers to the definition for `LOFD`s and the one for `LOFDs` refers back to that of `Dir.v2`. The two definitions are mutually recursive.

**Exercise 332.** Translate the directory tree in [figure 123](#) into a data representation according to model 2.

**Exercise 333.** Design the function `how-many`, which determines how many files a given `Dir.v2` contains. [Exercise 332](#) provides you with data examples. Compare your result with that of [exercise 331](#).

**Exercise 334.** 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 content) 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 String N String)
```

As indicated by the field names, the string represents the name of the file, the natural number its size, and the string its content.

Finally, let's split the content field of directories into two pieces: a list of files and a list of sub-directories. 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 String 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 335.** Translate the directory tree in [figure 123](#) into a data representation according to model 3. Use `" "` for the content of files.

**Exercise 336.** Design the function `how-many`, which determines how many files a given `Dir.v3` contains. [Exercise 335](#) provides you with data examples. Compare your result with that of [exercise 333](#).

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 337.** Use [List-of](#) to simplify the data definition `Dir.v3`. Then use ISL+’s list-processing functions from [figures 95](#) and [96](#) to simplify the function definition(s) for the solution of [exercise 336](#).

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.

## 20.3 Refining Functions

To make the following exercises somewhat realistic, DrRacket comes with the `dir.rkt` 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 representations of directory trees on your computer:

Add (`require htdp/dir`) to the definitions area.

```
; String -> Dir.v3
; creates a representation of the a-path directory
(define (create-dir a-path) ...)
```

For example, if you open DrRacket and enter the following three lines into the definitions area:

```
(define O (create-dir "/Users/...")) ; on OS X
(define L (create-dir "/var/log/")) ; on Linux
(define W (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.

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 allows you to complete the respective exercise. Feel free to use the data definition from [exercise 337](#) and the functions from [figures 95](#) and [96](#).

**Exercise 338.** Use `create-dir` to turn some of your directories into ISL+ data representations. Then use `how-many` from [exercise 336](#) to count how many files they contain. Why are you confident that `how-many` produces correct results for these directories?

**Exercise 339.** 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 340.** Design the function `ls`, which lists the names of all files and directories in a given *Dir*.

**Exercise 341.** 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:

```
; A Path is [List-of String].
; interpretation directions into a directory tree
```

Take a second look at [figure 123](#). In that diagram, the path from TS to part1 is (`list "TS" "Text" "part1"`). Similarly, the path from TS to Code is (`list "TS" "Libs" "Code"`).

**Exercise 342.** 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 sub-directory. 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!` in [figure 123](#). Design `find-all`, which generalizes `find` and produces the list of all paths that lead to *f* in *d*. What should `find-all` produce when `(find? d f)` is `#false`? Is this part of the problem really a challenge compared to the basic problem?

**Exercise 343.** Design the function `ls-R`, which lists the paths to **all** files contained in a given *Dir*.

**Exercise 344.** Redesign `find-all` from [exercise 342](#) using `ls-R` from [exercise 343](#). This is design by composition, and if you solved the challenge part of [exercise 342](#) your new function can find directories, too.

---

## 21 Refining Interpreters

DrRacket is a program. It is a complex one, 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 that 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 for this design project. As a matter of fact, the very idea of focusing on this aspect of DrRacket is another instance of refinement, namely, the obvious one of implementing only one piece of functionality.

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 that is also called *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 sister in fifth 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.

---

### 21.1 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. At first, 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. An addition expression, however, calls for compound data because it contains two expressions and because it is distinct from a multiplication expression, which also needs a data representation.

Following [Adding Structure](#), a straightforward way to represent additions and multiplications is to define two structure types, 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 concerns 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 deeply as you wish. See [figure 124](#) for additional examples.

| 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))    | (make-add (make-mul (make-add 1 2) 3))    |
| (* (* (+ 1 1) 2) 4)) | (make-mul (make-mul (make-add 1 1) 2) 4)) |

Figure 124: Representing BSL expressions in BSL

**Exercise 345.** Formulate a data definition for the representation of BSL expressions based on the structure type definitions of `add` and `mul`. Let's 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)) 3.14)

Here “interpret” means “translate from data into information.” In contrast, “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, either you can 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 346.** Formulate a data definition for the class of values to which a representation of a BSL expression can evaluate.

**Exercise 347.** Design eval-expression. The function consumes a representation of a BSL expression and computes its value.

**Exercise 348.** Develop a data representation for Boolean BSL expressions constructed from #true, #false, and, or, and not. Then design eval-bool-expression, which consumes (representations of) Boolean BSL expressions and computes their values. What kind of values do these Boolean expressions yield?

**Convenience and parsing** S-expressions offer a convenient way to represent BSL expressions in our programming language:

```
> (+ 1 1)
2
> '(+ 1 1)
(list '+ 1 1)
> (+ (* 3 3) (* 4 4))
25
> '(+ (* 3 3) (* 4 4))
(list '+ (list '* 3 3) (list '* 4 4))
```

By simply putting a quote in front of an expression, we get ISL+ data.

Interpreting an S-expression representation is clumsy, mostly because not all S-expressions represent BSL-exprs. For example, #true, "hello", and '(+ x 1) are not representatives of BSL expressions. As a result, S-expressions are quite inconvenient for the designers of interpreters.

Programmers invented *parsers* to bridge the gap between convenience of use and implementation. A parser simultaneously checks whether some piece of data conforms to a data definition and, if it does, builds a matching element from the chosen class of data. The latter is called a *parse tree*. If the given data does not conform, a parser signals an error, much like the checked functions from [Input Errors](#).

[Figure 125](#) presents a BSL parser for S-expressions. Specifically, parse consumes an S-expr and produces a BSL-expr—if and only if the given S-expression is the result of quoting a BSL expression that has a BSL-expr representative.

**Exercise 349.** Create tests for parse until DrRacket tells you that every element in the definitions area is covered during the test run.

**Exercise 350.** What is unusual about the definition of this program with respect to the design recipe?

**Note** One unusual aspect is that parse uses length on the list argument. Real parsers avoid length because it slows the functions down.

**Exercise 351.** Design interpreter-expr. The function accepts S-expressions. If parse recognizes them as BSL-expr, it produces their value. Otherwise, it signals the same error as parse.

```
; 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 WRONG)]
 [(symbol? s) (error WRONG)]))

```

Figure 125: From S-expr to BSL-expr

## 21.2 Interpreting Variables

Since the first section ignores constant definitions, an expression does not have a value if it contains a variable. Indeed, unless we 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 programmers 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 definition, say `(define y 3)`, and interactions that involve two variables:

```

> (+ (* x x)
 (* y y))

```

The preceding section 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))
(list '* 0.5 (list '* 'x 3))
```

One obvious alternative is a string, so that "x" would represent x, but this book is not about designing interpreters, so we stick with symbols. From this decision, it follows how to modify the data definition from [exercise 345](#):

```
; 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 |
|-----------------|----------------------------------|
| x               | 'x                               |
| (+ x 3)         | (make-add 'x 3)                  |
| (* 1/2 (* x 3)) | (make-mul 1/2 (make-mul 'x 3))   |
| (+ (* x x))     | (make-add (make-mul 'x 'x))      |
| (* y y))        | (make-mul 'y 'y))                |

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 a perfectly fine way.

**Exercise 352.** 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 353.** Design the `numeric?` function. It determines whether a **BSL-var-expr** is also a **BSL-expr**. Here we assume that your solution to [exercise 345](#) is the definition for **BSL-var-expr** without **Symbols**.

**Exercise 354.** Design `eval-variable`. The checked function consumes a **BSL-var-expr** and determines its value if `numeric?` yields true for the input. Otherwise it signals an error.

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 definitions area when it contains a series of constant definitions. For this exercise we use association lists:

```
; An AL (short for association list) is [List-of Association].
```

```
; An Association is a list of two items:
; (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. We provide this hint because the creation of this function requires a little design knowledge from [Simultaneous Processing](#).

**An environment model** [Exercise 354](#) relies on the mathematical understanding of constant definitions. If a name is defined to stand for some value, 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, dubbed the *environment model*, is to look up the value of a variable when needed. 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 355.** Design eval-var-lookup. This function has the same signature as eval-variable\*:

```
; BSL-var-expr AL -> Number
(define (eval-var-lookup e da) ...)
```

Instead of using substitution, the function traverses the expression in the manner that the design recipe for [BSL-var-expr](#) suggests. As it descends the expression, it “carries along” da. When it encounters a symbol x, it uses `assq` to look up the value of x in the association list. If there is no value, eval-var-lookup signals an error.

---

## 21.3 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 all of BSL.

The goal of this section is to refine the evaluator of [Interpreting Variables](#) so that it can cope with functions. Since function definitions show up in the definitions area, another way to describe the refined evaluator is to say that it simulates DrRacket when the definitions area contains a number of function definitions and a programmer enters an expression in the interactions area that contains uses of these functions.

For simplicity, let’s assume that all functions in the definitions area consume one argument and that there is only one such definition. The necessary domain knowledge dates back to school where you learned that  $f(x) = e$  represents the definition of function  $f$ , that  $f(a)$  represents the application of  $f$  to  $a$ , and that to evaluate the latter, you substitute  $a$  for  $x$  in  $e$ . As it turns out, the evaluation of function applications in a language such as BSL works mostly like that, too.

Before tackling the following exercises, you may wish to refresh your knowledge of the terminology concerning functions as presented in intermezzo 1. Most of the time, algebra courses gloss over this aspect of mathematics, but a precise use and understanding of terminology is needed when you wish to solve these problems.

**Exercise 356.** 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.

Represent these expressions: `(k (+ 1 1))`, `(* 5 (k (+ 1 1)))`, `(* (i 5) (k (+ 1 1)))`. We refer to this newly defined class of data as *BSL-fun-expr*.

**Exercise 357.** Design `eval-definition1`. The function 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 function's parameter; and
4. a [BSL-fun-expr](#) `b`, which represents the function's body.

It determines the value of `ex`. When `eval-definition1` encounters an application of `f` to some argument, it

1. 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:

```
((local ((define value (eval-definition1 arg f x b))
 (define plugd (subst b x arg-value)))
 (eval-definition1 plugd f x b)))
```

Notice that this line uses a form of recursion that has not been covered. 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 354](#). It also signals an error for function applications that refer to a function name other than `f`.

**Warning** The use of this uncovered form of recursion introduces a new element into your computations: non-termination. That is, a program may run forever instead of delivering a result or signaling an error. If you followed the design recipes of the first four parts, you cannot write down such programs. For fun, construct an input for `eval-definition1` that causes it to run forever. Use `STOP` to terminate the program.

For an evaluator that mimics the interactions area, we need a representation of the definitions area. We assume that it is a list of definitions.

**Exercise 358.** Provide a structure type and a data definition for function definitions. Recall that such a definition has three essential attributes:

1. the function's name, which is represented with a symbol;
2. the function's parameter, which is also a name; and
3. the function's body, which is a variable expression.

We use *BSL-fun-def* to refer to this class of data.

Use your data definition to represent these 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 a definitions area that consists of a number of one-argument function definitions. Translate the definitions area that defines f, g, and h into your data representation and name it da-fgh.

Finally, work on the following wish:

```
; BSL-fun-def* Symbol -> BSL-fun-def
; retrieves the definition of f in da
; signals an error if there is none
(check-expect (lookup-def da-fgh 'g) g)
(define (lookup-def da f) ...)
```

Looking up a definition is needed for the evaluation of applications.

**Exercise 359.** Design eval-function\*. The function consumes ex, a *BSL-fun-expr*, and da, a *BSL-fun-def\** representation of a definitions area. 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 357. 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, which comes with a parameter and a body;
3. substitutes the value of the argument for the function parameter in the function's body; and
4. evaluates the new expression via recursion.

Like DrRacket, eval-function\* signals an error when it encounters a variable or function name without definition in the definitions area.

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)
 (* 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` in the interactions area:

```
> (area-of-circle 1)
#i3.14
> (volume-of-10-cylinder 1)
#i31.4000000000000002
> (* 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 360.** Formulate a data definition for the representation of DrRacket's definitions 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 name this class of data *BSL-da-all*.

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 361.** Design `eval-all`. Like `eval-function*` from [exercise 359](#), 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 355](#).

**Exercise 362.** It is cumbersome to enter the structure-based data representation of BSL expressions and a definitions area. As the end of [Interpreting Expressions](#) demonstrates, it is much easier to quote expressions and (lists of) definitions.

Design a function `interpreter`. It consumes an S-expr and an S1. 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 361](#) to evaluate the expression. **Hint** You must adapt the ideas of [exercise 350](#) to create a parser for definitions and lists of definitions.

You should know that eval-all-sepxr makes it straightforward to check whether it really mimics DrRacket's evaluator.

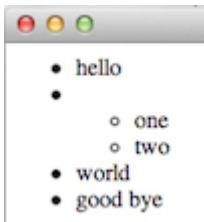
At this point, you know a lot about interpreting BSL. Here are some of the missing pieces: **Booleans** with `cond` or `if`; **Strings** and such operations `string-length` or `string-append`; and lists with `'()`, `empty?`, `cons`, `cons?`, `first`, `rest`; and so on. Once your evaluator can cope with 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.

---

## 22 Project: 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 comparison illustrates this idea with a concrete example:

| XML data                                                                                                                                                                                                                                      | rendered in a browser                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>&lt;ul&gt;   &lt;li&gt; hello &lt;/li&gt;   &lt;li&gt; &lt;ul&gt;     &lt;li&gt; one &lt;/li&gt;     &lt;li&gt; two &lt;/li&gt;   &lt;/ul&gt; &lt;/li&gt; &lt;li&gt; world &lt;/li&gt; &lt;li&gt; good bye &lt;/li&gt; &lt;/ul&gt;</pre> |  <p>A screenshot of a web browser window. The title bar has three colored buttons (red, yellow, green). The main content area displays a nested list. The first item is a bullet point followed by the word "hello". Below it is another bullet point. Underneath that bullet point are two more bullet points, "one" and "two", each preceded by a small circle. The next item in the list is a bullet point followed by the word "world". The final item is a bullet point followed by the phrase "good bye".</p> |

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. The next section 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.

If you think XML is too old-fashioned for 2018, feel free to redo the exercise for JSON or some other modern data exchange format. The design principles remain the same.

---

### 22.1 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 short-hand:

```
<machine />
```

But, as far as we are concerned here, this short-hand is equivalent to the explicitly bracketed version.

From an S-expression perspective, an XML element is a **named** pair of parentheses that surround some content. And indeed, representing the above with an S-expression is quite natural:

Racket's `xml` library represents XML with structures as well as S-expressions.

```
'(machine)
```

This piece of data has the opening and closing parentheses, 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 short-hand, 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:

```
<machine><action /><action /><action /></machine>
```

Stop! Expand the short-hand for `<action />` before you continue.

The S-expression representation continues to look simple. Here is the first one:

```
'(machine (action))
```

And this is the representation for the second one:

```
'(machine (action) (action) (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 a 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.

Naturally, S-expressions for these “machine” elements look much like their XML cousins:

```
'(machine ((initial "red")))
```

XML is 40 years younger than S-expressions.

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 has one opening parenthesis.

You may recall the idea from [Intermezzo 2: 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 nontrivial XML data and even templates of XML representations using backquote and [unquote](#). Of course, [Interpreting Expressions](#) 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 this old, poetic idea in practical terms. We proceed gradually to work out a data definition, putting iterative refinement to work. Here is a first attempt:

```
; An Xexpr.v0 (short for X-expression) is a one-item list:
; (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 a list:
; (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 has two clauses:

```
; An Xexpr.v2 is a list:
; - (cons Symbol Body)
; - (cons Symbol (cons [List-of Attribute] Body))
; where Body is short for [List-of Xexpr.v2]
; An Attribute is a list of two items:
; (cons Symbol (cons String '()))
```

**Exercise 363.** All elements of [Xexpr.v2](#) start with a [Symbol](#), but some are followed by a list of attributes and some by just a list of [Xexpr.v2](#)s. Reformulate the definition of [Xexpr.v2](#) to isolate the common beginning and highlight the different kinds of endings.

Eliminate the use of [List-of](#) from [Xexpr.v2](#).

**Exercise 364.** Represent this XML data as elements of [Xexpr.v2](#):

```
1. <transition from="seen-e" to="seen-f" />
2. <word /><word /><word />
```

Which one could be represented in [Xexpr.v0](#) or [Xexpr.v1](#)?

**Exercise 365.** Interpret the following elements of [Xexpr.v2](#) as XML data:

1. '(server ((name "example.org")))
2. '(carcas (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-attr`, which extracts the list of attributes; and
- `xexpr-content`, which extracts the list of content elements.

Once we have these functions, we can use lists to represent XML yet that act as if they were instances of a structure type.

These functions parse S-expressions, and parsers are tricky to design. So let's design them carefully, starting with some data examples:

```
(define a0 '((initial "X")))

(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-attr xe) '())
```

Here we focus on `xexpr-attr`; we leave the other two as exercises.

Making up functional examples requires a decision concerning the extraction of attributes from X-expressions without any. While our chosen representation completely omits missing attributes, we must supply '( )' for the structure-based representation of XML. The function therefore produces '( )' for such X-expressions:

```
(check-expect (xexpr-attr e0) '())
(check-expect (xexpr-attr e1) '((initial "X")))
(check-expect (xexpr-attr e2) '())
(check-expect (xexpr-attr e3) '())
(check-expect (xexpr-attr e4) '((initial "X")))
```

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's translate these two insights into a template:

```
(define (xexpr-attr 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-attr 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 already 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 -> ???
; determines 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-attr`; see [figure 126](#). If the first item is a list of attributes, the function produces it; otherwise there are no attributes.

```
(define (xexpr-attr 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
 '()))])))

```

Figure 126: The complete definition of `xexpr-attr`

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 x 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 signature combines two data definitions separated by the English word “or.” In ISL+ such an informal signature with a definite meaning is acceptable on occasion.

**Exercise 366.** Design `xexpr-name` and `xexpr-content`.

**Exercise 367.** The design recipe calls for a self-reference in the template for `xexpr-attr`. Add this self-reference to the template and then explain why the finished parsing function does not contain it.

**Exercise 368.** Formulate a data definition that replaces the informal “or” signature for the definition of the `list-of-attributes?` function.

**Exercise 369.** Design `find-attr`. 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, *Xexpr* refers to [Xexpr.v2](#). Also, we assume `xexpr-name`, `xexpr-attr`, and `xexpr-content` are defined. Finally, we use `find-attr` from [exercise 369](#) to retrieve attribute values.

## 22.2 Rendering XML Enumerations

XML is actually a **family** of languages. People define dialects for specific channels of communication. For example, XHTML is the language for sending web content 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 chapter.

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, even enumerations. With “unordered” HTML means is 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:

```
; An XWord is '(word ((text String))).
```

Here, we use this second option; Racket, the language from which the teaching languages are derived, offers libraries that include `String` in `Xexpr`.

**Exercise 370.** Make up three examples for `XWord`s. Design `word?`, which checks whether some ISL+ value is in `XWord`, and `word-text`, which extracts the value of the only attribute of an instance of `XWord`.

**Exercise 371.** Refine the definition of `Xexpr` so that you can represent XML elements, including items in enumerations, that are plain strings.

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 Attributes [List-of XItem.v1]))
; An XItem.v1 is one of:
; - (cons 'li (cons XWord '()))
; - (cons 'li (cons Attributes (cons XWord '()))))
```

For completeness, the data definition includes attribute lists, even though they do not affect rendering.

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:

```
• one
• two
```

The radius of the bullet and the distance between the bullet and the text are matters of aesthetics; here the idea matters.

To create this kind of image, you might use this ISL+ program:

We developed these expressions in the interactions area. What would you do?

```
(define e0-rendered
 (above/align
 'left
 (beside/align 'center BT (text "one" 12 'black))
 (beside/align 'center BT (text "two" 12 'black))))
```

assuming BT is a rendering of a bullet.

Now let's 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 an 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 a-word (word-text element)))
 (... a-word ...)))
```

From here, it is straightforward:

```
(define (render-item1 i)
 (local ((define content (xexpr-content i))
 (define element (first content)))
 (define a-word (word-text element))
 (define item (text a-word 12 'black)))
 (beside/align 'center BT item)))
```

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 372.** Before you read on, equip the definition of `render-item1` with tests. Make sure to formulate these tests in such a way that they don't depend on the `BT` constant. 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, the first two design steps are easy:

```
; XEnum.v1 -> Image
; renders a simple enumeration as an image
(check-expect (render-enum1 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) ...)) ; [List-of XItem.v1]
```

While the data-oriented design recipe tells you that you should design a separate function whenever you encounter a complex form of data, the abstraction-based design recipe from `Abstraction` tells you to reuse an existing abstraction, say, a list-processing function from [figures 95](#) and [96](#), 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 reuse design recipe:

```
(define (render-enum1 xe)
 (local ((define content (xexpr-content xe))
 ; XItem.v1 Image -> Image
 (define (deal-with-one item so-far)
 ...))
 (foldr deal-with-one empty-image content)))
```

From the type matching, you also know that:

1. the first argument to `foldr` must be a two-argument function;
2. the second argument must be an image; and
3. the last argument is the list representing XML content.

Naturally `empty-image` is the correct starting point.

This design-by-reuse focuses our attention on 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` articulates this insight. Since the first argument is an `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. To stack them, we use `above`:

```
(define (render-enum1 xe)
 (local ((define content (xexpr-content xe))
 ; XItem.v1 Image -> Image
 (define (deal-with-one item so-far)
 (above/align 'left
```

```

 (render-item1 item)
 so-far)))
(foldr deal-with-one empty-image content)))

```

```

; An XItem.v2 is one of:
; - (cons 'li (cons XWord '()))
; - (cons 'li (cons [List-of Attribute] (list XWord)))
; - (cons 'li (cons XEnum.v2 '()))
; - (cons 'li (cons [List-of Attribute] (list XEnum.v2)))
;
; An XEnum.v2 is one of:
; - (cons 'ul [List-of XItem.v2])
; - (cons 'ul (cons [List-of Attribute] [List-of XItem.v2]))

```

Figure 127: A realistic data representation of XML enumerations

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 arbitrarily nested enumerations that arrive over the web. 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. [Figure 127](#) displays the second revision of the data definition. It includes a revision of the data definition for enumerations so that the first definition refers to the correct form of item.

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 nesting and then use it.

```

(define SIZE 12) ; font size
(define COLOR "black") ; font color
(define BT ; a graphical constant
 (beside (circle 1 'solid 'black) (text " " SIZE COLOR)))

; Image -> Image
; marks item with bullet
(define (bulletize item)
 (beside/align 'center BT item))

; XEnum.v2 -> Image
; renders an XEnum.v2 as an image
(define (render-enum xe)
 (local ((define content (xexpr-content xe))
 ; XItem.v2 Image -> Image
 (define (deal-with-one item so-far)
 (above/align 'left (render-item item) so-far)))
 (foldr deal-with-one empty-image content)))

; XItem.v2 -> Image
; renders one XItem.v2 as an image
(define (render-item an-item)
 (local ((define content (first (xexpr-content an-item)))))
 (bulletize
 (foldr deal-with-one empty-image content))))

```

```

(cond
[(word? content)
 (text (word-text content) SIZE 'black))]
[else (render-enum content)])))

```

Figure 128: Refining functions to match refinements of data definitions

The next question is how this change to the data definition affects the rendering functions. Put differently, we need to revise `render-enum1` and `render-item1` so that they can cope with `XEnum.v2` and `XItem.v2`, respectively. Software engineers face these kinds of questions all the time, and it is another situation where the design recipe shines.

[Figure 128](#) 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 373.** [Figure 128](#) is missing test cases. Develop test cases for all the functions.

**Exercise 374.** The data definitions in [figure 127](#) use `list`. Rewrite them so they use `cons`. Then use the recipe to design the rendering functions for `XEnum.v2` and `XItem.v2` from scratch. You should come up with the same definitions as in [figure 128](#).

**Exercise 375.** The wrapping of `cond` with

```
(beside/align 'center BT ...)
```

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 376.** Design a program that counts all "hello"s in an instance of `XEnum.v2`.

**Exercise 377.** Design a program that replaces all "hello"s with "bye" in an enumeration.

## 22.3 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 *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.

While the construction of a DSL is often considered a task for an advanced programmer, you are actually in a position already to understand, appreciate, and implement a reasonably complex DSL. This section explains how it all works. It first reacquaints 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.

Because configurations abstract a program over various pieces of data, Paul Hudak argued in the 1990s that DSLs are the **ultimate abstractions**, that is, that they generalize the ideas of [Abstraction](#) to perfection.

**Finite State Machines Remembered** The theme of finite state machines is an important one in computing, and this book has presented it several times. Here we reuse the example from [Finite State Machines](#) as the component for which we wish to design and implement a configuration DSL.

```
; An FSM is a [List-of 1Transition]
; A 1Transition is a list of two items:
; (cons FSM-State (cons FSM-State '()))
; An FSM-State is a String that specifies a color

; data examples
(define fsm-traffic
 '(("red" "green") ("green" "yellow") ("yellow" "red")))

; FSM FSM-State -> FSM-State
; matches the keys pressed by a player with the given FSM
(define (simulate state0 transitions)
 (big-bang state0 ; FSM-State
 [to-draw
 (lambda (current)
 (square 100 "solid" current))]
 [on-key
 (lambda (current key-event)
 (find transitions current)))))

; [X Y] [List-of [List X Y]] X -> Y
; finds the matching Y for the given X in alist
(define (find alist x)
 (local ((define fm (assoc x alist)))
 (if (cons? fm) (second fm) (error "not found"))))
```

Figure 129: Finite state machines, revisited

For convenience, [figure 129](#) 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 378.** Modify the rendering function so that it overlays the name of the state onto the colored square.

**Exercise 379.** Formulate test cases for `find`.

**Exercise 380.** Reformulate the data definition for `1Transition` so that it is possible to restrict transitions to certain keystrokes. 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 229](#) for the original exercise statement.

**Configurations** The FSM simulation function uses two arguments, which jointly describe a machine. Rather than teach a potential “customer” how to open an 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 129](#). 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 `FSMs`:

```

; An XMachine is a nested list of this shape:
; `(machine ((initial ,FSM-State) [List-of X1T]))
; An X1T is a nested list of this shape:
; `(action ((state ,FSM-State) (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 381.** The definitions of [XMachine](#) and [X1T](#) use quote, which is highly inappropriate for novice program designers. Rewrite them first to use [list](#) and then [cons](#).

**Exercise 382.** Formulate an XML configuration for the BW machine, which switches from white to black and back for every key event. Translate the XML configuration into an [XMachine](#) representation. See [exercise 227](#) 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 suggests a complete outline:

```

; XMachine -> FSM-State
; simulates an FSM via the given configuration
(define (simulate-xmachine xm)
 (simulate))

```

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 [X1Ts](#) into a list of [1Transitions](#):

```
| (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 xm0

(check-expect (xm-state0 xm0) "red")

```

```

(define (xm-state0 xm0)
 (find-attr (xexpr-attr xm0) 'initial))

; XMachine -> [List-of 1Transition]
; extracts the transition table from xm

(check-expect (xm->transitions xm0) fsm-traffic)

(define (xm->transitions xm)
 (local (; X1T -> 1Transition
 (define (xaction->action xa)
 (list (find-attr (xexpr-attr xa) 'state)
 (find-attr (xexpr-attr xa) 'next))))
 (map xaction->action (xexpr-content xm))))

```

Figure 130: 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-attr` and then retrieves the value of the '`initial`' attribute.

Let's then turn to `xm->transitions`, which translates the transitions inside of an [XMachine](#) configuration into a transition table:

```

; XMachine -> [List-of 1Transition]
; extracts & translates the transition table from xm
(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 [X1Ts](#). 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 a use of `map`:

```

(define (xm->transitions xm)
 (local (; X1T -> 1Transition
 (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 130](#) displays the complete solution. Here 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 383.** Run the code in figure 130 with the BW Machine configuration from exercise 382.

```
machine-configuration.xml
```

```
<machine initial="red">
 <action state="red" next="green" />
 <action state="green" next="yellow" />
 <action state="yellow" next="red" />
</machine>
```

Figure 131: A file with a machine configuration

## 22.4 Reading XML

Systems administrators expect that sophisticated applications read configuration programs from a file or possibly from some place on the web. In ISL+ your programs can retrieve (some) XML information.

This section uses *2htdp/batch-io*, *2htdp/universe*, and *2htdp/image* libraries.

Figure 132 shows the relevant excerpt from the teachpack. For consistency, the figure uses the suffix .v3 for its XML representation, including those data definitions for which there is no version 2:

```
; An Xexpr.v3 is one of:
; - Symbol
; - String
; - Number
; - (cons Symbol (cons Attribute*.v3 [List-of Xexpr.v3]))
; - (cons Symbol [List-of Xexpr.v3])
;
; An Attribute*.v3 is a [List-of Attribute.v3].
;
; An Attribute.v3 is a list of two items:
; (list Symbol String)
```

```
; Any -> Boolean
; is x an Xexpr.v3
; effect displays bad piece if x is not an Xexpr.v3
(define (xexpr? x) ...)

; String -> Xexpr.v3
; produces the first XML element in file f
(define (read-xexpr f) ...)

; String -> Boolean
; #false, if this url returns a '404'; #true otherwise
(define (url-exists? u) ...)

; String -> [Maybe Xexpr.v3]
; retrieves the first XML (HTML) element from URL u
```

```

; #false if (not (url-exists? u))
(define (read-plain-xexpr/web u) ...)

; String -> [Maybe Xexpr.v3]
; retrieves the first XML (HTML) element from URL u
; #false if (not (url-exists? u))
(define (read-xexpr/web u) ...)

```

Figure 132: Reading X-expressions

Assume we have the file in [figure 131](#). If the `2htdp/batch-io` library is required, a program can read the element with `read-plain-xexpr`. The function retrieves the XML element in a format that matches the `XMachine` data definition. A function for retrieving XML elements from the web is also available in the teachpack. Try this in DrRacket:

```

> (read-plain-xexpr/web
 (string-append
 "http://www.ccs.neu.edu/"
 "home/matthias/"
 "HtDP2e/Files/machine-configuration.xml"))

```

If your computer is connected to the web, this expression retrieves our standard machine configuration.

Reading files or web pages introduces an entirely novel idea into our computational model. As [Intermezzo 1: Beginning Student Language](#) explains, a BSL program is evaluated in the same manner in which you evaluate variable expressions in algebra. Function definitions are also treated just like in algebra. Indeed, most algebra courses introduce conditional function definitions, meaning `cond` does not pose any challenges either. Finally, while ISL+ introduces functions as values, the evaluation model remains fundamentally the same.

One essential property of this computational model is that no matter how often you call a function `f` on some argument(s) `a ...`

```
(f a ...)
```

the answer remains the same. The introduction of `read-file`, `read-xexpr`, and their relatives destroys this property, however. The problem is that files and web sites may change over time so that every time a program reads files or web sites it may get a new result.

Consider the idea of looking up the stock price of a company. Point your browser to `google.com/finance` or any other such financial web site and enter the name of your favorite company, say, Ford. In response, the site will display the current price of the company's stock and other information—for example, how much the price has changed since the last time it was posted, the current time, and many other facts and ads. The important point is that as you reload this page over the course of a day or a week, some of the information on this web page will change.

An alternative to looking up such company information manually is to write a small program that retrieves such information on a regular basis, say, every 15 seconds. With ISL you can write a world program that performs this task. You would launch it like this:

```
> (stock-alert "Ford")
```

to see a world window that displays an image like the following:



17.09 +.06

To develop such a program requires skills beyond normal program design. First, you need to investigate how the web site formats its information. In the case of Google's financial service page, an inspection of the web source code shows the following pattern near the top:

```
<meta content="17.09" itemprop="price" />
<meta content="+0.07" itemprop="priceChange" />
<meta content="0.41" itemprop="priceChangePercent" />
<meta content="2013-08-12T16:59:06Z" itemprop="quoteTime" />
<meta content="NYSE real-time data" itemprop="dataSource" />
```

If we had a function that could search an [Xexpr.v3](#) and extract (the representation of XML) meta elements with the attribute value "price" and "priceChange", the rest of stock-alert would be straightforward.

```
(define PREFIX "https://www.google.com/finance?q=")
(define SIZE 22) ; font size

(define-struct data [price delta])
; A StockWorld is a structure: (make-data String String)

; String -> StockWorld
; retrieves the stock price of co and its change every 15s
(define (stock-alert co)
 (local ((define url (string-append PREFIX co)))
 ; [StockWorld -> StockWorld]
 (define (retrieve-stock-data __w)
 (local ((define x (read-xexpr/web url)))
 (make-data (get x "price")
 (get x "priceChange"))))
 ; StockWorld -> Image
 (define (render-stock-data w)
 (local (; [StockWorld -> String] -> Image
 (define (word sel col)
 (text (sel w) SIZE col)))
 (overlay (beside (word data-price 'black)
 (text " " SIZE 'white)
 (word data-delta 'red))
 (rectangle 300 35 'solid 'white))))))
 (big-bang (retrieve-stock-data 'no-use)
 [on-tick retrieve-stock-data 15]
 [to-draw render-stock-data])))
```

Figure 133: Web data as an event

Figure 133 displays the core of the program. The design of get is left to the exercises because its workings are all about intertwined data.

As the figure shows, the main function defines two local ones: a clock-tick handler and a rendering function. The `big-bang` specification requests that the clock tick every 15 seconds.

When the clock ticks, ISL+ applies `retrieve-stock-data` to the current world, which it ignores. Instead, the function visits the web site via `read-xexpr/web` and extracts the appropriate information with `get`. Thus, the new world is created from newly available information on the web, not some local data.

**Exercise 384.** [Figure 133](#) mentions `read-xexpr/web`. See [figure 132](#) for its signature and purpose statement and then read its documentation to determine the difference to its “plain” relative.

[Figure 133](#) is also missing several important pieces, in particular the interpretation of data and purpose statements for all the locally defined functions. Formulate the missing pieces so that you get to understand the program.

**Exercise 385.** Look up the current stock price for your favorite company at Google’s financial service page. If you don’t favor a company, pick Ford. Then save the source code of the page as a file in your working directory. Use `read-xexpr` in DrRacket to view the source as an [Xexpr.v3](#).

**Exercise 386.** Here is the `get` function:

```
; Xexpr.v3 String -> String
; retrieves the value of the "content" attribute
; from a 'meta element that has attribute "itemprop"
; with value s
(check-expect
 (get '(meta ((content "+1") (itemprop "F")))) "F")
 "+1")

(define (get x s)
 (local ((define result (get-xexpr x s)))
 (if (string? result)
 result
 (error "not found"))))
```

It assumes the existence of `get-xexpr`, a function that searches an arbitrary [Xexpr.v3](#) for the desired attribute and produces [\[Maybe String\]](#).

Formulate test cases that look for other values than “`F`” and that force `get` to signal an error.

Design `get-xexpr`. Derive functional examples for this function from those for `get`. Generalize these examples so that you are confident `get-xexpr` can traverse an arbitrary [Xexpr.v3](#). Finally, formulate a test that uses the web data saved in [exercise 385](#).

---

## 23 Simultaneous Processing

Some functions have to consume two arguments that belong to classes with nontrivial data definitions. How to design such functions depends on the relationship between the arguments. First, one of the arguments may have to be treated as if it were atomic. Second, it is possible that the function must process the two arguments in lockstep. Finally, the function may process the given data in accordance to all possible cases. This chapter illustrates the three

cases with examples and provides an augmented design recipe. The last section discusses the equality of compound data.

## 23.1 Processing Two Lists Simultaneously: Case 1

Consider the following signature, purpose statement, and header:

```
; [List-of Number] [List-of Number] -> [List-of Number]
; replaces the final '() in front with end
(define (replace-eol-with front end)
 front)
```

The signature says that the function consumes two lists. Let's see how the design recipe works in this case.

We start by working through examples. If the first argument is '(), replace-eol-with must produce the second one, no matter what it is:

```
(check-expect (replace-eol-with '() '(a b)) '(a b))
```

In contrast, if the first argument is not '(), the purpose statement requires that we replace '() at the end of front with end:

```
(check-expect (replace-eol-with (cons 1 '()) '(a)))
 (cons 1 '(a)))
(check-expect (replace-eol-with
 (cons 2 (cons 1 '())))
 '(a))
 (cons 2 (cons 1 '(a))))
```

The purpose statement and the examples suggest that as long as the second argument is a list, the function does not need to know anything about it. By implication, its template should be that of a list-processing function with respect to the first argument:

```
(define (replace-eol-with front end)
 (cond
 [(empty? front) ...]
 [else
 (... (first front) ...
 ... (replace-eol-with (rest front) end) ...)]))
```

Let's fill the gaps in the template following the fifth step of the design recipe. If `front` is '(), `replace-eol-with` produces `end`. If `front` is not '(), we must recall what the template expressions compute:

- `(first front)` evaluates to the first item on the list, and
- `(replace-eol-with (rest front) end)` replaces the final '() in `(rest front)` with `end`.

Stop! Use the table method to understand what these bullets mean for the running example.

From here it is a small step to the complete definition:

```
(define (replace-eol-with front end)
```

```
(cond
 [(empty? front) end]
 [else
 (cons (first front)
 (replace-eol-with (rest front) end))]))
```

**Exercise 387.** Design `cross`. The function consumes a list of symbols and a list of numbers and produces all possible ordered pairs of symbols and numbers. That is, when given '(`a` `b` `c`) and '(`1` `2`), the expected result is '((`a` `1`) (`a` `2`) (`b` `1`) (`b` `2`) (`c` `1`) (`c` `2`)).

## 23.2 Processing Two Lists Simultaneously: Case 2

[Functions that Produce Lists](#) presents the function `wages*`, which computes the weekly wages of some workers given their work hours. It consumes a list of numbers, which represents the hours worked per week, and produces a list of numbers, which are the corresponding weekly wages. While the problem assumes that all employees received the same pay rate, even a small company pays its workers differentiated wages.

Here we look at a slightly more realistic version. The function now consumes **two** lists: the list of hours worked and the list of corresponding hourly wages. We translate this revised problem into a revised header:

```
; [List-of Number] [List-of Number] -> [List-of Number]
; multiplies the corresponding items on
; hours and wages/h
; assume the two lists are of equal length
(define (wages*.v2 hours wages/h)
 '())
```

Making up examples is straightforward:

```
(check-expect (wages*.v2 '() '()) '())
(check-expect (wages*.v2 (list 5.65) (list 40))
 (list 226.0))
(check-expect (wages*.v2 '(5.65 8.75) '(40.0 30.0))
 '(226.0 262.5))
```

As required, all three examples use lists of equal length.

The assumption concerning the inputs can also be exploited for the development of the template. More concretely, the condition says that (`empty?` `hours`) is true when (`empty?` `wages/h`) is true, and furthermore, (`cons?` `hours`) is true when (`cons?` `wages/h`) is true. It is thus acceptable to use a template for one of the two lists:

```
(define (wages*.v2 hours wages/h)
 (cond
 [(empty? hours) ...]
 [else
 (... (first hours)
 ... (first wages/h) ...
 ... (wages*.v2 (rest hours) (rest wages/h))))])
```

In the first `cond` clause, both `hours` and `wages/h` are '`()`'. Hence no selector expressions are needed. In the second clause, both `hours` and `wages/h` are constructed lists, which means we need four selector expressions. Finally, because the last two are lists of equal length, they make up a natural candidate for the natural recursion of `wages*.v2`.

The only unusual aspect of this template is that the recursive application consists of two expressions, both selector expressions for the two arguments. But, this idea directly follows from the assumption.

From here, it is a short step to a complete function definition:

```
(define (wages*.v2 hours wages/h)
 (cond
 [(empty? hours) '()]
 [else
 (cons
 (weekly-wage (first hours) (first wages/h))
 (wages*.v2 (rest hours) (rest wages/h))))])
```

The first example implies that the answer for the first `cond` clause is '`()`'. In the second one, we have three values available:

1. `(first hours)`, which represents the first number of weekly hours;
2. `(first wages/h)`, which is the first pay rate; and
3. `(wages*.v2 (rest hours) (rest wages/h))`, which, according to the purpose statement, computes the list of weekly wages for the remainders of the two lists.

Now we just need to combine these values to get the final answer. As suggested by the examples, we must compute the weekly wage for the first employee and construct a list from that wage and the rest of the wages:

```
(cons (weekly-wage (first hours) (first wages/h))
 (wages*.v2 (rest hours) (rest wages/h)))
```

The auxiliary function `weekly-wage` uses the number of hours worked and the pay rate to compute the weekly wage for one worker:

```
; Number Number -> Number
; computes the weekly wage from pay-rate and hours
(define (weekly-wage pay-rate hours)
 (* pay-rate hours))
```

Stop! Which function do you need to use if you wish to compute the wages for one worker? Which function do you need to change if you wish to deal with income taxes?

**Exercise 388.** In the real world, `wages*.v2` consumes lists of employee structures and lists of work records. An employee structure contains an employee's name, social security number, and pay rate. A work record also contains an employee's name and the number of hours worked in a week. The result is a list of structures that contain the name of the employee and the weekly wage.

Modify the program in this section so that it works on these realistic versions of data. Provide the necessary structure type definitions and data definitions. Use the design recipe to guide the modification process.

**Exercise 389.** Design the `zip` function, which consumes a list of names, represented as strings, and a list of phone numbers, also strings. It combines those equally long lists into a list of phone records:

```
(define-struct phone-record [name number])
; A PhoneRecord is a structure:
; (make-phone-record String String)
```

Assume that the corresponding list items belong to the same person.

---

### 23.3 Processing Two Lists Simultaneously: Case 3

Here is a third type of problem:

**Sample Problem** Given a list of symbols `lOS` and a natural number `n`, the function `list-pick` extracts the `n`th symbol from `lOS`; if there is no such symbol, it signals an error.

The question is how well the recipe works for the design of `list-pick`.

While the data definition for a list of symbols is fairly familiar by now, recall the class of natural numbers from [Natural Numbers](#):

```
; N is one of:
; - 0
; - (add1 N)
```

Now we can proceed to the second step:

```
; [List-of Symbol] N -> Symbol
; extracts the nth symbol from l;
; signals an error if there is no such symbol
(define (list-pick l n)
 'a)
```

Both lists of symbols and natural numbers are classes with complex data definitions. This combination makes the problem nonstandard, meaning we must pay attention to every detail for every step of the design recipe.

At this point, we usually pick some input examples and figure out what the desired output is. We start with inputs for which the function has to work flawlessly: `'(a b c)` and `2`. For a list of three symbols and the index `2`, `list-pick` must return a symbol. The question is whether it is `'b` or `'c`. In grade school, you would have counted `1`, `2`, and picked `'b` without a first thought. But this is computer science, not grade school. Here people start counting from `0`, meaning that `'c` is an equally appropriate choice. And indeed, this is the choice we use:

```
(check-expect (list-pick '(a b c) 2) 'c)
```

Now that we have eliminated this fine point of `list-pick`, let's look at the actual problem, the choice of inputs. The goal of the example step is to cover the input space as much as possible. We do so by picking one input per clause in the description of complex forms of data. Here this procedure suggests we pick at least two elements from each class because each data definition has two clauses. We choose '`()`' and `(cons 'a '())` for the first argument, and `0` and `3` for the latter. Two choices per argument means four examples total; after all, there is no immediately obvious connection between the two arguments and no restriction in the signature.

As it turns out, only one of these pairings produces a proper result; the remaining ones choose a position that does not exist because the list doesn't contain enough symbols:

```
(check-error (list-pick '() 0) "list too short")
(check-expect (list-pick (cons 'a '()) 0) 'a)
(check-error (list-pick '() 3) "list too short")
```

The function is expected to signal an error, and we pick our favorite message here.

Stop! Put these fragments into DrRacket's definitions area and run the partial program.

The discussion on examples indicates that there are indeed four independent cases that we must inspect for the design of the function. One way to discover these cases is to arrange the conditions for each of the clauses into a two-dimensional table:

	<code>(empty? l)</code>	<code>(cons? l)</code>
<code>(= n 0)</code>		
<code>(&gt; n 0)</code>		

The horizontal dimension of the table lists those questions that `list-pick` must ask about lists; the vertical dimension lists the questions about natural numbers. By this arrangement, we naturally get four squares, where each represents the case when both the conditions on the horizontal and the vertical axis are true.

Our table suggests that the `cond` for the function template has four clauses. We can figure out the appropriate condition for each of these clauses by `and-ing` the horizontal and vertical condition for each box in the table:

	<code>(empty? l)</code>	<code>(cons? l)</code>
<code>(= n 0)</code>	<code>(and (= n 0) (empty? l))</code>	<code>(and (= n 0) (cons? l))</code>
<code>(&gt; n 0)</code>	<code>(and (&gt; n 0) (empty? l))</code>	<code>(and (&gt; n 0) (cons? l))</code>

The `cond` outline of the template is merely a translation of this table into a conditional:

```
(define (list-pick l n)
 (cond
 [(and (= n 0) (empty? l)) ...]
 [(and (> n 0) (empty? l)) ...]
 [(and (= n 0) (cons? l)) ...]
 [(and (> n 0) (cons? l)) ...]))
```

As always, the `cond` expression allows us to distinguish the four possibilities and to focus on each individually as we add selector expressions to each `cond` clause:

```
(define (list-pick l n)
 (cond
 [(and (= n 0) (empty? l))
 ...]
 [(and (> n 0) (empty? l))
 (... (sub1 n) ...)]
 [(and (= n 0) (cons? l))
 (... (first l) ... (rest l) ...)]
 [(and (> n 0) (cons? l))
 (... (sub1 n) ... (first l) ... (rest l) ...))]))
```

The first argument, `l`, is a list, and a template's `cond` clause for non-empty lists contains two selector expressions. The second argument, `n`, belongs to `N`, and the template's `cond` clause for non-`0` numbers needs only one selector expression. In those cases where either `(empty? l)` or `(= n 0)` holds, the respective argument is atomic and there is no need for a corresponding selector expression.

The final step of the template construction demands that we annotate the template with recursions where the results of selector expressions belong to the same class as the inputs. For this first example, we focus on the last `cond` clause, which contains selector expressions for both arguments. It is, however, unclear how to form the natural recursions. If we disregard the purpose of the function, there are three possible recursions:

1. `(list-pick (rest l) (sub1 n))`
2. `(list-pick l (sub1 n))`
3. `(list-pick (rest l) n)`

Each one represents a feasible combination of the available expressions. Since we cannot know which one matters or whether all three matter, we move on to the next development stage.

```
; [List-of Symbol] N -> Symbol
; extracts the nth symbol from l;
; signals an error if there is no such symbol
(define (list-pick l n)
 (cond
 [(and (= n 0) (empty? l))
 (error 'list-pick "list too short")]
 [(and (> n 0) (empty? l))
 (error 'list-pick "list too short")]
 [(and (= n 0) (cons? l)) (first l)]
 [(and (> n 0) (cons? l)) (list-pick (rest l) (sub1 n))]))
```

Figure 134: Indexing into a list

Following the design recipe for the fifth step, let's analyze each `cond` clause in the template and decide what a proper answer is:

1. If `(and (= n 0) (empty? l))` holds, `list-pick` must pick the first symbol from an empty list, which is impossible. The answer must be an error signal.

2. If `(and (> n 0) (empty? l))` holds, `list-pick` is again asked to pick a symbol from an empty list.
3. If `(and (= n 0) (cons? l))` holds, `list-pick` is supposed to produce the first symbol from `l`. The selector expression `(first l)` is the answer.
4. If `(and (> n 0) (cons? l))` holds, we must analyze what the available expressions compute. As we have seen, it is a good idea to work through an existing example for this step. We pick a shortened variant of the first example:

```
| (check-expect (list-pick '(a b) 1) 'b)
```

Here is what the three natural recursions compute with these values:

- a. `(list-pick '(b) 0)` produces '`b`';
- b. `(list-pick '(a b) 0)` evaluates to '`a`', the wrong answer;
- c. and `(list-pick '(b) 1)` signals an error.

From this, we conclude that `(list-pick (rest l) (sub1 n))` computes the desired answer in the last `cond` clause.

**Exercise 390.** Design the function `tree-pick`. The function consumes a tree of symbols and a list of directions:

```
(define-struct branch [left right])

; A TOS is one of:
; – Symbol
; – (make-branch TOS TOS)

; A Direction is one of:
; – 'left
; – 'right

; A list of Directions is also called a path.
```

Clearly a `Direction` tells the function whether to choose the left or the right branch in a nonsymbolic tree. What is the result of the `tree-pick` function? Don't forget to formulate a full signature. The function signals an error when given a symbol and a non-empty path.

## 23.4 Function Simplification

The `list-pick` function in [figure 134](#) is far more complicated than necessary. The first two `cond` clauses signal an error. That is, if either

```
| (and (= n 0) (empty? alos))
```

or

```
| (and (> n 0) (empty? alos))
```

holds, the answer is an error. We can translate this observation into code:

```
(define (list-pick alos n)
 (cond
 [(or (and (= n 0) (empty? alos))
 (and (> n 0) (empty? alos)))
 (error 'list-pick "list too short")]
 [(and (= n 0) (cons? alos)) (first alos)]
 [(and (> n 0) (cons? alos))
 (list-pick (rest alos) (sub1 n))]))
```

To simplify this function even more, we need to get acquainted with algebraic laws concerning Booleans:

These equations are known as de Morgan's laws.

```
(or (and bexp1 a-bexp) == (and (or bexp1 bexp2)
 (and bexp2 a-bexp))
```

A similar law applies when the sub-expressions of the `ands` are swapped. Applying these laws to `list-pick` yields this:

```
(define (list-pick n alos)
 (cond
 [(and (or (= n 0) (> n 0)) (empty? alos))
 (error 'list-pick "list too short")]
 [(and (= n 0) (cons? alos)) (first alos)]
 [(and (> n 0) (cons? alos))
 (list-pick (rest alos) (sub1 n))]))
```

Now consider `(or (= n 0) (> n 0))`. It is always `#true` because `n` belongs to `N`. Since `(and #true (empty? alos))` is equivalent to `(empty? alos)`, we can rewrite the function again:

```
(define (list-pick alos n)
 (cond
 [(empty? alos) (error 'list-pick "list too short")]
 [(and (= n 0) (cons? alos)) (first alos)]
 [(and (> n 0) (cons? alos))
 (list-pick (rest alos) (sub1 n))]))
```

This last definition is already significantly simpler than the definition in [figure 134](#), but we can do even better than this. Compare the first condition in the latest version of `list-pick` with the second and third. Since the first `cond` clause filters out all those cases when `alos` is empty, `(cons? alos)` in the last two clauses is always going to evaluate to `#true`. If we replace the condition with `#true` and simplify the `and` expressions again, we get a three-line version of `list-pick`

```
; list-pick: [List-of Symbol] N[>= 0] -> Symbol
; determines the nth symbol from alos, counting from 0;
; signals an error if there is no nth symbol
(define (list-pick alos n)
 (cond
 [(empty? alos) (error 'list-pick "list too short")]
```

```
[(= n 0) (first alos)]
[(> n 0) (list-pick (rest alos) (sub1 n))])
```

Figure 135: Indexing into a list, simplified

[Figure 135](#) displays this simplified version of `list-pick`. While it is far simpler than the original, it is important to understand that we designed the original in a systematic manner and that we were able to transform the first into the second with well-established algebraic laws. We can therefore trust this simple version. If we try to find the simple versions of functions directly, we sooner or later fail to take care of a case in our analysis, and we are guaranteed to produce flawed programs.

**Exercise 391.** Design `replace-eol-with` using the strategy of [Processing Two Lists Simultaneously: Case 3](#).

**Start from the tests.** Simplify the result systematically.

**Exercise 392.** Simplify the function `tree-pick` from [exercise 390](#).

## 23.5 Designing Functions that Consume Two Complex Inputs

The proper approach to designing functions of two (or more) complex arguments is to follow the general recipe. You must conduct a data analysis and define the relevant classes of data. If the use of parametric definitions such as `List-of` and short-hand examples such as `'(1 b &)` confuses you, expand them so that the constructors become explicit. Next you need a function signature and purpose. At this point, you can think ahead and decide which of the following three situations you are facing:

1. If one of the parameters plays a dominant role, think of the other as an atomic piece of data as far as the function is concerned.
2. In some cases the parameters range over the same class of values and must have the same size. For example, two lists must have the same length, or two web pages must have the same length and where one of them contains an embedded page, the other one does, too. If the two parameters have this equal status and the purpose suggests that they are processed in a synchronized manner, you choose one parameter, organize the function around it, and traverse the other in a parallel manner.
3. If there is no obvious connection between the two parameters, you must analyze all possible cases with examples. Then use this analysis to develop the template, especially the recursive parts.

Once you decide that a function falls into the third category, develop a two-dimensional table to make sure that no case falls through the cracks. Let's use a nontrivial pair of data definitions to explain this idea again:

```
; An LOD is one of: ; A TID is one of:
; - '() ; - Symbol
; - (cons Direction LOD) ; - (make-binary TID TID)
 ; - (make-with TID Symbol TID)
```

The left data definition is the usual list definition; the right one is a three-clause variant of `TOS`. It uses two structure type definitions:

```
(define-struct with [lft info rght])
```

```
| (define-struct binary [lft rght])
```

Assuming the function consumes an **LOD** and a **TID**, the table that you should come up with has this shape:

	(empty? l)    (cons? l)
(symbol? t)	
(binary? t)	
(with? t)	

Along the horizontal direction we enumerate the conditions that recognize the sub-classes for the first parameter, here **LOD**, and along the vertical direction we enumerate the conditions for the second parameter, **TID**.

The table guides the development of both the function examples and the function template. As explained, the examples must cover all possible cases; that is, there must be at least one example for each cell in the table. Similarly, the template must have one **cond** clause per cell; its condition combines the horizontal and the vertical conditions in an **and** expression. Each **cond** clause, in turn, must contain all feasible selector expressions for both parameters. If one of the parameters is atomic, there is no need for a selector expression. Finally, you need to be aware of the feasible natural recursions. In general, all possible combinations of selector expressions (and optionally, atomic arguments) are candidates for a natural recursion. Because we can't know which ones are necessary and which ones aren't, we keep them in mind for the coding step.

In summary, the design of multiparameter functions is just a variation on the old design-recipe theme. The key idea is to translate the data definitions into a table that shows all feasible and interesting combinations. The development of function examples and the template exploit the table as much as possible.

---

## 23.6 Finger Exercises: Two Inputs

**Exercise 393.** Figure 62 presents two data definitions for finite sets. Design the **union** function for the representation of finite sets of your choice. It consumes two sets and produces one that contains the elements of both.

Design **intersect** for the same set representation. It consumes two sets and produces the set of exactly those elements that occur in both.

**Exercise 394.** Design **merge**. The function consumes two lists of numbers, sorted in ascending order. It produces a single sorted list of numbers that contains all the numbers on both inputs lists. A number occurs in the output as many times as it occurs on the two input lists together.

**Exercise 395.** Design **take**. It consumes a list **l** and a natural number **n**. It produces the first **n** items from **l** or all of **l** if it is too short.

Design **drop**. It consumes a list **l** and a natural number **n**. Its result is **l** with the first **n** items removed or just **'()** if **l** is too short.

; An HM-Word is a [List-of Letter or "\_"]  
; interpretation "\_" represents a letter to be guessed

```

; HM-Word N -> String
; runs a simplistic hangman game, produces the current state
(define (play the-pick time-limit)
 (local ((define the-word (explode the-pick))
 (define the-guess (make-list (length the-word) "_"))
 ; HM-Word -> HM-Word
 (define (do-nothing s) s)
 ; HM-Word KeyEvent -> HM-Word
 (define (checked-compare current-status ke)
 (if (member? ke LETTERS)
 (compare-word the-word current-status ke)
 current-status)))
 (implode
 (big-bang the-guess ; HM-Word
 [to-draw render-word]
 [on-tick do-nothing 1 time-limit]
 [on-key checked-compare]))))

; HM-Word -> Image
(define (render-word w)
 (text (implode w) 22 "black"))

```

Figure 136: A simple hangman game

**Exercise 396.** Hangman is a well-known guessing game. One player picks a word, the other player gets told how many letters the word contains. The latter picks a letter and asks the first player whether and where this letter occurs in the chosen word. The game is over after an agreed-upon time or number of rounds.

Figure 136 presents the essence of a time-limited version of the game. See [Local Definitions Add Expressive Power](#) for why `checked-compare` is defined locally.

The goal of this exercise is to design `compare-word`, the central function. It consumes the word to be guessed, a word `s` that represents how much/little the guessing player has discovered, and the current guess. The function produces `s` with all `"_"` where the guess revealed a letter.

Once you have designed the function, run the program like this:

```

(define LOCATION "/usr/share/dict/words") ; on OS X
(define AS-LIST (read-lines LOCATION))
(define SIZE (length AS-LIST))

(play (list-ref AS-LIST (random SIZE)) 10)

```

See [figure 74](#) for an explanation. Enjoy and refine as desired!

**Exercise 397.** In a factory, employees punch time cards as they arrive in the morning and leave in the evening. Electronic time cards contain an employee number and record the number of hours worked per week. Employee records always contain the name of the employee, an employee number, and a pay rate.

Design `wages*.v3`. The function consumes a list of employee records and a list of time-card records. It produces a list of wage records, which contain the name and weekly wage of an

employee. The function signals an error if it cannot find an employee record for a time card or vice versa.

**Assumption** There is at most one time card per employee number.

**Exercise 398.** A linear combination is the sum of many linear terms, that is, products of variables and numbers. The latter are called coefficients in this context. Here are some examples:

$$5 \cdot x \quad 5 \cdot x + 17 \cdot y \quad 5 \cdot x + 17 \cdot y + 3 \cdot z$$

In all examples, the coefficient of  $x$  is 5, that of  $y$  is 17, and the one for  $z$  is 3.

If we are given values for variables, we can determine the value of a polynomial. For example, if  $x = 10$ , the value of  $5 \cdot x$  is 50; if  $x = 10$  and  $y = 1$ , the value of  $5 \cdot x + 17 \cdot y$  is 67; and if  $x = 10$ ,  $y = 1$ , and  $z = 2$ , the value of  $5 \cdot x + 17 \cdot y + 3 \cdot z$  is 73.

There are many different representations of linear combinations. We could, for example, represent them with functions. An alternative representation is a list of its coefficients. The above combinations would be represented as:

```
(list 5)
(list 5 17)
(list 5 17 3)
```

This choice of representation assumes a fixed order of variables.

Design `value`. The function consumes two equally long lists: a linear combination and a list of variable values. It produces the value of the combination for these values.

**Exercise 399.** Louise, Jane, Laura, Dana, and Mary decide to run a lottery that assigns one gift recipient to each of them. Since Jane is a developer, they ask her to write a program that performs this task in an impartial manner. Of course, the program must not assign any of the sisters to herself.

Here is the core of Jane's program:

```
; [List-of String] -> [List-of String]
; picks a random non-identity arrangement of names
(define (gift-pick names)
 (random-pick
 (non-same names (arrangements names)))))

; [List-of String] -> [List-of [List-of String]]
; returns all possible permutations of names
; see exercise 213
(define (arrangements names)
 ...)
```

It consumes a list of names and randomly picks one of those permutations that do not agree with the original list at any place.

Your task is to design two auxiliary functions:

```
; [NList-of X] -> X
```

```

; returns a random item from the list
(define (random-pick l)
 (first l))

; [List-of String] [List-of [List-of String]]
; ->
; [List-of [List-of String]]
; produces the list of those lists in ll that do
; not agree with names at any place
(define (non-same names ll)
 ll)

```

Recall that `random` picks a random number; see [exercise 99](#).

**Exercise 400.** Design the function `DNAprefix`. The function takes two arguments, both lists of '`a`', '`c`', '`g`', and '`t`', symbols that occur in DNA descriptions. The first list is called a pattern, the second one a search string. The function returns `#true` if the pattern is identical to the initial part of the search string; otherwise it returns `#false`.

Also design `DNAdelta`. This function is like `DNAprefix` but returns the first item in the search string beyond the pattern. If the lists are identical and there is no DNA letter beyond the pattern, the function signals an error. If the pattern does not match the beginning of the search string, it returns `#false`. The function must not traverse either of the lists more than once.

Can `DNAprefix` or `DNAdelta` be simplified?

**Exercise 401.** Design `sexp=?`, a function that determines whether two S-expressions are equal. For convenience, here is the data definition in condensed form:

```

; An S-expr (S-expression) is one of:
; – Atom
; – [List-of S-expr]
;
; An Atom is one of:
; – Number
; – String
; – Symbol

```

Whenever you use `check-expect`, it uses a function like `sexp=?` to check whether the two arbitrary values are equal. If not, the check fails and `check-expect` reports it as such.

**Exercise 402.** Reread [exercise 354](#). Explain the reasoning behind our hint to think of the given expression as an atomic value at first.

---

## 23.7 Project: Database

Many software applications use a database to keep track of data. Roughly speaking, a database is a table that comes with an explicitly stated organization rule. The former is the *content*; the latter is called a *schema*. [Figure 137](#) shows two examples. Each table consists of two parts: the schema above the line and the content below.

Let's focus on the table on the left. It has three columns and four rows. Each column comes with a two-part rule:

This section pulls together knowledge from all four parts of the book.

1. the rule in the left-most column says that the label of the column is “Name” and that every piece of data in this column is a [String](#);
2. the middle column is labeled “Age” and contains [Integers](#); and
3. the label of the right-most one is “Present”; its values are [Boolean](#).

Stop! Explain the table on the right in the same way.

Name	Age	Present	Present	Description
<a href="#">String</a>	<a href="#">Integer</a>	<a href="#">Boolean</a>	<a href="#">Boolean</a>	<a href="#">String</a>
"Alice"	35	#true	#true	"presence"
"Bob"	25	#false	#false	"absence"
"Carol"	30	#true		
"Dave"	32	#false		

Figure 137: Databases as tables

Computer scientists think of these tables as *relations*. The schema introduces terminology to refer to columns of a relation and to individual cells in a row. Each row relates a fixed number of values; the collection of all rows makes up the entire relation. In this terminology, the first row of the left table in [figure 137](#) relates "Alice" to 35 and #true. Furthermore, the first cell of a row is called the “Name” cell, the second the “Age” cell, and the third one the “Present” cell.

In this section, we represent databases via structures and lists:

```
(define-struct db [schema content])
; A DB is a structure:
; (make-db Schema Content)

; A Schema is a [List-of Spec]
; A Spec is a [List Label Predicate]
; A Label is a String
; A Predicate is a [Any -> Boolean]

; A (piece of) Content is a [List-of Row]
; A Row is a [List-of Cell]
; A Cell is Any
; constraint cells do not contain functions

; integrity constraint In (make-db sch con),
; for every row in con,
; (I1) its length is the same as sch's, and
; (I2) its ith Cell satisfies the ith Predicate in sch
```

Stop! Translate the databases from [figure 137](#) into the chosen data representation. Note that the content of the tables already uses ISL+ data.

```

(define school-schema
 `(("Name" ,string?)
 ("Age" ,integer?)
 ("Present" ,boolean?)))

(define school-content
 `(("Alice" 35 #true)
 ("Bob" 25 #false)
 ("Carol" 30 #true)
 ("Dave" 32 #false)))

(define school-db
 (make-db school-schema
 school-content))

(define presence-schema
 `(("Present" ,boolean?)
 ("Description" ,string?)))

(define presence-content
 `((#true "presence")
 (#false "absence")))

(define presence-db
 (make-db presence-schema
 presence-content))

```

Figure 138: Databases as ISL+ data

Figure 138 shows how to represent the two tables in figure 137 as DBs. Its left-hand side represents the schema, the content, and the database from the left-hand side table in figure 137; its right part corresponds to the right-hand side table. For succinctness, the examples use the `quasiquote` and `unquote` notation. Recall that it allows the inclusion of a value such as `boolean?` in an otherwise quoted list. If you feel uncomfortable with this notation, reformulate these examples with `list`.

**Exercise 403.** A `Spec` combines one `Label` and one `Predicate` into a list. While this choice is perfectly acceptable for a mature programmer, it violates our guideline of using a structure type when the represented information consists of a fixed number of pieces.

Here is an alternative data representation:

```

(define-struct spec [label predicate])
; Spec is a structure: (make-spec Label Predicate)

```

Use this alternative definition to represent the databases from figure 137.

**Integrity Checking** The use of databases critically relies on their integrity. Here “integrity” refers to the constraints (I1) and (I2) from the data definition. Checking database integrity is clearly a task for a function:

```

; DB -> Boolean
; do all rows in db satisfy (I1) and (I2)

(check-expect (integrity-check school-db) #true)
(check-expect (integrity-check presence-db) #true)

(define (integrity-check db)
 #false)

```

The wording of the two constraints suggests that some function has to produce `#true` for every row in the content of the given database. Expressing this idea in code calls for a use of `andmap` on the content of `db`:

```

(define (integrity-check db)

```

```
(local (; Row -> Boolean
 (define (row-integrity-check row)
 ...))
 (andmap row-integrity-check (db-content db))))
```

Following the design recipe for the use of existing abstractions, the template introduces the auxiliary function via a `local` definition.

The design of `row-integrity-check` starts from this:

```
; Row -> Boolean
; does row satisfy (I1) and (I2)
(define (row-integrity-check row)
 #false)
```

As always, the goal of formulating a purpose statement is to understand the problem. Here it says that the function checks **two** conditions. When two tasks are involved, our design guidelines call for functions and the combination of their results:

```
(and (length-of-row-check row)
 (check-every-cell row))
```

Add these functions to your wish list; their names convey their purpose.

Before we design these functions, we must contemplate whether we can compose existing primitive operations to compute the desired value. For example, we know that `(length row)` counts how many cells are in `row`. Pushing a bit more in this direction, we clearly want

```
(= (length row) (length (db-schema db)))
```

This condition checks that the length of `row` is equal to that of `db`'s schema.

Similarly, `check-every-cell` calls for checking that some function produces `#true` for every cell in the `row`. Once again, it looks like `andmap` might be called for:

```
(andmap cell-integrity-check row)
```

The purpose of `cell-integrity-check` is obviously to check constraint (I2), that is,

whether “the *i*th `Cell` satisfies the *i*th `Predicate` in `db`'s schema.”

And now we are stuck because this purpose statement refers to the relative position of the given cell in `row`. The point of `andmap` is, however, to apply `cell-integrity-check` to every cell **uniformly**.

When you are stuck, you must work through examples. For auxiliary or `local` functions, it's best to derive these examples from the ones for the main function. The first example for `integrity-check` asserts that `school-content` satisfies the integrity constraints. Clearly all rows in `school-content` have the same length as `school-schema`. The question is why a row such as

```
(list "Alice" 35 #true)
```

satisfies the predicates in the corresponding schema:

```
(list (list "Name" string?)
 (list "Age" integer?)
 (list "Present" boolean?))
```

The answer is that all three applications of the three predicates to the respective cells yields true:

```
> (string? "Alice")
#true
> (integer? 35)
#true
> (boolean? #true)
#true
```

From here, it is just a short step to see that the function must process these two lists—db's schema and the given row—in parallel.

**Exercise 404.** Design the function `andmap2`. It consumes a function `f` from two values to `Boolean` and two equally long lists. Its result is also a `Boolean`. Specifically, it applies `f` to pairs of corresponding values from the two lists, and if `f` always produces `#true`, `andmap2` produces `#true`, too. Otherwise, `andmap2` produces `#false`. In short, `andmap2` is like `andmap` but for two lists.

Stop! Solve [exercise 404](#) before reading on.

If we had `andmap2` in ISL+, checking the second condition on `row` would be straightforward:

```
(andmap2 (lambda (s c) [(second s) c])
 (db-schema db)
 row)
```

The given function consumes a `Spec` `s` from db's schema, extracts the predicate in the second position, and applies it to the given `Cell` `c`. Whatever the predicate returns is the result of the `lambda` function.

Stop again! Explain `[(second s) c]`.

As it turns out, `andmap` in ISL+ is already like `andmap2`:

```
(define (integrity-check db)
 (local (; Row -> Boolean
 ; does row satisfy (I1) and (I2)
 (define (row-integrity-check row)
 (and (= (length row)
 (length (db-schema db)))
 (andmap (lambda (s c) [(second s) c])
 (db-schema db)
 row))))
 (andmap row-integrity-check (db-content db))))
```

Stop a last time! Develop a test for which `integrity-check` must fail.

**Note on Expression Hoisting** Our definition of `integrity-check` suffers from several problems, some visible, some invisible. Clearly, the function extracts db's schema twice. With

the existing `local` definition it is possible to introduce a definition and avoid this duplication:

```
(define (integrity-check.v2 db)
 (local ((define schema (db-schema db))
 ; Row -> Boolean
 ; does row satisfy (I1) and (I2)
 (define (row-integrity-check row)
 (and (= (length row) (length schema))
 (andmap (lambda (s c) [(second s) c])
 schema
 row))))
 (andmap row-integrity-check (db-content db))))
```

We know from [Local Definitions](#) that lifting such an expression may shorten the time needed to run the integrity check. Like the definition of `inf` in [figure 100](#), the original version of `integrity-check` extracts the schema from `db` for every single row, even though it obviously stays the same.

```
(define (integrity-check.v3 db)
 (local ((define schema (db-schema db))
 (define content (db-content db))
 (define width (length schema))
 ; Row -> Boolean
 ; does row satisfy (I1) and (I2)
 (define (row-integrity-check row)
 (and (= (length row) width)
 (andmap (lambda (s c) [(second s) c])
 schema
 row))))
 (andmap row-integrity-check content)))
```

Figure 139: The result of systematic expression hoisting

**Terminology** Computer scientists speak of “*hoisting* an expression.” In a similar vein, the `row-integrity-check` function determines the length of `db`’s schema every single time it is called. The result is always the same. Hence, if we are interested in improving the performance of this function, we can use a `local` definition to name the `width` of the database content once and for all. [Figure 139](#) displays the result of hoisting `(length schema)` out of the `row-integrity-check`. For readability, this final definition also names the `content` field of `db`. **End**

**Projections and Selections** Programs need to extract data from databases. One kind of extraction is to *select* content, which is explained in [Real-World Data: iTunes](#). The other kind of extraction produces a reduced database; it is dubbed *projection*. More specifically, a projection constructs a database by retaining only certain columns from a given database.

The description of a projection suggests the following:

```
; DB [List-of Label] -> DB
; retains a column from db if its label is in labels
(define (project db labels) (make-db '() '()))
```

Given the complexity of a projection, it is best to work through an example first. Say we wish to eliminate the age column from the database on the left in [figure 137](#). Here is what this transformation looks like in terms of tables:

the original database			... eliminating the “Age” column	
Name	Age	Present	Name	Present
<b>String</b>	<b>Integer</b>	<b>Boolean</b>	<b>String</b>	<b>Boolean</b>
"Alice"	35	#true	"Alice"	#true
"Bob"	25	#false	"Bob"	#false
"Carol"	30	#true	"Carol"	#true
"Dave"	32	#false	"Dave"	#false

A natural way to articulate the example as a test reuses [figure 138](#):

```
(define projected-content
 `((("Alice" #true)
 ("Bob" #false)
 ("Carol" #true)
 ("Dave" #false))))

(define projected-schema
 `(("Name" ,string?) ("Present" ,boolean?)))

(define projected-db
 (make-db projected-schema projected-content))
; Stop! Read this test carefully. What's wrong?
(check-expect (project school-db '("Name" "Present"))
 projected-db)
```

```
(define (project db labels)
 (local ((define schema (db-schema db))
 (define content (db-content db))
 ; Spec -> Boolean
 ; does this spec belong to the new schema
 (define (keep? c) ...))
 ; Row -> Row
 ; retains those columns whose name is in labels
 (define (row-project row) ...))
 (make-db (filter keep? schema)
 (map row-project content)))
```

Figure 140: A template for project

If you run the above code in DrRacket, you get the error message

first argument of equality cannot be a function

before DrRacket can even figure out whether the test succeeds. Recall from [Functions Are Values](#) that functions are infinitely large objects and it is impossible to ensure that two functions always produce the same result when applied to the same arguments. We therefore weaken the test case:

```
(check-expect
```

```
(db-content (project school-db '("Name" "Present")))
 projected-content)
```

For the template, we again reuse existing abstractions; see [figure 140](#). The `local` expression defines two functions: one for use with `filter` for narrowing down the schema of the given database and the other for use with `map` for thinning out the content. In addition, the function again extracts and names the schema and the content from the given database.

Before we turn to the wish list, let's step back and study the decision to go with two reuses of existing abstraction. The signature tells us that the function consumes a structure and produces an element of `DB`, so

```
(local ((define schema (db-schema db))
 (define content (db-content db)))
 (make-db ... schema ...
 ... content ...))
```

is clearly called for. It is also straightforward to see that the new schema is created from the old schema and the new content from the old content. Furthermore, the purpose statement of `project` calls for the retention of only those labels that are mentioned in the second argument. Hence, the `filter` function correctly narrows down the given `schema`. In contrast, the rows per se stay except that each of them loses some cells. Thus, `map` is the proper way of processing content.

Now we can turn to the design of the two auxiliary functions. The design of `keep?` is straightforward. Here is the complete definition:

```
; Spec -> Boolean
; does this spec belong to the new schema
(define (keep? c)
 (member? (first c) labels))
```

The function is applied to a `Spec`, which combines a `Label` and a `Predicate` in a list. If the former belongs to `labels`, the given `Spec` is kept.

For the design of `row-project`, the goal is to keep those `Cells` in each `Row` of content whose name is a member of the given `labels`. Let's work through the above example. The four rows are:

```
(list "Alice" 35 #true)
(list "Bob" 25 #false)
(list "Carol" 30 #true)
(list "Dave" 32 #false)
```

Each of these rows is as long as `school-schema`:

```
(list "Name" "Age" "Present")
```

The names in the schema determine the names of the cells in the given rows. Hence, `row-project` must keep the first and third cell of each row because it is their names that are in the given `labels`.

Since `Row` is defined recursively, this matching process between the content of the `Cells` and their names calls for a recursive helper function that `row-project` can apply to the content

and the labels of the cells. Let's specify it as a wish:

```
; Row [List-of Label] -> Row
; retains those cells whose corresponding element
; in names is also in labels
(define (row-filter row names) '())
```

Using this wish, `row-project` is a one-liner:

```
(define (row-project row)
 (row-filter row (map first schema)))
```

The `map` expression extracts the names of the cells, and those names are handed to `row-filter` to extract the matching cells.

**Exercise 405.** Design the function `row-filter`. Construct examples for `row-filter` from the examples for `project`.

**Assumption** The given database passes an integrity check, meaning each row is as long as the schema and thus its list of names.

[Figure 141](#) puts all the pieces together. The function `project` has the suffix `.v1` because it calls for some improvement. The following exercises ask you to implement some of those.

```
(define (project.v1 db labels)
 (local ((define schema (db-schema db))
 (define content (db-content db))

 ; Spec -> Boolean
 ; does this column belong to the new schema
 (define (keep? c)
 (member? (first c) labels))

 ; Row -> Row
 ; retains those columns whose name is in labels
 (define (row-project row)
 (row-filter row (map first schema)))

 ; Row [List-of Label] -> Row
 ; retains those cells whose name is in labels
 (define (row-filter row names)
 (cond
 [(empty? names) '()]
 [else
 (if (member? (first names) labels)
 (cons (first row)
 (row-filter (rest row) (rest names)))
 (row-filter (rest row) (rest names))))]))
 (make-db (filter keep? schema)
 (map row-project content))))
```

Figure 141: Database projection

**Exercise 406.** The `row-project` function recomputes the labels for every row of the database's content. Does the result differ from function call to function call? If not, hoist the expression.

**Exercise 407.** Redesign `row-filter` using `foldr`. Once you have done so, you may merge `row-project` and `row-filter` into a single function. **Hint** The `foldr` function in ISL+ may consume two lists and process them in parallel.

The final observation is that `row-project` checks the membership of a label in `labels` for every single cell. For the cells of the same column in different rows, the result is going to be the same. Hence it also makes sense to hoist this computation out of the function.

This form of hoisting is somewhat more difficult than plain expression hoisting. We basically wish to pre-compute the result of

```
(member? label labels)
```

for all rows and pass the results into the function instead of the list of labels. That is, we replace the list of labels with a list of `Booleans` that indicate whether the cell in the corresponding position is to be kept. Fortunately, computing those `Booleans` is just another application of `keep?` on the schema:

```
(map keep? schema)
```

Instead of keeping some `Specs` from the given `schema` and throwing away the others, this expression just collects the decisions.

```
(define (project db labels)
 (local ((define schema (db-schema db))
 (define content (db-content db))

 ; Spec -> Boolean
 ; does this column belong to the new schema
 (define (keep? c)
 (member? (first c) labels))

 ; Row -> Row
 ; retains those columns whose name is in labels
 (define (row-project row)
 (foldr (lambda (cell m c) (if m (cons cell c) c))
 '()
 row
 mask))
 (define mask (map keep? schema)))
 (make-db (filter keep? schema)
 (map row-project content))))
```

Figure 142: Database projection

Figure 142 shows the final version of `project` and integrates the solutions for the preceding exercises. It also uses `local` to extract and name `schema` and `content`, plus `keep?` for checking whether the label in some `Spec` is worth keeping. The remaining two definitions introduce `mask`, which stands for the list of `Booleans` discussed above, and the revised version of `row-project`. The latter uses `foldr` to process the given `row` and `mask` in parallel.

Compare this revised definition of project with project.v1 in [figure 141](#). The final definition is both simpler and faster than the original version. Systematic design combined with careful revisions pays off; test suites ensure that revisions don't mess up the functionality of the program.

**Exercise 408.** Design the function `select`. It consumes a database, a list of labels, and a predicate on rows. The result is a list of rows that satisfy the given predicate, projected down to the given set of labels.

**Exercise 409.** Design `reorder`. The function consumes a database `db` and list `lol` of [Labels](#). It produces a database like `db` but with its columns reordered according to `lol`. **Hint** Read up on [list-ref](#).

At first assume that `lol` consists exactly of the labels of `db`'s columns. Once you have completed the design, study what has to be changed if `lol` contains fewer labels than there are columns and strings that are not labels of a column in `db`.

**Exercise 410.** Design the function `db-union`, which consumes two databases with the exact same schema and produces a new database with this schema and the joint content of both. The function must eliminate rows with the exact same content.

Assume that the schemas agree on the predicates for each column.

**Exercise 411.** Design `join`, a function that consumes two databases: `db-1` and `db-2`. The schema of `db-2` starts with the exact same [Spec](#) that the schema of `db-1` ends in. The function creates a database from `db-1` by replacing the last cell in each row with the *translation* of the cell in `db-2`.

Here is an example. Take the databases in [figure 137](#). The two satisfy the assumption of these exercises, that is, the last [Spec](#) in the schema of the first is equal to the first [Spec](#) of the second. Hence it is possible to join them:

Name	Age	Description
<a href="#">String</a>	<a href="#">Integer</a>	<a href="#">String</a>
"Alice"	35	"presence"
"Bob"	25	"absence"
"Carol"	30	"presence"
"Dave"	32	"absence"

Its translation maps `#true` to "presence" and `#false` to "absence".

**Hints** (1) In general, the second database may “translate” a cell to a row of values, not just one value. Modify the example by adding additional terms to the row for "presence" and "absence".

(2) It may also “translate” a cell to several rows, in which case the process adds several rows to the new database. Here is a second example, a slightly different pair of databases from those in [figure 137](#):

Name	Age	Present	Present	Description
<a href="#">String</a>	<a href="#">Integer</a>	<a href="#">Boolean</a>	<a href="#">Boolean</a>	<a href="#">String</a>
"Alice"	35	<code>#true</code>	<code>#true</code>	"presence"
"Bob"	25	<code>#false</code>	<code>#true</code>	"here"
"Carol"	30	<code>#true</code>	<code>#false</code>	"absence"

```
"Dave" 32 #false | #false "there"
```

Joining the left database with the one on the right yields a database with eight rows:

Name	Age	Description
String	Integer	String
"Alice"	35	"presence"
"Alice"	35	"here"
"Bob"	25	"absence"
"Bob"	25	"there"
"Carol"	30	"presence"
"Carol"	30	"here"
"Dave"	32	"absence"
"Dave"	32	"there"

(3) Use iterative refinement to solve the problem. For the first iteration, assume that a “translation” finds only one row per cell. For the second one, drop the assumption.

**Note on Assumptions** This exercise and the entire section mostly rely on informally stated assumptions about the given databases. Here, the design of join assumes that “the schema of db-2 starts with the exact same Spec that the schema of db-1 ends in.” In reality, database functions must be checked functions in the spirit of [Input Errors](#). Designing checked-join would be impossible for you, however. A comparison of the last Spec in the schema of db-1 with the first one in db-2 calls for a comparison of functions. For practical solutions, see a text on databases.

---

## 24 Summary

This fourth part of the book is about the design of functions that process data whose description requires many intertwined definitions. These forms of data show up everywhere in the real world, from your computer’s local file system to the world wide web and geometric shapes used in animated movies. After working through this part of the book carefully, you know that the design recipe scales to these forms of data, too:

1. When the description of program data calls for several mutually referential data definitions, the design recipe calls for the simultaneous development of templates, one per data definition. If a data definition A refers to a data definition B, then the template function-for-A refers to function-for-B in the exact same place and manner. Otherwise the design recipes work as before, function for function.
2. When a function has to process two types of complex data, you need to distinguish three cases. First, the function may deal with one of the arguments as if it were atomic. Second, the two arguments are expected to have the exact same structure, and the function traverses them in a completely parallel manner. Third, the function may have to deal with all possible combinations separately. In this case, you make a two-dimensional table that along one dimension enumerates all kinds of data from one data definition and along the other one deals with the second kind of data. Finally you use the table’s cells to formulate conditions and answers for the various cases.

This part of the book deals with functions on two complex arguments. If you ever encounter one of those rare cases where a function receives three complex pieces of data, you know you need (to imagine) a three-dimensional table.

You have now seen all forms of structural data that you are likely to encounter over the course of your career, though the details will differ. If you are ever stuck, remember the design recipe; it will get you started.

## Intermezzo 4: 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,

These chunks are called *bits*, *bytes*, and *words*.

we don't worry about how many digits we process; in principle, we can deal with numbers 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 sections 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, and so on. 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

For pure scientific notation, the base is between 0 and 9; we ignore this constraint.

2. an *exponent*, which is used to determine a 10-based factor.

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 this representation.

We can also use negative exponents, which add 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
; makes 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 "bad values given")])))

; Inex -> Number
; converts 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 143: 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 one of:
; - 1
; - -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 143](#). 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 `Inex` data definition:

```

> (create-inex 120 1 1)
bad values given

```

For other numbers, though, we can find two `Inex` equivalents. One example is [5e-19](#):

```

> (create-inex 50 -1 20)
(make-inex 50 -1 20)
> (create-inex 5 -1 19)
(make-inex 5 -1 19)

```

Use `inex->number` to confirm the equivalence of these two numbers.

With `create-inex` it is also easy to delimit the range of representable numbers, which is actually quite small for many applications:

```
(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)
```

Compare this with 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 in **Inex** arithmetic.

We can also multiply **Inex** numbers.

Recall that

$$\begin{aligned} & (a \cdot 10^n) \cdot (b \cdot 10^m) \\ &= (a \cdot b) \cdot 10^{n+m} \\ &= (a \cdot b) \cdot 10^{(n+m)} \end{aligned}$$

And **inexact** is appropriate.

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 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)

```

**Exercise 412.** 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:

```

(check-expect
 (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 413.** 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 414.** As this section illustrates, gaps in the data representation lead to round-off errors when numbers are mapped to `Inexes`. The problem is, such round-off errors accumulate across computations.

Design `add`, a function that adds up  $n$  copies of `#i1/185`. For your examples, use `0` and `1`; 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?

While scientific notation expands the range of numbers we can represent with fixed-size chunks of data, it is still finite. 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 [Inex](#) arithmetic produces numbers that are too large to be represented, we speak of (arithmetic) *overflow*.

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 415.** 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 an underflow, however, means dropping the entire mantissa, meaning the result is not within a predictable percentage range of the true result.

**Exercise 416.** ISL+ uses `#i10.0` 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 415.

---

## \*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 with digits from the 2-based number system.

Inexact real representations come in several flavors:  
`float`, `double`, `extflonum`, and so on.

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 a 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, `sqrt` returns an inexact number for both the exact and inexact representation of `2`. In contrast, `sqrt` 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, you may wonder 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.

For an accessible introduction—using Racket—read [Practically Accurate Floating-Point Math](#), an article on error analysis by Neil Toronto and Jay McCarthy. It is also fun to watch [Debugging Floating-Point Math in Racket](#), Neil Toronto's RacketCon 2011 lecture, available on YouTube.

As a first taste, the following exercises illustrate how bad things can get. Work through them to never lose sight of the problems of inexact numbers.

**Exercise 417.** Evaluate `(expt 1.001 1e-12)` in Racket and in ISL+. Explain what you see.

**Exercise 418.** 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)`? And how about `(my-expt exac 30)`? Which answer is more useful?

```
(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))
```

Figure 144: A Janus-faced series of inexact numbers

**Exercise 419.** 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, we run into problems when adding numbers that vary by more than a factor of  $10^{16}$ :

$$1.0 \cdot 10^{16} + 1 = 1.000000000000001 \cdot 10^{16},$$

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 list of numbers in [figure 144](#) and determine the values of these 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?

Generic advice on inexact calculations tells programmers to start additions with the smallest numbers. While adding a big number to two small numbers might yield the big one, adding small numbers first creates a large one, which 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

```

This trick may **not** work; see the JANUS interactions above.

In a language such as ISL+, you can convert the numbers to exact rationals, use exact arithmetic on those, and convert the result back:

```
(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 420.** JANUS is just a fixed list, but take a look at this function:

```

(define (oscillate n)
 (local ((define (0 i)
 (cond
 [(> i n) '()]
 [else
 (cons (expt #i-0.99 i) (0 (+ i 1))))])
 (0 1)))

```

Applying `oscillate` to a natural number  $n$  produces the first  $n$  elements of a mathematical series. It is best understood as a graph, like the one in [figure 145](#). Run `(oscillate 15)` in DrRacket and inspect the result.

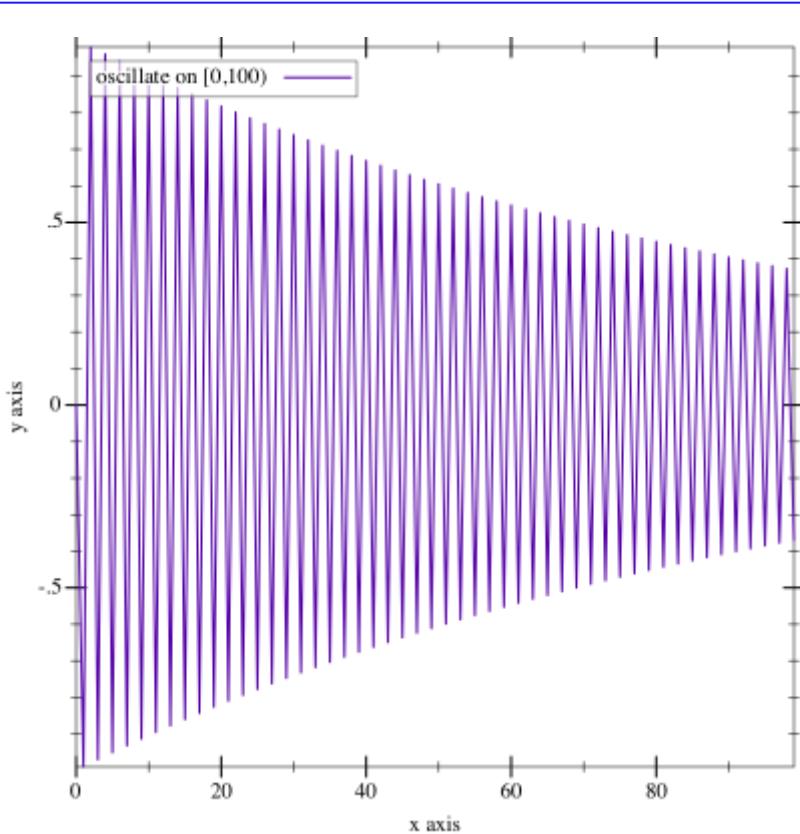


Figure 145: The graph of oscillate

Summing its results 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
```

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 world of financial statements, 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 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 Generative Recursion

If you follow the design recipe of the first four parts, either you 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

Some functions merely compose such functions; we group those with the “structural” group.

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 world, some problems cannot be solved with a structural approach to design.

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 rearrange 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 reused 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 commonsense knowledge about breaking up sequences

In Greek, it's “eureka!”

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.

---

## 25 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.

---

## 25.1 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 definitions 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 `1String`s. The function’s task is to bundle up chunks 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 chunks 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 `1String`s:

```
(list "a" "b" "c" "d" "e" "f" "g" "h")
```

If we tell `bundle` to bundle this list into pairs—that is, `n` is 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 test:

```
(check-expect (bundle (explode "abcdefg") 3)
 (list "abc" "def" "g")))
```

Note the use of `explode`; it makes the test readable.

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 must 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 compound, s considered atomic
; (Processing Two Lists Simultaneously: Case 1)
(define (bundle s n)
 (cond
 [(zero? n) (...)]
 [else (... s ... n ... (bundle s (sub1 n))))])

; [List-of 1String] as compound, n atomic
; (Processing Two Lists Simultaneously: Case 1)
(define (bundle s n)
 (cond
 [(empty? s) (...)]
 [else (... s ... n ... (bundle (rest s) n))]))

; [List-of 1String] and N are on equal footing
; (Processing Two Lists Simultaneously: Case 2)
(define (bundle s n)
 (cond
 [(and (empty? s) (zero? n)) (...)]
 [else (... s ... n ... (bundle (rest s) (sub1 n))))]))

; consider all possibilities
; (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 (... (bundle s (sub1 n)) ...
 ... (bundle (rest s) n) ...)]))
```

Figure 146: Useless templates for breaking up strings into chunks

The template step reveals that a structural approach cannot work. Figure 146 shows four possible templates. Since both arguments to `bundle` are complex, the first two consider one of the arguments atomic. That clearly cannot be the case because the function has to take apart

each argument. The third template is based on the assumption that the two arguments are processed in lockstep, 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 processed independently, meaning there are four possibilities to proceed at each stage. This final design decouples the arguments too much because the list and the counting number must be processed together. In short, we must admit that the structural templates appear to be useless for this design problem.

```

; [List-of 1String] N -> [List-of String]
; bundles chunks of s into strings of length n
; idea take n items and drop n at a time
(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]
; keeps the first n items from 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]
; removes 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 147: Generative recursion

Figure 147 shows a complete definition for `bundle`. The definition uses the `drop` and `take` functions requested in [exercise 395](#); 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 1Strings` 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`; and
4. finally, `cons` combines the string from 2 with the list of strings from 3 to create the result for the complete list.

List item 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 421.** Is `(bundle '("a" "b" "c") 0)` a proper use of the `bundle` function? What does it produce? Why?

**Exercise 422.** 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 423.** Define `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 421](#).

**Note** The `partition` function is somewhat closer to what a cooperative DrRacket environment would need than `bundle`.

---

## 25.2 Recursion that Ignores Structure

Recall that the `sort>` function from [Design by Composition](#) consumes a list of numbers and rearranges it 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 reprocesses the result of the natural recursions.

Hoare's quick-sort algorithm goes about sorting lists in a radically different manner and has become the classic example of generative recursion. The underlying generative step uses the time-honored strategy of divide-and-conquer. That is, it divides the nontrivial 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
- and another one with all those items that are strictly larger.

Then the two smaller lists are sorted via the quick-sort algorithm. Once the two lists are sorted, the results are composed with the first item placed in the middle. Owing to its special

role, the first item on the list is called the *pivot item*.

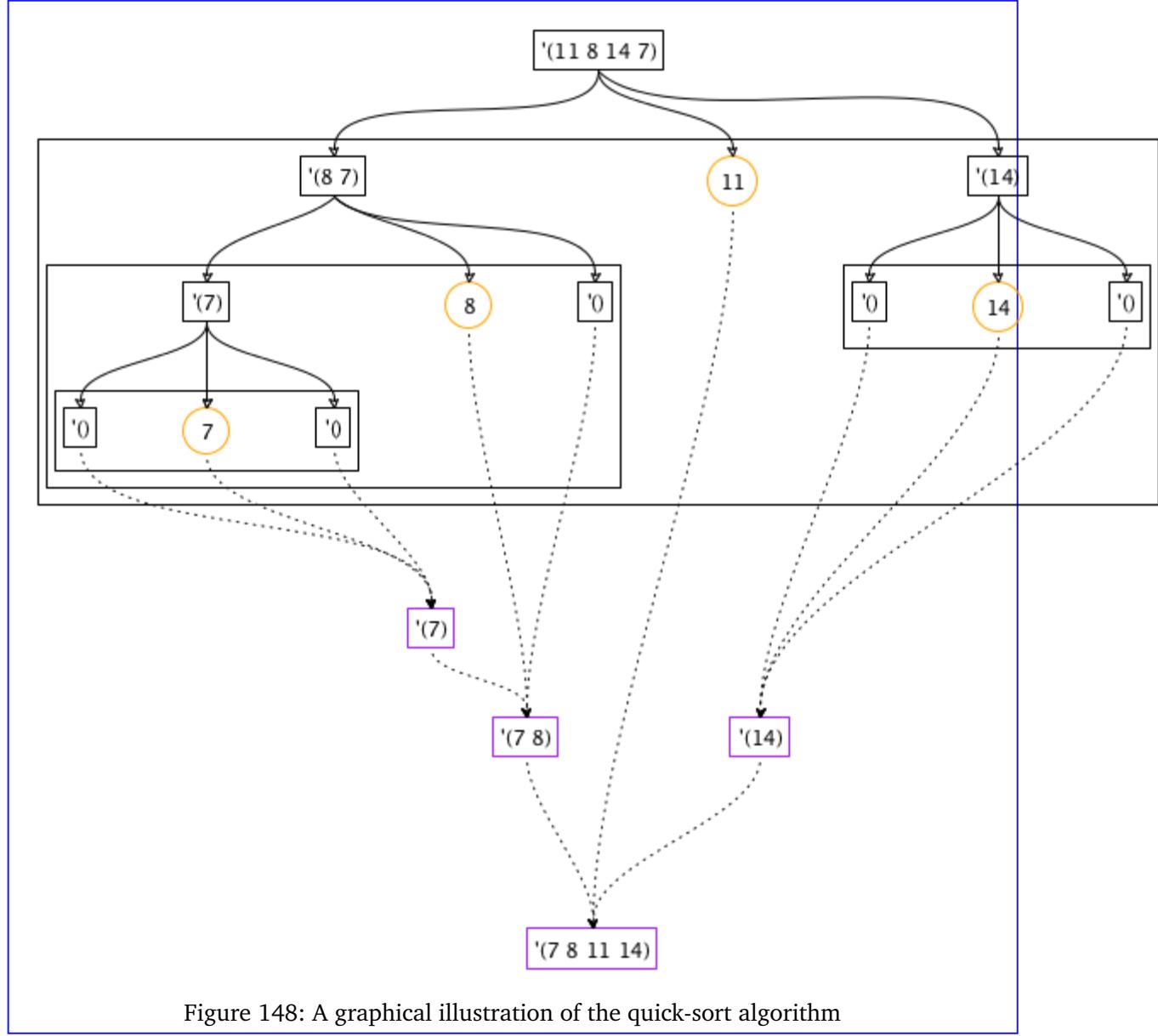


Figure 148: A graphical illustration of the quick-sort algorithm

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 148 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 424.** Draw a quick-sort diagram like the one in [figure 148](#) 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]
; produces a sorted version of alon
(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: `smallers` and `largers`. 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 149](#).

```
; [List-of Number] -> [List-of Number]
; produces a sorted version of alon
; assume the numbers are all distinct
(define (quick-sort< alon)
 (cond
 [(empty? alon) '()]
 [else (local ((define pivot (first alon)))
 (append (quick-sort< (smallers alon pivot))
 (list pivot)
 (quick-sort< (largers alon pivot))))]))

; [List-of Number] Number -> [List-of Number]
(define (largers alon n)
 (cond
 [(empty? alon) '()]
 [else (if (> (first alon) n)
 (cons (first alon) (largers (rest alon) n))
 (largers (rest alon) n))]))
```

```

[else (if (> (first alon) n)
 (cons (first alon) (largers (rest alon) n))
 (largers (rest alon) n)))))

; [List-of Number] Number -> [List-of Number]
(define (smallers alon n)
 (cond
 [(empty? alon) '()]
 [else (if (< (first alon) n)
 (cons (first alon) (smallers (rest alon) n))
 (smallers (rest alon) n))))])

```

Figure 149: 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< (smallers alon pivot))`, which sorts the list of items smaller than the pivot; and
2. `(quick-sort< (largers 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< (smallers alon pivot))
 (list (first alon))
 (quick-sort< (largers alon pivot)))

```

Figure 149 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 148](#).

Both [figure 148](#) 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 425.** Articulate purpose statements for `smallers` and `largers` in [figure 149](#).

**Exercise 426.** Complete the hand-evaluation from above. A close inspection of the evaluation suggests an additional trivial case for `quick-sort<`. Every time `quick-sort<` consumes a list of one item, it returns it as is. After all, the sorted version of a list of one item is the list itself.

Modify `quick-sort<` to take advantage of this observation. Evaluate the example again. How many steps does the revised algorithm save?

**Exercise 427.** 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<` (an appropriately adapted variant of `sort>` from [Auxiliary Functions that Recur](#)) if the length of the input is below some threshold.

**Exercise 428.** 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 429.** Use `filter` to define `smallers` and `largers`.

**Exercise 430.** 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 the pivot 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.

---

## 26 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 shows how to adapt the process dimension of the design recipe to generative recursion. The second section homes 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 contrast structural and generative recursion.

---

### 26.1 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 148](#) 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 empty list or a one-item list are trivially solvable. A list with many items is a nontrivial problem. For these nontrivial 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.

For this part of the book, “trivial” is a technical term.

Based on this 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 150](#) spells out how some simple numeric examples work out for the nontrivial cases. From these examples it is straightforward to guess that the answer to the fourth question is to append the sorted list of smaller numbers, the pivot number, and the sorted list of larger numbers, 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.

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 150: The table-based guessing approach for combining solutions

**Exercise 431.** Answer the four key questions for the `bundle` problem and the first three questions for the `quick-sort<` problem. How many instances of `generate-problem` are needed?

**Exercise 432.** Exercise 219 introduces the function `food-create`, which consumes a `Posn` and produces a randomly chosen `Posn` that is guaranteed to be distinct from the given one. First reformulate the two functions as a single definition, using `local`; then justify the design of `food-create`.

## 26.2 Termination

Generative recursion adds an entirely new aspect to computations: non-termination. A function such as `bundle` may never produce a value or signal an error for certain inputs.

Exercise 421 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 how evaluating `(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 generative recursion, an algorithm may generate new problems without any

limitations. If the design recipe required a guarantee that the new problems 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 mostly intact and supplement them with a seventh step: the *termination argument*. [Figure 151](#) presents the first part of the design recipe for generative recursion, and [figure 152](#) the second one. They show the design recipe in the conventional tabular form. The unmodified steps come with a dash 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 in [figure 152](#) is completely new.

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 theorem for such arguments. The second kind 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 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.

<b>steps</b>	<b>outcome</b>	<b>activity</b>
problem analysis	data representation and definition	—
header	a purpose statement concerning the “how” of the function	supplement the explanation of <b>what</b> the function computes with a one-liner on <b>how</b> it computes the result
examples	examples and tests	work through the “how” with several examples
template	fixed template	—

Figure 151: Designing algorithms (part 1)

<b>steps</b>	<b>outcome</b>	<b>activity</b>
definition	full-fledged function definition	formulate conditions for trivially solvable problems; formulate answers for these trivial cases; determine how to generate new problems for nontrivial 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	investigate whether the problem data for each recursive data is smaller than the given data; find examples that cause the function to loop

call or (2)  
examples of  
exceptions to  
termination

---

Figure 152: Designing algorithms (part 2)

Let's illustrate the two kinds of termination arguments with 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. For `quick-sort<`, the key observation is that each recursive use of `quick-sort<` receives a list that is shorter than `alon`:

```
; [List-of Number] -> [List-of Number]
; creates a sorted variant of alon
; termination both recursive calls to quick-sort<
; receive list that miss the pivot item
(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 433.** 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 434.** Consider the following definition of `smallers`, one of the two “problem generators” for `quick-sort<`:

```
; [List-of Number] Number -> [List-of Number]
(define (smallers l n)
 (cond
 [(empty? l) '()]
 [else (if (<= (first l) n)
 (cons (first l) (smallers (rest l) n))
 (smallers (rest l) n))]))
```

What can go wrong when this version is used with the `quick-sort<` definition from [Recursion that Ignores Structure?](#)

**Exercise 435.** When you worked on [exercise 430](#) or [exercise 428](#), you may have produced looping solutions. Similarly, [exercise 434](#) actually reveals how brittle the termination argument is for `quick-sort<`. In all cases, the argument relies on the idea that `smallers` and `largers` produce lists that are maximally as long as the given list, and on 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 receive lists that are shorter than the given one.

**Exercise 436.** Formulate a termination argument for `food-create` from exercise 432.

## 26.3 Structural versus Generative Recursion

The template for algorithms is so general that it includes structurally recursive functions. Consider the left side of figure 153. 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 153.

```
(define (general P) (define (special P)
 (cond (cond
 [(trivial? P) (solve P)] [(empty? P) (solve P)]
 [else [else
 (combine-solutions (combine-solutions
 P P
 (general (special (rest P))))))
 (generate P))))]))
```

Figure 153: From generative to structural recursion

**Exercise 437.** 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?

Now you may wonder whether there is 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 unhelpful.

## 26.4 Making Choices

When you interact with a function `f` that sorts lists of numbers, it is impossible for you to know whether `f` is `sort<` 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. More generally, when we can design a function using structural recursion and generative recursion, we must figure out which one to pick.

*Observable equivalence* is a central concept from the study of programming languages.

To illustrate the consequences of this choice, we discuss a classical example from mathematics: the problem of finding the greatest common divisor (*gcd*) of two positive natural numbers. All such numbers have 1 as divisor in common. On occasion—say, 2 and 3—this is also the only common divisor. Both 6 and 25 are numbers with several divisors:

John Stone suggested the greatest common divisor as a suitable example.

- 6 is evenly divisible by 1, 2, 3, and 6;
- 25 is evenly divisible by 1, 5, and 25.

And yet, 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
(define (gcd n m)
 (check-expect (gcd 6 25) 1)
 (check-expect (gcd 18 24) 6)
 (define (gcd n m) 42))
```

The signature specifies the inputs as natural numbers greater than 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 154](#) 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 gcd 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 154: Finding the greatest common divisor via structural recursion

**Exercise 438.** In your words: how does `greatest-divisor-<=` work? Use the design recipe to find the right words. Why does the locally defined `greatest-divisor-<=` recur on `(min n m)`?

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. To get there, gcd-structural checks the “evenly divisible” condition for 45014640, that is, it checks  $45014640 - 177$  remainders. Checking that many `remainders`—twice!—is a large effort, and even reasonably fast computers need time to complete this task.

**Exercise 439.** Copy gcd-structural into 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,  $L$  for **large** and  $S$  for **small**, the greatest common divisor is equal to the greatest common divisor of  $S$  and the remainder of  $L$  divided by  $S$ .

Here is how we can articulate this insight as an equation:

```
(gcd L S) == (gcd S (remainder L S))
```

Since `(remainder L S)` is smaller than both  $L$  and  $S$ , the right-hand side use of `gcd` consumes  $S$  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 gcd as 18 and 6.
- And these two have the same greatest common divisor as 6 and 0.

Now 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 numbers is 0, we face a trivial case;
- the larger of the two numbers is the solution in the trivial case;
- generating a new problem requires one `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 short, the answers for the four design-recipe questions fall out.

```
(define (gcd-generative n m)
 (local (; N[>= 1] N[>=1] -> N
```

```

; generative recursion
; (gcd L S) == (gcd S (remainder L S))
(define (clever-gcd L S)
 (cond
 [(= S 0) L]
 [else (clever-gcd S (remainder L S))]))
(clever-gcd (max m n) (min m n)))

```

Figure 155: Finding the greatest common divisor via generative recursion

Figure 155 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 above example,

```
(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 440.** 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, programmers who read functions can easily understand structurally recursive functions, even without much documentation. The generative step of an algorithm,

though, is based on a “eureka!” and, without a good explanation, is difficult to understand for future readers—and that includes older versions of yourself.

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 the result turns out to be too slow for the task at hand—and only then—is it time to explore the use of generative recursion.

**Exercise 441.** 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 442.** 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 often wins over `quick-sort<` for short lists and vice versa?

Determine the cross-over point. Use it to build a `clever-sort` function that behaves like `quick-sort<` for large lists and like `sort<` for lists below this cross-over point. Compare with [exercise 427](#).

**Exercise 443.** 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 444.** [Exercise 443](#) 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 S L)
 (largest-common (divisors S S) (divisors S L)))

; N[>= 1] N[>= 1] -> [List-of N]
; computes the 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)
```

Why do you think `divisors` consumes two numbers? Why does it consume `S` as the first argument in both uses?

---

## 27 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, immersion in an application domain, 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 `1String`s, also a common problem in real-world programming.

---

### 27.1 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.”

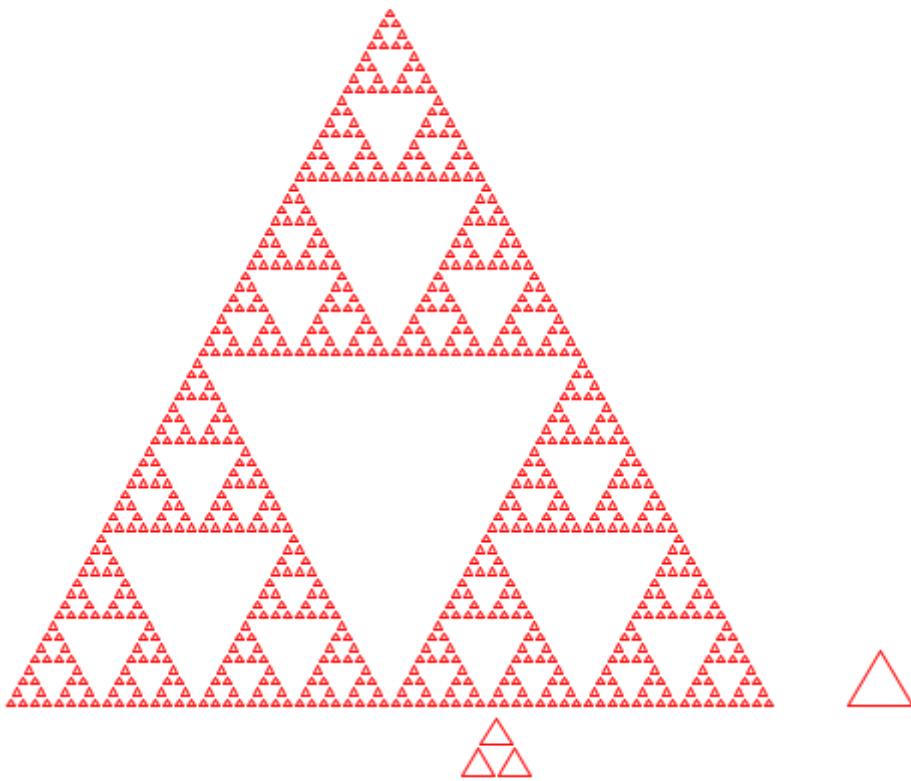


Figure 156: The Sierpinski triangle

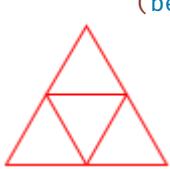
Figure 156 displays an example of a fractal shape, known as the Sierpinski triangle. The basic shape is an (equilateral) triangle, like the one in the center. When this triangle is composed sufficiently many times in a triangular fashion, we get the left-most shape.

The right-most image in figure 156 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:

```
> (s-triangle 3)

> (beside (s-triangle 3) (s-triangle 3))

> (above (s-triangle 3)
 (beside (s-triangle 3) (s-triangle 3)))

```

We owe this solution to Marc Smith.

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, a purpose statement, and a 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  $side$ .

```
(define SMALL 4) ; a size measure in terms of pixels

(define small-triangle (triangle SMALL 'outline 'red))

; Number -> Image
; generative creates Sierpinski Δ of size side by generating
; one for (/ side 2) and placing one copy above two 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 157: The Sierpinski algorithm

With these answers, it is straightforward to define the function. [Figure 157](#) spells out the details. The “triviality condition” translates to `(<= side SMALL)` for some constant `SMALL`. For the trivial answer, the function returns a 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, it composes this image via `above` and `beside`.

The figure highlights two other points. First, the purpose statement is articulated as an explanation of **what** the function accomplishes

```
; creates 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.

---

## 27.2 Binary Search

Applied mathematicians model the real world with nonlinear 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  $f$ .

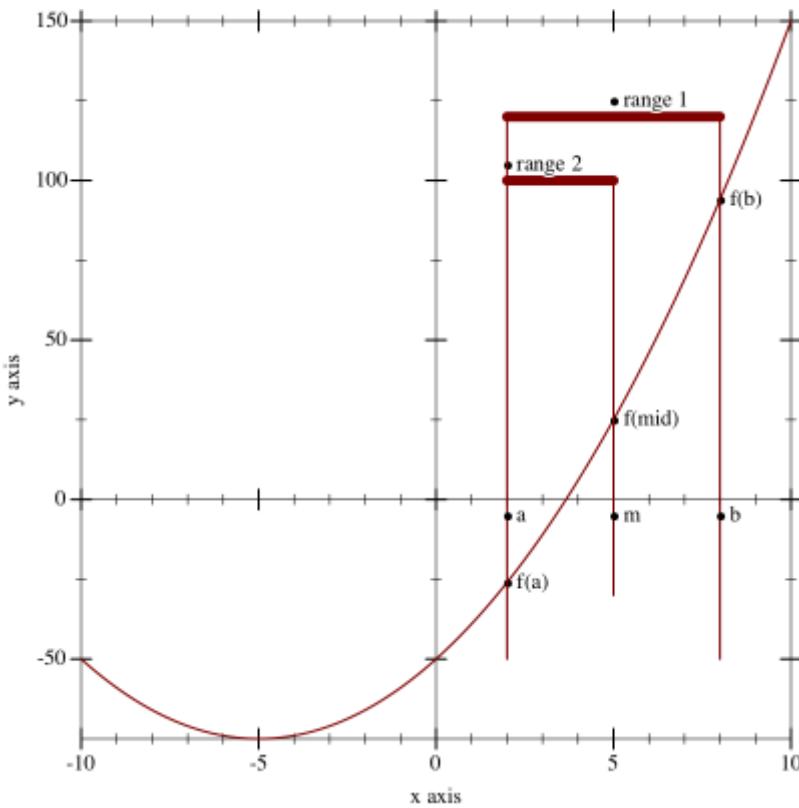


Figure 158: A numeric function  $f$  with root in interval  $[a,b]$  (stage 1)

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 toward 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 if  $d_0$ ,  $a$ , and  $v$  satisfy certain conditions.

Generally such problems call for more complexity than quadratic equations. 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* (IVT), an early result of analysis. The resulting algorithm is a primary example of generative recursion based on a mathematical theorem. Computer scientists have generalized it to the *binary search* algorithm.

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 opposite sides 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.

[Figure 158](#) illustrates the Intermediate Value Theorem. The function  $f$  is a continuous function, as suggested by the uninterrupted, smooth graph. It is below the x-axis at  $a$  and above at  $b$ , and indeed, it intersects the x-axis somewhere in this interval, 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 sized intervals. We can now compute the value of  $f$  at  $m$  and see whether it is below 0 or above. 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 158](#).

We now have a description of the key step in the root-finding process. Next, we translate this description into an ISL+ algorithm. Our first task is to state its purpose. 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 the following holds:

```
(or (<= (f left) 0 (f right))
 (<= (f right) 0 (f left)))
```

that is, `(f left)` and `(f right)` must be on opposite sides 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  $\varepsilon$ . 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:

DrRacket allows the use of Greek symbols such as  $\varepsilon$ .  
But you can also write EPSILON instead.

```
; [Number -> Number] Number Number -> Number
; determines R such that f has a root in [R, (+ R ε)]
; assume f is continuous
; (2) (or (<= (f left) 0 (f right)) (<= (f right) 0 (f left)))
; generative divides interval in half, the root is in
; one of the two halves, picks according to (2)
(define (find-root f left right)
 0)
```

**Exercise 445.** 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 for  $\epsilon = 0$ :

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	?	?

Our next task is to address the four 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) ε)
```

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 requires determining the midpoint and its function value:

```
(local ((define mid (/ (+ left right) 2))
 (define f@m (f mid)))
 ...)
```

The midpoint is then used to pick the next interval. Following IVT, the interval  $[left, mid]$  is the next candidate if

```
(or (<= (f left) 0 f@m) (<= f@m 0 (f left)))
```

while  $[mid, right]$  is used for the recursive call if

```
(or (<= f@m 0 (f right)) (<= (f right) 0 f@m))
```

Translated into code, the body of `local` must be a conditional:

```
(cond
 [(or (<= (f left) 0 f@m) (<= f@m 0 (f left)))
 (... (find-root f left mid) ...)]
 [(or (<= f@m 0 (f right)) (<= (f right) 0 f@m))
 (... (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 nothing else to do.

The completed function is displayed in figure 159; the following exercises elaborate on its design.

```
; [Number -> Number] Number Number -> Number
; determines R such that f has a root in [R,(+ R ε)]
; assume f is continuous
; assume (or (<= (f left) 0 (f right)) (<= (f right) 0 (f left)))
; generative divides interval in half, the root is in one of the two
; halves, picks according to assumption
(define (find-root f left right)
 (cond
 [(<= (- right left) ε) 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 159: The *find-root* algorithm

**Exercise 446.** Add the test from [exercise 445](#) to the program in [figure 159](#). Experiment with different values for  $\epsilon$ .

**Exercise 447.** The `poly` function has two roots. Use `find-root` with `poly` and an interval that contains both roots.

**Exercise 448.** 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  $S_1$ . How large is the distance between  $left$  and  $right$  for the first and second recursive call to `find-root`? After how many steps is  $(- right left)$  smaller than or equal to  $\epsilon$ ?

**Exercise 449.** As presented in [figure 159](#), `find-root` computes the value of  $f$  for each boundary value twice to generate the next interval. Use `local` to avoid this recomputation.

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 recomputations 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 450.** A function  $f$  is *monotonically increasing* if  $(\leq (f a) (f b))$  holds whenever  $(\leq a b)$  holds. Simplify `find-root` assuming the given function is not only continuous but also monotonically increasing.

**Exercise 451.** 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 structure:
; (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 a number in `(table-array t)` that is close 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 the 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 as to which interval to generate next. To make this concrete, imagine a table with 1024 slots and the root at 1023. How many calls to `find` are needed in `find-linear` and `find-binary`, respectively?

## 27.3 A Glimpse at Parsing

As mentioned in [Iterative Refinement](#), computers come with files, which provide a form of permanent memory. From our perspective a *file* is just a list of [1Strings](#), 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 represents the content of a file
; "\n" is 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 a [List-of 1String].
```

Many functions need to process files as list of lines. The [read-lines](#) from the [2htdp/batch-io](#) library is one of them. Concretely, the function turns 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" "?")
```

into 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 representations 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 one `"\n"` or some other [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
(define (file->list-of-lines afile)
 (cond
 [(empty? afile) '()]
 [else
 (cons (first-line afile)
 (file->list-of-lines (remove-first-line afile)))]))

; File -> Line
```

```

(define (first-line afile)
 (cond
 [(empty? afile) '()]
 [(string=? (first afile) NEWLINE) '()]
 [else (cons (first afile) (first-line (rest afile)))]))

; File -> Line

(define (remove-first-line afile)
 (cond
 [(empty? afile) '()]
 [(string=? (first afile) NEWLINE) (rest afile)]
 [else (remove-first-line (rest afile))]))

(define NEWLINE "\n") ; the 1String

```

Figure 160: 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 traverse their inputs in a structurally recursive manner; their development is a straightforward exercise. [Figure 160](#) presents the complete program code.

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")
 '())))

```

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 452.** Both `first-line` and `remove-first-line` are missing purpose statements. Articulate proper statements.

**Exercise 453.** 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 454.** Design `create-matrix`. The function consumes a number  $n$  and a list of  $n^2$  numbers. It produces an  $n \times n$  matrix, for example:

```
(check-expect
 (create-matrix 2 (list 1 2 3 4))
 (list (list 1 2)
 (list 3 4)))
```

Make up a second example.

---

## 28 Mathematical Examples

Many solutions to mathematical problems employ generative recursion. A future programmer must get to know such solutions for two reasons. On the 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.

---

### 28.1 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 about tangents: their slopes and 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  $\epsilon$  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 determine a line and a slope:

$$\text{slope}(f, r_1) = \frac{f(r_1 + \epsilon) - f(r_1 - \epsilon)}{(r_1 + \epsilon) - (r_1 - \epsilon)} = \frac{1}{2\epsilon} \cdot (f(r_1 + \epsilon) - f(r_1 - \epsilon))$$

**Exercise 455.** Translate this mathematical formula into the ISL+ function `slope`, which maps function `f` and a number `r1` to the slope of `f` at `r1`. 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

$$\text{tangent}(x) = \text{slope}(f, r_1) \cdot (x - r_1) + f(r_1)$$

Finding the root of `tangent` means finding a value `root-of-tangent` so that `tangent(root-of-tangent)` equals 0:

$$0 = \text{slope}(f, r_1) \cdot (\text{root-of-tangent} - r_1) + f(r_1).$$

We can solve this equation in a straightforward manner:

$$\text{root-of-tangent} = r_1 - \frac{f(r_1)}{\text{slope}(f, r_1)}.$$

**Exercise 456.** Design `root-of-tangent`, a function that maps `f` and `r1` to the root of the tangent through  $(r_1, f(r_1))$ .

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`:

```
; [Number -> Number] Number -> Number
; finds a number r such that (f r) is small
; generative repeatedly generates improved guesses
(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  $(f r_1)$  is close enough to 0, the problem is solved. Close to 0 could mean  $(f r_1)$  is a small positive number or a small negative number. Hence we check its absolute value:

```
| (=< (abs (f r1)) ε)
```

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.

```

; [Number -> Number] Number -> Number
; finds a number r such that (\leq (abs (f r)) ϵ)

(check-within (newton poly 1) 2 ϵ)
(check-within (newton poly 3.5) 4 ϵ)

(define (newton f r1)
 (cond
 [(\leq (abs (f r1)) ϵ) r1]
 [else (newton f (root-of-tangent f r1))]))

; see exercise 455
(define (slope f r) ...)

; see exercise 456
(define (root-of-tangent f r) ...)

```

Figure 161: The Newton process

Figure 161 displays newton. It includes two tests that are derived from the tests in [Binary Search](#) for find-root. After all, both functions search for the root of a function, and poly has two known roots.

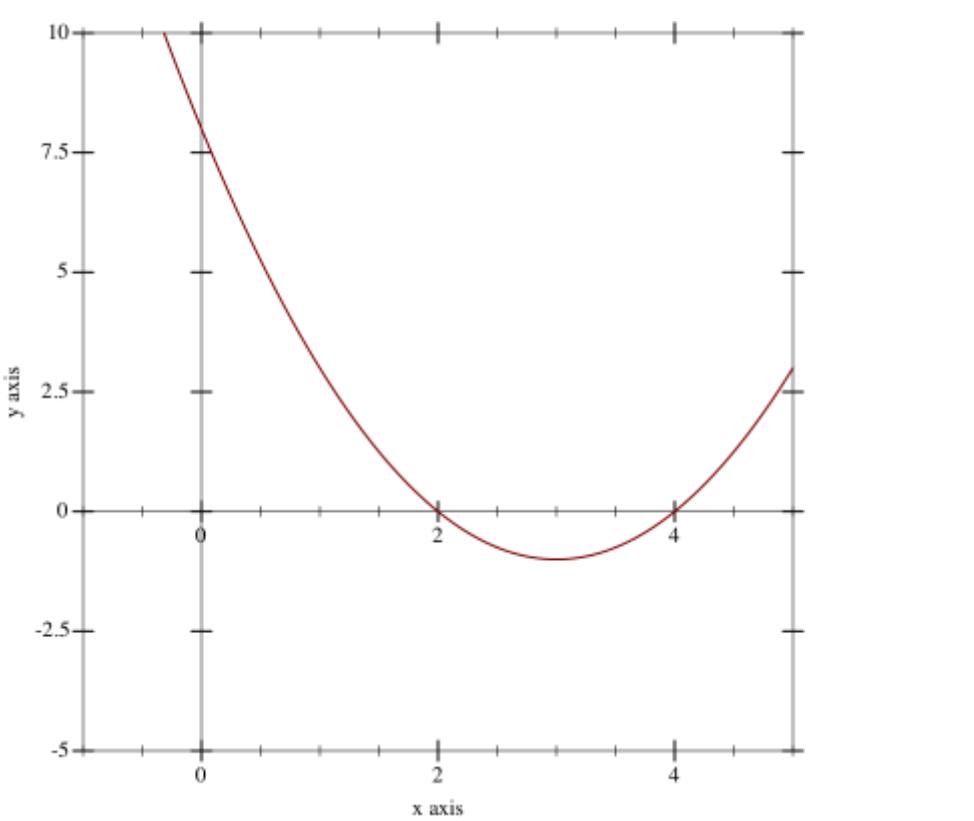


Figure 162: The graph of poly on the interval [-1,5]

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:

```
; Number -> Number
```

```
| (define (poly x) (* (- x 2) (- x 4)))
```

As mentioned, its roots are 2 and 4. The graph of poly in figure 162 confirms these roots and also shows that between the two roots the function flattens out. For a mathematically inclined person, this shape raises the question of 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 the root-of-tangent function turns it into an error:

```
| > (slope poly 3)
| 0
| > (root-of-tangent poly 3)
|/:division by zero
```

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 seconds 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 problems when it comes to complex termination behavior. For some inputs,

The calculation in newton turns #i+inf.0 into +nan.0, a piece of data that says “not a number.”

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.

Most arithmetic operations propagate this value, which explains the behavior of `newton`.

**Exercise 457.** 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.

This exercise was suggested by 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  $72/r$  month as long as the interest rate  $r$  is “small.”

## 28.2 Numeric Integration

Many physics problems boil down to determining the area under a curve:

**Sample Problem** A car drives at a 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  $12 \text{ m/s}^2$ . What height does it reach after [5](#), [10](#), [15](#) seconds?

Physics tells us that a vehicle travels  $d_{con}(t) = v \cdot t$  meters if it moves at a constant speed  $v$  for  $t$  seconds. For vehicles that accelerate, the distance traveled depends on the square of the time  $t$  passed:

$$d_{acc}(t) = \frac{1}{2} \cdot a \cdot t^2$$

In general, the law tells us that the distance corresponds to the area under the graph of speed  $v(t)$  over time  $t$ .

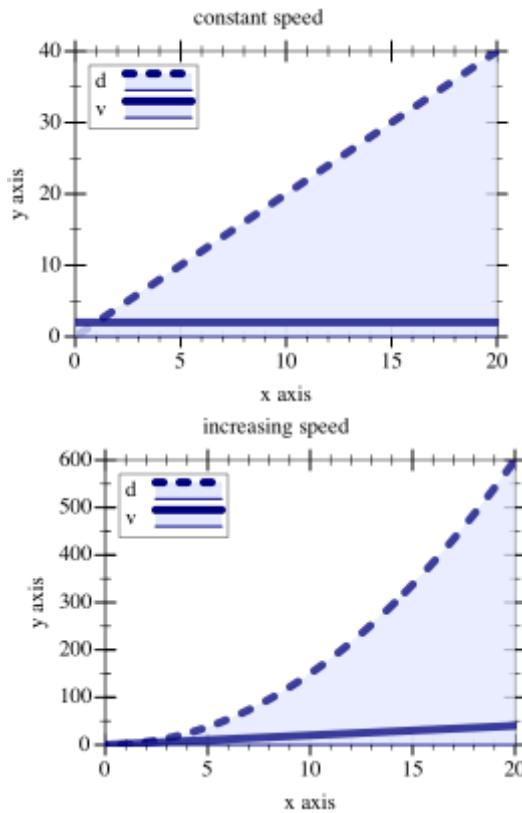


Figure 163: Distance traveled with constant vs accelerating speed

[Figure 163](#) 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 164](#), 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.

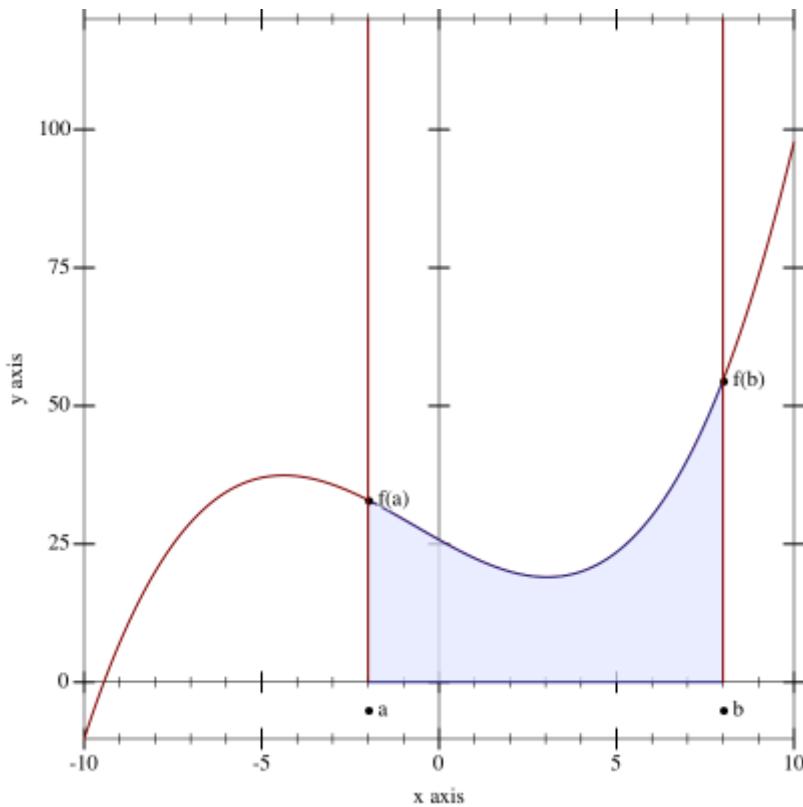


Figure 164: Integrating a function  $f$  between  $a$  and  $b$

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 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 ε 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 ε)
(check-within (integrate (lambda (x) (* 2 x)) 0 10) 100 ε)
(check-within (integrate (lambda (x) (* 3 (sqr x))) 0 10)
 1000
 ε)

(define (integrate f a b) #i0.0)
```

Figure 165: A generic integration function

Figure 165 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 come 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.

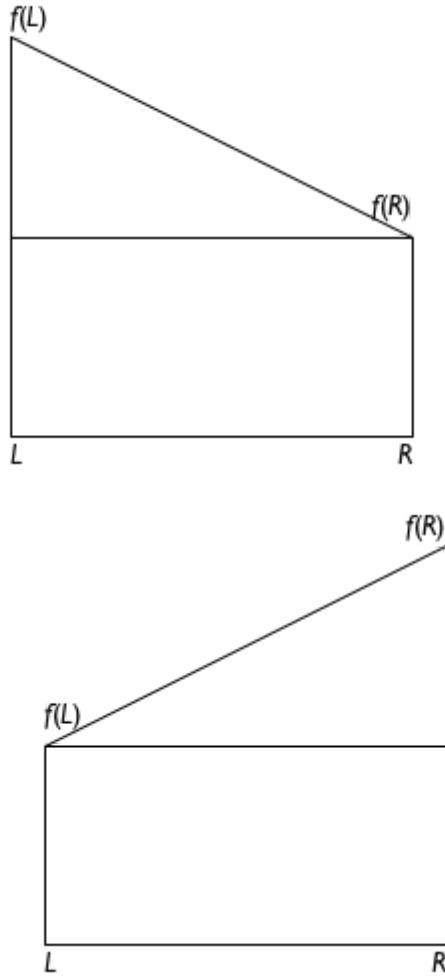
The following two exercises show how to turn domain knowledge into integration functions. Both functions compute rather crude approximations. While the design of the first uses only mathematical formulas, the second also exploits a bit of structural design ideas. Solving these exercises creates the necessary appreciation for the core of this section, which presents a generative-recursive integration algorithm.

**Exercise 458.** Kepler suggested a simple integration method. To compute an estimate of the area under  $f$  between  $a$  and  $b$ , proceed as follows:

The method is known as *Kepler's rule*.

1. divide the interval into half at  $mid = (a + b) / 2$ ;
2. compute the areas of these two trapezoids:
  - 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.

**Domain Knowledge** Let's take a look at these trapezoids. Here are the two possible shapes, with minimal annotations to reduce clutter:



The left shape assumes  $f(L) > f(R)$  while the right one shows the case where  $f(L) < f(R)$ . Despite the asymmetry, it is still 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, while it **subtracts** the triangle from the area of the large rectangle for the right one.

Also show that the above formula is equal to

$$\frac{1}{2} \cdot (R - L) \cdot (f(L) + f(R))$$

This is a mathematical validation of the asymmetry of the formula.

Design the function `integrate-kepler`. That is, turn the mathematical knowledge into an ISL+ function. Adapt the test cases from [figure 165](#) to this use. Which of the three tests fails and by how much?

**Exercise 459.** 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

$$W = (b - a)/R .$$

The height of one of these rectangles is the value of  $f$  at its midpoint. The first midpoint is clearly at  $a$  plus half of the width of the rectangle,

$$S = \text{width}/2 ,$$

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 these rectangles, we can now determine the area under the graph:

$$\begin{aligned} \sum_{i=0}^{i=R-1} W \cdot f(a + i \cdot W + S) &= W \cdot f(a + 0 \cdot W + S) \\ &\quad + \\ &\quad \dots \\ &\quad + \\ &\quad \dots \\ &= W \cdot f(a + (R - 1) \cdot W + S) . \end{aligned}$$

Turn the description of the process into an ISL+ function. Adapt the test cases from [figure 165](#) to this case.

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 an  $\epsilon$  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 458](#).

The Kepler method of [exercise 458](#) 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 460.** 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 divide-and-conquer approach of [exercise 460](#) is wasteful. Consider a function whose graph is level in one part and rapidly changes in another; see [figure 166](#) 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.

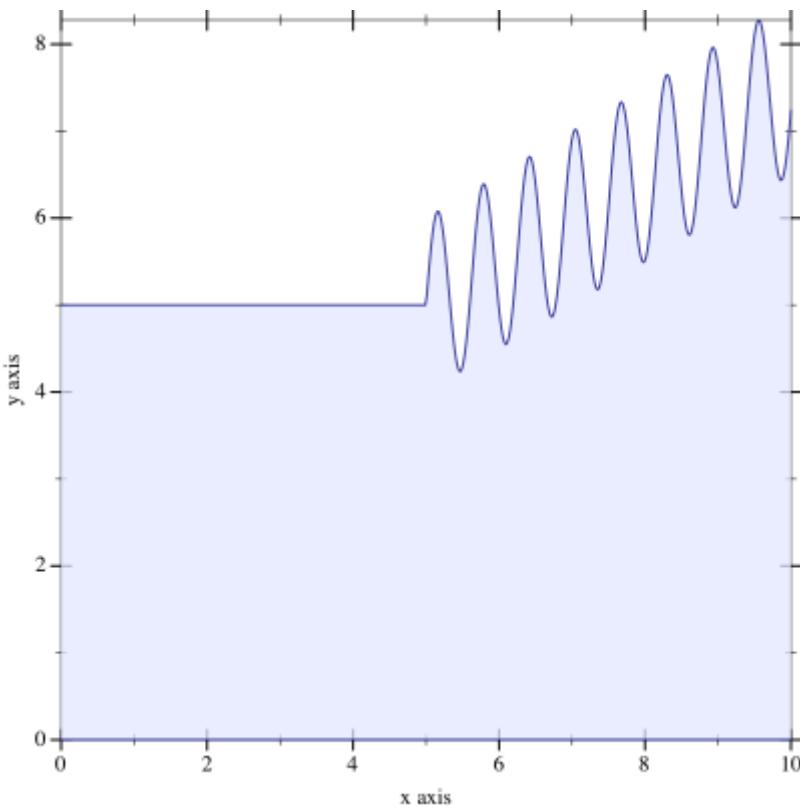


Figure 166: A candidate for adaptive integration

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  $\varepsilon$  and width  $b - a$ ,

$$\varepsilon \cdot (b - a)$$

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 461.** Design `integrate-adaptive`. That is, turn the recursive process description into an ISL+ algorithm. Make sure to adapt the test cases from [figure 165](#) to this use.

Do not discuss the termination of `integrate-adaptive`.

Does `integrate-adaptive` always 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 need 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.

## 28.3 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 gas ( $z$ ) are determined by these exchange equations:

$$\begin{array}{rcl} 2 \cdot x + 2 \cdot y + 3 \cdot z & = & 10 \\ 2 \cdot x + 5 \cdot y + 12 \cdot z & = & 31 \\ 4 \cdot x + 1 \cdot y - 2 \cdot z & = & 1 \end{array} \quad (\dagger)$$

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:

$$\begin{array}{rcl} 2 \cdot 1 + 2 \cdot 1 + 3 \cdot 2 & = & 10 \\ 2 \cdot 1 + 5 \cdot 1 + 12 \cdot 2 & = & 31 \\ 4 \cdot 1 + 1 \cdot 1 - 2 \cdot 2 & = & 1 \end{array}$$

The three equations reduce to

$$10 = 10, 31 = 31, \text{ and } 1 = 1.$$

```
; An SOE is a non-empty Matrix.
; constraint for (list r1 ... rn), (length ri) is (+ n 1)
; interpretation represents a system of linear equations

; An Equation is a [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 a [List-of Number]

(define M ; an SOE
 (list (list 2 2 3 10) ; an Equation
 (list 2 5 12 31)
 (list 4 1 -2 1)))

(define S '(1 1 2)) ; a Solution
```

Figure 167: A data representation for systems of equations

Figure 167 introduces a data representation for our problem domain. It includes an example of a system of equations and its solution. 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, it is convenient to use these functions:

```
; Equation -> [List-of Number]
; extracts the left-hand side from a row in a matrix
(check-expect (lhs (first M)) '(2 2 3))
(define (lhs e)
 (reverse (rest (reverse e)))))

; Equation -> Number
; extracts the right-hand side from a row in a matrix
(check-expect (rhs (first M)) 10)
(define (rhs e)
 (first (reverse e)))
```

**Exercise 462.** 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.

**Exercise 463.** 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 462](#).

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 equation from the second 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))
```

That is, 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.

Mathematics teaches how to prove such facts. We use them.

**Exercise 464.** Check that the following system of equations

$$\begin{array}{rcl} 2 \cdot x & + & 2 \cdot y & + & 3 \cdot z = 10 \\ & & 3 \cdot y & + & 9 \cdot z = 21 \\ - & 3 \cdot y & - & 8 \cdot z & = -19 \end{array} \quad (\ddagger)$$

has the same solution as the one labeled with  $(\dagger)$ . Again do so by hand and with check-solution from [exercise 462](#).

**Exercise 465.** Design `subtract`. The function consumes two [Equations](#) of equal length. It “subtracts” a multiple of the second equation from the first, item by item, so that the resulting [Equation](#) has 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**, so we can 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. Doing so yields

```
(list (list 3 9 21)
 (list 1 2))
```

The rest of this SOE is a single equation and cannot be simplified.

**Exercise 466.** Here is a representation for triangular SOEs:

```
; A TM is an [NEList-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 466** 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 for these equations.

**Exercise 467.** Revise the algorithm `triangulate` from [exercise 466](#) 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 [SOEs](#) don't have a solution. Consider this one:

$$\begin{array}{rcl} 2 \cdot x + 2 \cdot y + 2 \cdot z & = & 6 \\ 2 \cdot x + 2 \cdot y + 4 \cdot z & = & 8 \\ 2 \cdot x + 2 \cdot y + 1 \cdot z & = & 2 \end{array}$$

If you try to triangulate this SOE—by hand or with your solution from [exercise 467](#)—you arrive at an intermediate matrix all of whose equations start with  $0$ :

$$\begin{array}{rcl} 0 \cdot x + 0 \cdot y + 2 \cdot z & = & 6 \\ 0 \cdot x + 0 \cdot y - 1 \cdot z & = & 0 \end{array}$$

**Exercise 468.** Modify `triangulate` from [exercise 467](#) 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 463](#), 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  $y$ :

$$y = (21 - 9 \cdot 2)/3 .$$

Now that we have  $z = 2$  and  $y = 1$ , we can plug these values into the first equation:

$$2 \cdot x + 2 \cdot 1 + 3 \cdot 2 = 10 .$$

This yields another equation in a single variable, which we solve like this:

$$x = (10 - (2 \cdot 1 + 3 \cdot 2))/2 .$$

This finally yields a value for  $x$  and thus the complete solution for the entire SOE.

**Exercise 469.** Design the `solve` function. It consumes triangular SOEs 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 design `solve`.

**Exercise 470.** Define `gauss`, which combines the `triangulate` function from [exercise 468](#) and the `solve` function from [exercise 469](#).

---

## 29 Algorithms that Backtrack

Problem solving doesn't always progress along some straight line. Sometimes we may follow one approach and discover that we are stuck because we took a wrong turn. One obvious option is to backtrack to the place where we made the fateful decision 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.

---

### 29.1 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 who know each other. You have just created your first undirected graph.

Now take a look at [figure 168](#), 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 an 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 168](#), though it might end up looking much more complex, almost impenetrable.

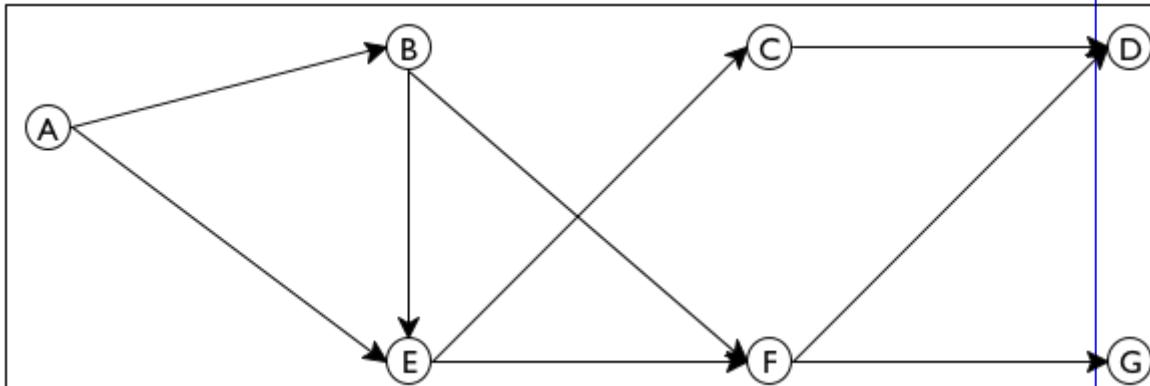


Figure 168: A directed graph

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. Similarly they use such graphs to predict the probable activities of people, even without knowledge of the content of their emails.

Mathematical scientists call the desired sequence a *path*.

Figure 168 makes 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 168, you cannot move from *C* to *G* by following the arrows.

Looking at figure 168 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 168 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 restart 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 168:

```
(define sample-graph
 '((A (B E))
 (B (E F))
 (C (D)))
 (D ()))
 (E (C F))
 (F (D G)))
```

```
(define sample-graph
 '((A B E)
 (B E F)
 (C D))
 (D))
 (E C F)
 (F D G))
```

(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 168](#). 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 471.** 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 which nodes it 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. Then the answer for the first

call would be `'()` because `'D` is an immediate neighbor of `'C`. Of course, `'()` could then no longer signal failure.

Concerning the lack-of-a-path issue, we must choose a distinct value for signaling this notion. Because `#false` is distinct, is 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 a [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 that consumes a list of nodes and generates 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 for it:

```
; [List-of Node] Node Graph -> [Maybe Path]
; finds a path from some node on lo-originations to
; destination; otherwise, it 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 471](#) 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 that `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) ...]
 [(cons? candidate) ...]))]))
```

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 169: Finding a path in a graph

Figure 169 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 its answer. Otherwise, `find-path/list` 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 a 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 168 and any two nodes in this graph, `find-path` always produces some answer. Stop! Solve the next exercise before you read on.

**Exercise 472.** Test `find-path`. Use the function to find a path from '`A`' to '`G`' in `sample-graph`. Which one does it find? Why?

Design `test-on-all-nodes`, a function that consumes a graph `g` and determines whether there is a path between any pair of nodes.

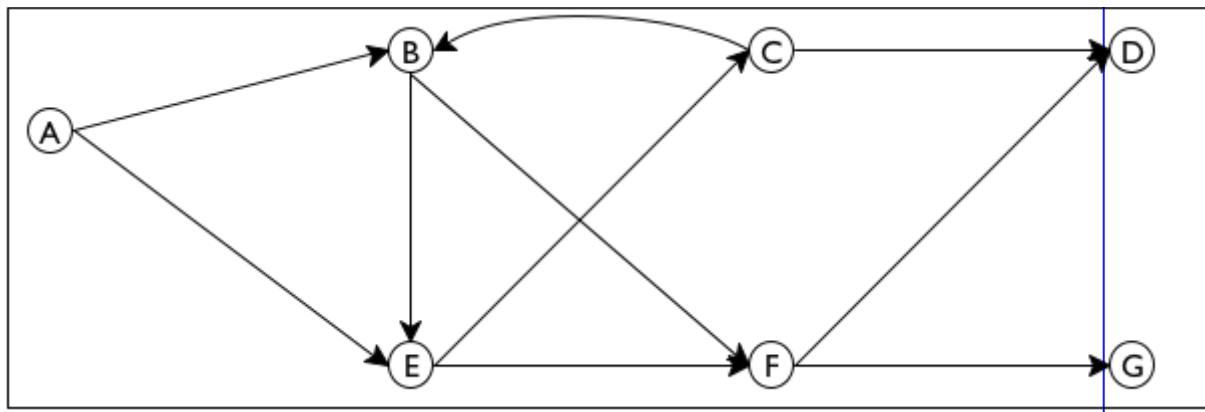


Figure 170: A directed graph with cycle

For other graphs, however, `find-path` may not terminate for certain pairs of nodes. Consider the graph in figure 170.

Stop! Define `cyclic-graph` to represent the graph in this figure.

Compared to figure 168, this new graph contains only one extra edge, from  $C$  to  $B$ . This seemingly small addition, though, allows us to start a search in a node and to return to the same node. 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 this graph, it fails to stop, as a 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 that it started with. Since the same input triggers the same evaluation for any function, `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 473.** Test `find-path` on ' $B$ ', ' $C$ ', and the graph in figure 170. Also use `test-on-all-nodes` from exercise 472 on this graph.

**Exercise 474.** Redesign the `find-path` program as a single function.

**Exercise 475.** Redesign `find-path/list` so that it uses an existing list abstraction from figures 95 and 96 instead of explicit structural recursion. **Hint** 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**.

As [Abstraction](#) says, data abstraction works just like function 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 476.** [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])

; An FSM is a structure:
; (make-fsm FSM-State [List-of 1Transition] FSM-State)
; A 1Transition is a structure:
; (make-transition FSM-State 1String FSM-State)
; An FSM-State is String.

; data example: see exercise 109

(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 109](#), "acbd", "ad", and "abcd" are examples of acceptable strings; "da", "aa", or "d" do not match.

In this context, you are designing the following function:

```
; FSM String -> Boolean
; does an-fsm recognize the given string
(define (fsm-match? an-fsm a-string)
 #false)
```

**Hint** Design the necessary auxiliary function locally to the `fsm-match?` function. In this context, represent the problem as a pair of parameters: the current state of the finite state machine and the remaining list of `1Strings`.

```
; [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)]))

(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 171: A definition of `arrangements` using generative recursion

**Exercise 477.** Inspect the function definition of `arrangements` in [figure 171](#). The figure displays a generative-recursive solution of the extended design problem covered by [Word Games, the Heart of the Problem](#), namely

given a word, create all possible rearrangements of the letters.

We thank Mark Engelberg for suggesting this exercise.

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 171](#) 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 171](#) create the same lists as the solution of [Word Games, the Heart of the Problem?](#)

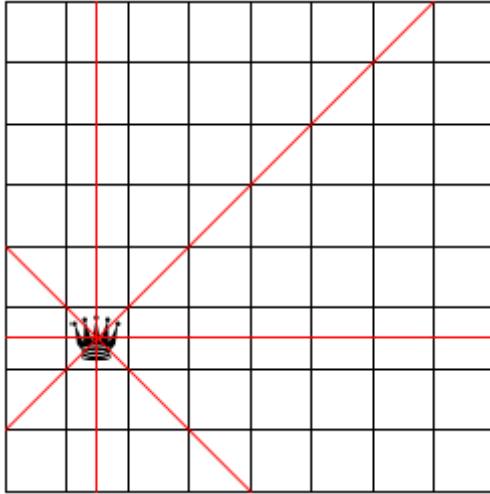


Figure 172: A chess board with a single queen and the positions it threatens

## 29.2 Project: Backtracking

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 chess board 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 172](#)

We thank Mark Engelberg for his reformulation of this section.

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.

The classical queens problem is to place 8 queens on an 8 by 8 chess board 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  $n$  by  $n$ , chess board such that the queens don’t pose a threat to each other.

For  $n = 2$ , the complete puzzle obviously has no solution. A queen placed on any of the four squares threatens all remaining squares.

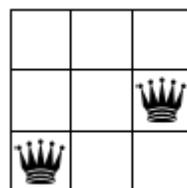
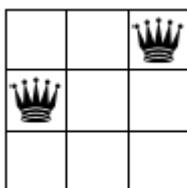
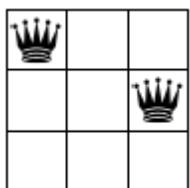


Figure 173: Three queen configurations for a 3 by 3 chess board

There is also no solution for  $n = 3$ . [Figure 173](#) presents all different placements of two queens, that is, solutions for  $k = 3$  and  $n = 2$ . In each case, the left queen occupies a square in the left 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 threatens all remaining, unoccupied squares, meaning it is impossible to place a third queen.

**Exercise 478.** You can also 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 173](#).

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?

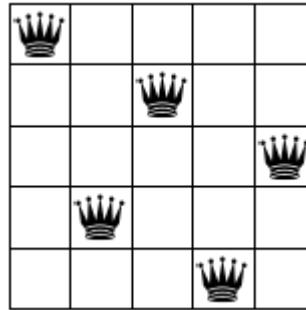
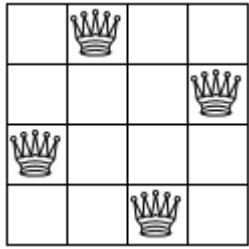


Figure 174: Solutions for the  $n$  queens puzzle for 4 by 4 and 5 by 5 boards

[Figure 174](#) displays two solutions for the  $n$  queens puzzle: the left one is for  $n = 4$ , the right one for  $n = 5$ . The figure shows how in each case a solution has one queen in each row and column, 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, 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. For another queen, we consider only nonthreatened spots.
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, this solution process is like the “find a path” algorithm.

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:

```
(define QUEENS 8)
; A QP is a structure:
; (make-posn CI CI)
```

```

; A CI is an N in [0,QUEENS).
; interpretation (make-posn r c) denotes the square at
; the r-th row and c-th column

```

After all, the chess board basically dictates the choice.

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 dimensions, 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 479.** 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 172](#). 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 remain 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 diagonals. Don't forget to include arguments for which `threatening?` must produce `#false`.

**Exercise 480.** Design `render-queens`. The function consumes a natural number `n`, a list of `QPs`, and an [Image](#). It produces an image of an `n` by `n` chess board with the given image placed according to the given `QPs`.

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 is another exercise in data abstraction. Indeed, a data definition for `Board` isn't even necessary to state the signature for the algorithm proper:

```

; N -> [Maybe [List-of QP]]
; finds a solution to the n queens problem

; data example: [List-of QP]
(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)

```

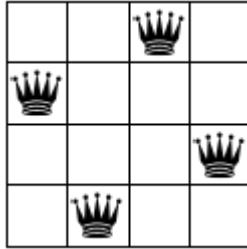
The complete puzzle is about finding a placement for  $n$  queens on an  $n$  by  $n$  chess board. So clearly, the algorithm consumes nothing else but a natural number, and it produces a

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 174](#) 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 represent these two images. [Figure 175](#) sketches some. Fill in the rest.

```
; N -> [Maybe [List-of QP]]
; finds a solution to the n queens problem

(define 0-1 (make-posn 0 1))
(define 1-3 (make-posn 1 3))
(define 2-0 (make-posn 2 0))
(define 3-2 (make-posn 3 2))

(check-member-of
 (n-queens 4)
 (list 0-1 1-3 2-0 3-2)
 (list 0-1 1-3 3-2 2-0)
 (list 0-1 2-0 1-3 3-2)
 (list 0-1 2-0 3-2 1-3)
 (list 0-1 3-2 1-3 2-0)
 (list 0-1 3-2 2-0 1-3)
 ...
 (list 3-2 2-0 1-3 0-1))

(define (n-queens n)
 (place-queens (board0 n) n))
```

Figure 175: Solutions for the 4 queens puzzle

**Exercise 481.** The tests in [figure 175](#) 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 contain 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 one 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 482.** The key idea to is to design a function that places *n* queens on a chess board that may already contain some queens:

```
; Board N -> [Maybe [List-of QP]]
; places n queens on board; otherwise, returns #false
(define (place-queens a-board n)
 #false)
```

[Figure 175](#) 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 175](#) 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. It calls for a data representation of **Boards** that supports the three functions on the wish list. This, then, is your remaining problem.

**Exercise 483.** Develop a data definition for **Board** and design the three functions specified in [exercise 482](#). 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. Use a structure with three fields to represent a square: one for  $x$ , 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 482](#) and confirm that it works with any of your data representations of **Board**.

---

## 30 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 will 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 specialists, 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.

## Intermezzo 5: 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 respond with this:

```
(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 some inputs.

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,

You may also wish to reread [Local Definitions](#) and the discussion of integrity checks in [Project: Database](#).

`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.

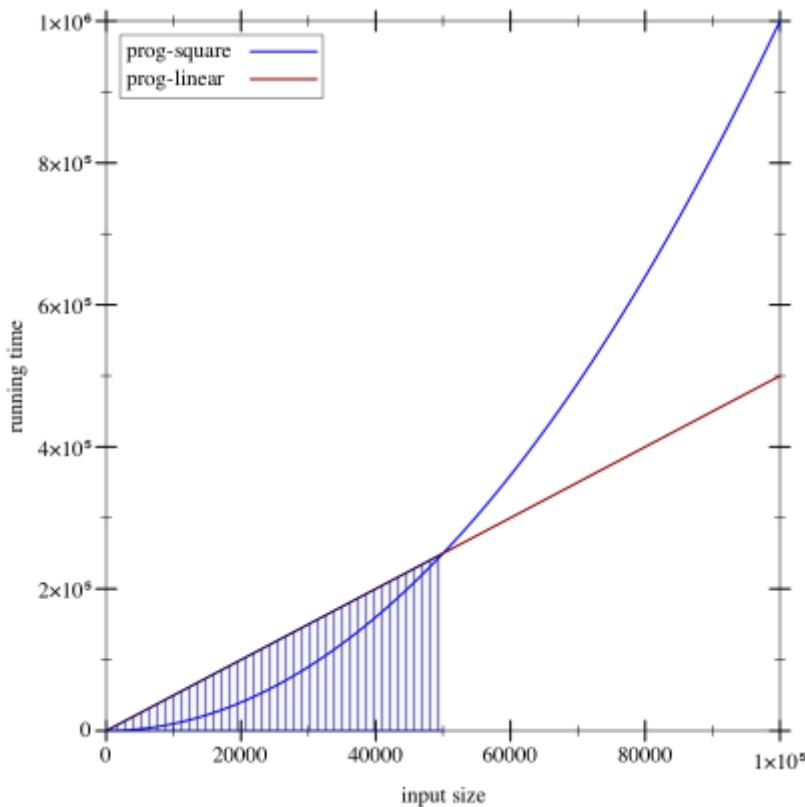


Figure 176: A comparison of two running time expressions

How confident are you that you wish to use `prog-linear` instead of `prog-square`? Consider the graph in [figure 176](#). 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 more 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 worse 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 shaded region and if your clients were to run your program mostly on inputs that fall in the nonshaded region, you would be delivering the wrong program.

This intermezzo introduces the idea of *algorithmic analysis*, which allows programmers to make general statements about a program’s performance and everyone else about the growth of a function. Any serious programmer and scientist must eventually become thoroughly familiar with this notion. It is the basis for analyzing performance attributes of programs. To understand the idea properly, you will need to work through a text book.

We thank Prabhakar Ragde for sharing his notes on connecting the first edition of this book with algorithmic 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 1: Beginning Student Language](#), we can look at this process 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

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.

“Abstract” because the measure ignores the details of how much time primitive steps take.

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? lo-names)
 (cond
 [(empty? lo-names) #false]
 [(cons? lo-names)
 (or (string=? (first lo-names) 'flatt)
 (contains-flatt? (rest lo-names))))]))
```

For a list that starts with '`flatt`', say,

```
(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, as in,

```
(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, an *average analysis* starts from the ideas 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?` finds, on the average, '`flatt`' somewhere in the middle of the input list. Thus, if the latter consists of  $n$  items, the average running time of `contains-flatt?` is  $n/2$ , that is, it recurs half as often as there are 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  $n/2$  steps” brings us to one more characteristic of abstract running time. Because it ignores the exact time it takes to evaluate primitive computation steps—checking predicates, selecting values, picking `cond` clauses—we can drop the division by 2. Here is why. By assumption, each basic step takes  $k$  units of time, meaning `contains-flatt?` takes time

$$k \cdot \frac{1}{2} \cdot n.$$

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:

$$k \cdot \frac{1}{2} \cdot n = \frac{1}{2} \cdot k \cdot n = c \cdot n$$

that is, the abstract running time is always  $n$  multiplied by a constant, and that’s all that matters to say “on the order of  $n$ .”

Now consider our sorting program from [figure 72](#). 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 one, so it is not surprising that the performance of `insert` and `contains-flatt?` are alike. The applications of `insert` to a list of  $l$  items triggers between 0 and  $l$  recursive steps. On the average, we assume it requires  $l/2$ , which means that `insert` takes “on the order of  $l$  steps” where  $l$  is the length of the given list.

The question is how long 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

$$\frac{1}{4} \cdot n^2 - \frac{1}{4} \cdot n$$

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 486](#) ask you to argue why it is correct to simplify this claim in this way.

See [exercise 486](#) for 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 486](#) for details. Note This analysis assumes that comparing two items on the list takes a fixed amount of time. End

Our final example is the `inf` program from [Local 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 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 484.** 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))`. 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 261](#), which asks you to explore a similar problem.

**Exercise 485.** 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.

How do you decide whether one such function is "better" than another?

[Exercise 245](#) tackles a different question, namely, whether we can formulate a program that decides whether two other programs are equal. In this intermezzo, we are not writing a program; we are using plain mathematical arguments.

Let's return to the imaginary programs from the introduction: `prog-linear` and `prog-square`. They compute the same results but their performance differs. The `prog-linear` program requires "on the order of  $n$  steps" while `prog-square` 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 176](#), 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 are 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

$$\text{for all } n \geq bigEnough \text{ it is true that } f(n) \leq c \cdot g(n).$$

**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  $H \in O(G)$ , 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:

$$bigEnough = 1000, c = 1.$$

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 spelled out:

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  $c$  by working your way backward 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.

Finally, most people use  $O$  together with a short-hand 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 486.** 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 487.** Consider the functions  $f(n) = 2^n$  and  $g(n) = 1000 n$ . 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 488.** Compare  $f(n) = n \log(n)$  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) (define (searchS x l)
 (cond (cond
 [(empty? l) #false] [= (length l) 0] #false]
 [else [else
 (or (= (first l) x) (or (= (first l) x)
 (searchL (searchS
 x (rest l))))))) 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 whose second one deals with all others. The question in the first `cond` line uses `empty?` and the second one uses `cons?` of `else`.

The design of `searchS` 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.

It really 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 `length` is more like `how-many`, however, `searchS` is going to be 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 checks whether `length` is a constant-time function 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 Number Number]
; 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 `10000` and `20000`. If `length` is like `empty?`, the times for the second run will be roughly twice those of the first one; otherwise, the time for `searchS` will increase dramatically.

Stop! Conduct the experiment.

Assuming you have completed the experiment, you now know that `length` takes time proportionally to the size of the given list. 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 this is done requires one more design concept, accumulators, the concern of the final part of this book.

See [Data Representations with Accumulators](#) for how other languages track the size of a container.

---

## 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 true 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.

While context-independence facilitates the design of functions, it causes two problems. In general, 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 performance cost. For functions that employ generative recursion, the loss means that the function may not be able to compute the result at all. 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 or where it is evaluated, the 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 regular arguments while accumulators change in relation to those and the context.

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.

---

### 31 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

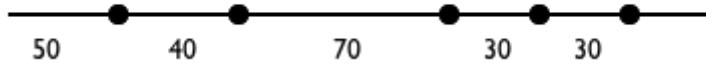
functions. While the first section is about structural recursion, the second one addresses concerns in the generative realm.

## 31.1 A Problem with Structural Processing

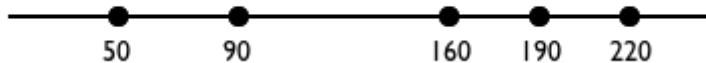
Let's start with a seemingly straightforward example:

**Sample Problem** You are working for a geometer team that will measure the length of road segments. The team asked you to design a program that translates these relative distances between a series of road points into absolute distances from 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 a mere exercise in structural function design. [Figure 177](#) 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 one on (`rest l`). Because the first 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]
; converts a list of relative to absolute distances
; the first number 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]
; adds 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 177: 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:

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 5: 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 489.** Reformulate `add-to-each` using `map` and `lambda`.

The times will differ from computer to computer and year to year. These measurements were conducted in 2017 on a MacMini running OS X 10.11; the previous measurement took place in 1998, and the times were 100x larger.

**Exercise 490.** Develop a formula that describes 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 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 steps along the line. Why can't a program do so?

Let's attempt to design a version of the function that is close to our manual method. We still start from the list-processing template:

```

(define (relative->absolute/a l)
 (cond
 [(empty? l) ...]
 [else
 (... (first l) ...
 ... (relative->absolute/a (rest l)) ...)]))

```

Now let's simulate a hand-evaluation:

```

(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.

Again, the problem is that recursive functions are independent of their context. A function processes L in (cons N L) the same way as in (cons K L). Indeed, if given L by itself, it would also process the list in that way.

To make up for the loss of “knowledge,” we equip the function with an additional parameter: accu-dist. The latter 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 traverses the list, it must add its numbers 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 both arguments are changing for every call, the change in the second one strictly depends on the first argument. The function is still a plain list-processing procedure.

Now let’s evaluate our running example again:

```

(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 ???)))
== (cons 3 (cons 5 (cons 12 '())))

```

Stop! Fill in the question marks in line 4.

The hand-evaluation shows just how much the use of an accumulator simplifies the conversion process. 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 problem 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 178](#) shows the result. Now, `relative->absolute` is indistinguishable from `relative->absolute.v2` with respect to input-output.

```

; [List-of Number] -> [List-of Number]
; converts a list of relative to absolute distances
; the first number 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 178: 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 491.** With a bit of design and a bit of tinkering, a friend of yours came up with the following solution for the sample problem:

Adrian German and Mardin Yadegar suggested this exercise.

```
(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 an answer, see [Recognizing the Need for an Accumulator](#), but reflect on the question first. **Hint** Try to design `reverse` on your own.

## 31.2 A Problem with Generative Recursion

Let's 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 such a graph, each node has exactly one, directional connection to another, and possibly itself.

[Algorithms that Backtrack](#) covers 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 179](#). There are six nodes, *A* through *F*, and six connections. A path from *A* to *E* must contain *B* and *C*. There is no path, though, from *A* to *F* or from any other node besides itself.

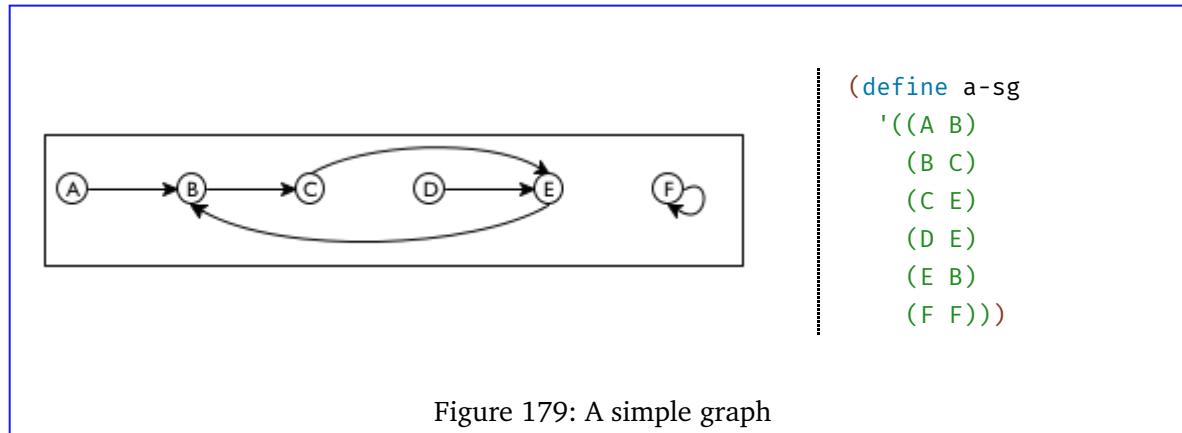


Figure 179: A simple graph

The right part of [figure 179](#) 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 a list of two items:
; (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 origin to destination
; in the simple graph sg

(check-expect (path-exists? 'A 'E a-sg) #true)
(check-expect (path-exists? 'A 'F a-sg) #false)

(define (path-exists? origin destination sg)
 #false)
```

What we need are answers to the four basic questions of the recipe for generative recursion:

- The problem is trivial if `origin` is the same as `destination`.
- The trivial solution is `#true`.
- If `origin` 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 `origin`'s neighbor is connected to `destination`, then so is `origin`. 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 origin to destination in sg

(check-expect (path-exists? 'A 'E a-sg) #true)
(check-expect (path-exists? 'A 'F a-sg) #false)

(define (path-exists? origin destination sg)
 (cond
 [(symbol=? origin destination) #t]
 [else (path-exists? (neighbor origin sg)
 destination
 sg)]))

; Node SimpleGraph -> Node
; determine the node that is connected to a-node in sg
(check-expect (neighbor 'A a-sg) 'B)
(check-error (neighbor 'G a-sg) "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 180: Finding a path in a simple graph

[Figure 180](#) contains the complete program, including the function for finding the neighbor of a node in a simple graph—a straightforward exercise in structural recursion—and test cases for both possible outcomes. 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 `origin` 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 179](#). 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) *** (F F)))
== (path-exists? 'E 'D '((A B) *** (F F)))
== (path-exists? 'B 'D '((A B) *** (F F)))
== (path-exists? 'C 'D '((A 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` above. Like `relative->absolute`, the design of `path-exists?` uses a recipe and assumes that recursive calls are independent of their context. 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 starter nodes that the function has encountered, starting with the original application. Its initial value must be `'()`. As the function checks on a specific `origin` and moves to its neighbors, `origin` 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 origin to destination
; assume there are no paths for the nodes in seen
(define (path-exists?/a origin destination sg seen)
 (cond
 [(symbol=? origin destination) #true]
 [else (path-exists?/a (neighbor origin sg)
 destination
 sg
 (cons origin 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) *** (F F)) '())
```

shows, it provides the foundation for one:

```
-- (path-exists?/a 'E 'D '((A B) *** (F F)) '(C))
```

```

== (path-exists?/a 'B 'D '((A B) *** (F F)) '(E C))
== (path-exists?/a 'C 'D '((A 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 tells us 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 `origin` is already an item in `seen`. If so, the problem is also trivially solvable, yielding `#false` as the solution. [Figure 181](#) 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 origin to destination in sg

(check-expect (path-exists.v2? 'A 'E a-sg) #true)
(check-expect (path-exists.v2? 'A 'F a-sg) #false)

(define (path-exists.v2? origin destination sg)
 (local (; Node Node SimpleGraph [List-of Node] -> Boolean
 (define (path-exists?/a origin seen)
 (cond
 [(symbol=? origin destination) #t]
 [(member? origin seen) #f]
 [else (path-exists?/a (neighbor origin sg)
 (cons origin seen))]))
 (path-exists?/a origin '())))

```

Figure 181: 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 signature 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 492.** Modify the definitions in [figure 169](#) so that the program produces `#false`, even if it encounters the same `origin` twice.

---

## 32 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; and
2. an understanding of what the accumulator represents.

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.

```
; [List-of X] -> [List-of X]
; constructs the reverse of a lox

(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]
; adds an-x to the end of a lox

(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))))])
```

Figure 182: Design with accumulators, a structural example

---

## 32.1 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 ones. In either case, it is critical that we first built a complete function based on a **conventional** design recipe. Then we study the function and proceed as follows:

1. If a structurally recursive function traverses 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` in [figure 182](#). The result of the recursive application produces the reverse of the rest of the list. It uses `add-as-last` to add the first item to this reversed list and thus creates the reverse of the entire list. This second, auxiliary function is also recursive. We have thus identified an accumulator candidate.

It is now time to study some hand-evaluations, as in [A Problem with Structural Processing](#), to see whether an accumulator helps. Consider the following:

```
(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 '(c)))
== (add-as-last 'a '(c b))
== '(c b a)
```

Stop! Replace the dots with the two missing steps. Then you can see that `invert` eventually 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 493.** Argue that, in the terminology of [Intermezzo 5: The Cost of Computation](#), `invert` consumes  $O(n^2)$  time when the given list consists of  $n$  items.

**Exercise 494.** Does the insertion `sort>` function from [Auxiliary Functions that Recur](#) need an accumulator? If so, why? If not, why not?

---

## 32.2 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 will want to proceed as follows.

1. Create an accumulator template:

```
; Domain -> Range
(define (function d0)
 (local (; Domain AccuDomain -> 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 often called an *invariant*.

## 3. Use the invariant to determine the initial value `a0` for `a`.

## 4. Also exploit the invariant 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 `function/a`.

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:

```
(define (invert.v2 alox0)
 (local (; [List-of X] ??? -> [List-of X]
 ; constructs 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 example 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]
 ; constructs 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` merely traverses the list. In contrast, `invert` reprocesses every result of its natural recursion with `add-as-last`. Stop! Measure how much faster `invert.v2` runs than `invert`.

**Terminology** Programmers use the phrase *accumulator-style function* to discuss functions that use an accumulator parameter. Examples of functions in accumulator style are `relative-absolute/a`, `invert/a`, and `path-exists?/a`.

---

### 32.3 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 calls 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 these definitions of the `sum` function:

```
(define (sum.v1 alon)
 (cond
```

```

[(empty? alon) 0]
[else (+ (first alon) (sum.v1 (rest alon))))]
)

```

Here is the first step toward an accumulator version:

```

(define (sum.v2 alon0)
 (local (; [List-of Number] ??? -> Number
 ; computes the sum of the numbers on alon
 ; accumulator ...
 (define (sum/a alon a)
 (cond
 [(empty? alon) ...]
 [else (... (sum/a (rest alon)
 a ...) ...)])))
 (sum/a alon0 ...)))

```

Stop! Supply a signature and a test case that works for both.

As suggested by our first step, we have put the template for `sum/a` into a `local` definition, added an accumulator parameter, and renamed the parameter of `sum`.

<code>(sum.v1 '(10 4))</code> <code>== (+ 10 (sum.v1 '(4)))</code> <code>== (+ 10 (+ 4 (sum.v1 '())))</code> <code>== (+ 10 (+ 4 (+ 0)))</code> <code>...</code> <code>== 14</code>	<code>(sum.v2 '(10 4))</code> <code>== (sum/a '(10 4) a0)</code> <code>== (sum/a '(4) ... 10 ... a0)</code> <code>== (sum/a '() ... 4 ... 10 ... a0)</code> <code>...</code> <code>== 14</code>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 183: Calculating with accumulator-style templates

Figure 183 shows two side-by-side sketches of hand-evaluations. A comparison immediately suggests the central idea, namely, that `sum/a` can use the accumulator to add up the numbers it encounters. Concerning the accumulator invariant, the calculations suggest `a` represents the sum of the numbers encountered so far:

`a` is the sum of the numbers that `alon` lacks from `alon0`

For example, the invariant forces the following relationships to hold:

<code>if      alon0                is    '(10 4 6)    '(10 4 6)    '(10 4 6)</code> <code>and     alon                is    '(4 6)                '(6)                '()</code> <code>then    a    should be        10                14                20</code>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Given this precise invariant, the rest of the design is straightforward:

```

(define (sum.v2 alon0)
 (local (; [List-of Number] ??? -> Number
 ; computes the sum of the numbers on alon
 ; accumulator a is the sum of the numbers
 ; that alon lacks from alon0
 (define (sum/a alon a)
 (cond
 [(empty? alon) a]
 [else (... (sum/a (rest alon)
 a ...) ...)])))
 (sum/a alon0 ...)))

```

```

[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 495.** Complete the manual evaluation of `(sum/a '(10 4) 0)` in [figure 183](#). 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 5: The Cost of Computation](#).

For the second example, we turn to the well-known factorial function:

```

; N -> N
; computes (* n (- n 1) (- n 2) ... 1)
(check-expect (!.v1 3) 6)

```

The factorial function is useful for the analysis  
of algorithms.

```

(define (!.v1 n)
 (cond
 [(zero? n) 1]
 [else (* n (!.v1 (sub1 n)))]))

```

While `relative-2-absolute` and `invert` processed lists, the factorial function works on natural numbers, which its template reflects.

We proceed as before with a template for an accumulator-style version:

```

(define (!.v2 n0)
 (local (; N ??? -> N
 ; computes (* 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:

$  \begin{aligned}  & (!.v1 3) \\  & = ( * 3 (!.v1 2) ) \\  & = ( * 3 ( * 2 (!.v1 1) ) ) \\  & \dots \\  & = 6  \end{aligned}  $	$  \begin{aligned}  & (!.v2 3) \\  & = (!/a 3 a0) \\  & = (!/a 2 ... 3 ... a0) \\  & \dots \\  & = 6  \end{aligned}  $
----------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------

The left column indicates how the original version works; the right one sketches how the accumulator-style function proceeds. Both structurally traverse a natural number until they reach `0`. While the original version schedules only multiplications, the accumulator keeps track of each number as the structural traversal 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 496.** 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 multiplying the current accumulator with  $n$  is the proper update operation:

```
(define (!.v2 n0)
 (local (; N N -> N
 ; computes (* 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 497.** Like `sum`, `!.v1` performs the primitive computations, 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 `(!.v1 20)` 1,000 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 commonly used for complicated data definitions as it is for lists and natural numbers.

Here are the relevant definitions:

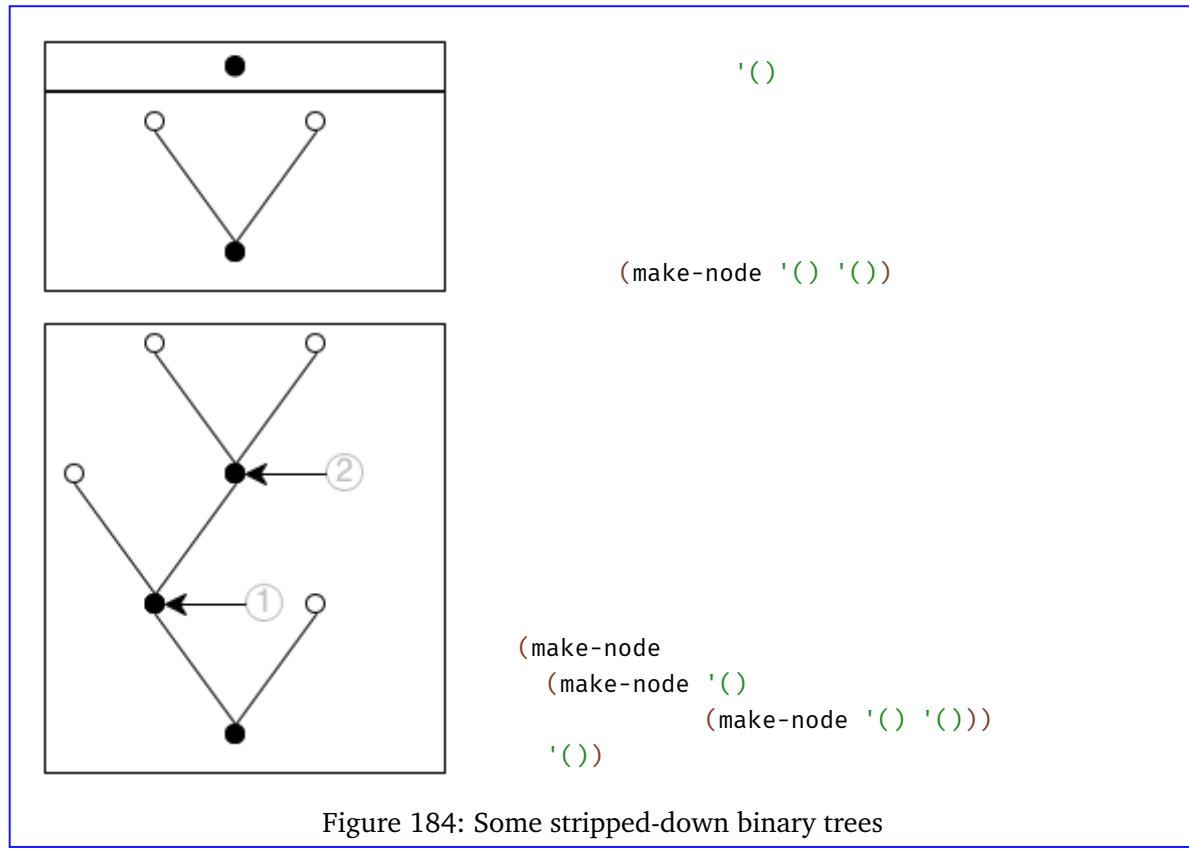
```
(define-struct node [left right])
; A Tree is one of:
; - '()
; - (make-node Tree Tree)
(define example
 (make-node (make-node '() (make-node '() '())) '()))
```

These trees carry no information; their leafs are '`()`'. Still, there are many different trees, as figure 184 shows; it also uses suggestive graphics to bring across what these pieces of data look like as trees.

One property that one may wish to compute is the height of such a tree:

```
(define (height abt)
 (cond
 [(empty? abt) 0]
 [else (+ (max (height (node-left abt))
 (height (node-right abt))) 1)])))
```

Stop! Supply a signature and a test. The table in figure 184 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.



To transform this function into an accumulator-style function, we follow the standard path. We begin with an appropriate template:

```
(define (height.v2 abt0)
 (local (; Tree ??? -> N
 ; measures 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 184](#). 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 to 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
 ; measures 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 dictates 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.

```

; Tree -> N
; measures the height of abt0
(check-expect (height.v2 example) 3)
(define (height.v2 abt0)
 (local (; Tree N -> N
 ; measures 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 185: The accumulator-style version of *height*

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 185](#) for the complete definition.

**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, something we have not encountered before:

```

... ; Tree N N -> N
; measures 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 ...) ...)])) ...

```

**Exercise 498.** Complete the design of `height.v3`. **Hint** In terms of the bottom-most tree of [figure 184](#), 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 design both variants systematically, following the same design recipe. When you have complete function definitions, you can compare and contrast the results, and you can then decide which one to keep, based on evidence. **End**

**Exercise 499.** Design an accumulator-style version of `product`, the function that computes the product of a list of numbers. Stop when you have formulated the accumulator invariant and have someone check it.

The performance of `product` is  $O(n)$  where  $n$  is the length of the list. Does the accumulator version improve on this?

**Exercise 500.** Design an accumulator-style version of `how-many`, which is the function that determines the number of items on a list. Stop when you have formulated the invariant and have someone check it.

The performance of `how-many` is  $O(n)$  where  $n$  is the length of the list. Does the accumulator version improve on this?

Computer scientists refer to this space as *stack space*, but you can safely ignore this terminology for now.

When you evaluate `(how-many some-non-empty-list)` by hand,  $n$  applications of `add1` are pending by the time the function reaches `'( )`—where  $n$  is the number of items on the list. Computer scientists sometime say that `how-many` needs  $O(n)$  space to represent these pending function applications. Does the accumulator reduce the amount of space needed to compute the result?

**Exercise 501.** Design an accumulator-style version of `add-to-pi`. The function adds a natural number to `pi` without using `+`:

```
; N -> Number
; adds n to pi without using +
(check-within (add-to-pi 2) (+ 2 pi) 0.001)
(define (add-to-pi n)
 (cond
 [(zero? n) pi]
 [else (add1 (add-to-pi (sub1 n))))]))
```

Stop when you have formulated the accumulator invariant and have someone check it.

**Exercise 502.** Design the function `palindrome`, which accepts a non-empty list and constructs a palindrome by mirroring the list around the last item. When given `(explode "abc")`, it yields `(explode "abcba")`.

**Hint** Here is a solution designed by function composition:

```
; [NEList-of 1String] -> [NEList-of 1String]
; creates a palindrome from s0
(check-expect
 (mirror (explode "abc")) (explode "abcba"))
(define (mirror s0)
```

```
(append (all-but-last s0)
 (list (last s0))
 (reverse (all-but-last s0))))
```

See [Auxiliary Functions that Generalize](#) for `last`; design `all-but-last` in an analogous manner.

This solution traverses `s0` four times:

1. via `all-but-last`,
2. via `last`,
3. via `all-but-last` again, and
4. via `reverse`, which is ISL+'s version of `invert`.

Even with `local` definition for the result of `all-but-last`, the function needs three traversals. While these traversals aren't "stacked" and therefore don't have a disastrous impact on the function's performance, an accumulator version can compute the same result with a single traversal.

**Exercise 503.** [Exercise 467](#) implicitly asks for the design of a function that rotates a `Matrix` until the first coefficient of the first row differs from `0`. In the context of [Exercise 467](#), the solution calls for a generative-recursive function that creates a new matrix by shifting the first row to the end when it encounters a `0` in the first position. Here is the solution:

```
; Matrix -> Matrix
; finds a row that doesn't start with 0 and
; uses it as the first one
; generative moves the first row to last place
; no termination if all rows start with 0
(check-expect (rotate-until.v2 '((0 4 5) (1 2 3)))
 '((1 2 3) (0 4 5)))
(define (rotate M)
 (cond
 [(not (= (first (first M)) 0)) M]
 [else
 (rotate (append (rest M) (list (first M))))]))
```

Stop! Modify this function so that it signals an error when all rows start with `0`.

If you measure this function on large instances of `Matrix`, you get a surprising result:

rows in M	1000	2000	3000	4000	5000
<code>rotate</code>	17	66	151	272	436

As the number of rows increases from 1,000 to 5,000, the time spent by `rotate` does not increase by a factor of five but by twenty.

The problem is that `rotate` uses `append`, which makes a brand-new list like `(rest M)` only to add `(first M)` at the end. If `M` consists of 1,000 rows and the last row is the only one with a non-`0` coefficient, that's roughly

$$1,000 \cdot 1,000 = 1,000,000$$

lists. How many lists do we get if  $M$  consists of 5,000 lines?

Now suppose we conjecture that the accumulator-style version is faster than the generative one. Here is the accumulator template for a structurally recursive version of `rotate`:

```
(define (rotate.v2 M0)
 (local (; Matrix ... -> Matrix
 ; accumulator ...
 (define (rotate/a M seen)
 (cond
 [(empty? M) ...]
 [else (... (rotate/a (rest M)
 ... seen ...)
 ))])
 (rotate/a M0)))
```

The goal is to remember the first row when its leading coefficient is `0` without using `append` for every recursion.

Formulate an accumulator statement. Then follow the accumulator design recipe to complete the above function. Measure how fast it runs on a `Matrix` that consists of rows with leading `0`s except for the last one. If you completed the design correctly, the function is quite fast.

**Exercise 504.** Design `to10`. It consumes a list of digits and produces the corresponding number. The first item on the list is the **most significant** digit. Hence, when applied to `'(1 0 2)`, it produces `102`.

**Domain Knowledge** You may recall from grade school that the result is determined by  $1 \cdot 10^2 + 0 \cdot 10^1 + 2 \cdot 10^0 = ((1 \cdot 10 + 0) \cdot 10) + 2 = 102$ .

**Exercise 505.** Design the function `is-prime`, which consumes a natural number and returns `#true` if it is prime and `#false` otherwise.

**Domain Knowledge** A number  $n$  is prime if it is not divisible by any number between  $n - 1$  and 2.

**Hint** The design recipe for `N [≥=1]` suggests the following template:

```
; N [≥=1] -> Boolean
; determines whether n is a prime number
(define (is-prime? n)
 (cond
 [(= n 1) ...]
 [else (... (is-prime? (sub1 n))))])
```

This template immediately tells you that the function forgets  $n$ , its initial argument as it recurs. Since  $n$  is definitely needed to determine whether  $n$  is divisible by  $(- n 1)$ ,  $(- n 2)$ , and so on, you know that you need an accumulator-style function.

**Note on Speed** Programmers who encounter accumulator-style functions for the first time often get the impression that they are always faster than their plain counterparts. So let's take a look at the solution of [exercise 497](#):

An explanation of these times is beyond the scope of this book.

	! .v1	5.760	5.780	5.800	5.820	5.870	5.806
	! .v2	5.970	5.940	5.980	5.970	6.690	6.111

The table's top row shows the number of seconds for five runs of `(! .v1 20)`, while the bottom one lists those of running `(! .v2 20)`. The last column shows the averages. In short, the table shows that people jump to premature conclusions; the performance of at least one accumulator-style function is worse than that of the original. **Do not trust prejudices.** Instead, measure performance characteristics of your programs for yourself. **End**

**Exercise 506.** Design an accumulator-style version of `map`.

**Exercise 507.** [Exercise 257](#) explains how to design `foldl` with the design recipes and guidelines of the first two parts of the book:

```
(check-expect (f*fldl + 0 '(1 2 3))
 (foldl + 0 '(1 2 3)))
(check-expect (f*fldl cons '() '(a b c))
 (foldl cons '() '(a b c)))

; version 1
(define (f*fldl f e l)
 (foldr f e (reverse l)))
```

That is, `foldl` is the result of reversing the given list and then using `foldr` to fold the given function over this intermediate list.

The `f*fldl` function obviously traverses the list twice, but once we design all the functions, it becomes clear how much harder it has to work:

```
; version 2
(define (f*fldl f e l)
 (local ((define (reverse l)
 (cond
 [(empty? l) '()]
 [else (add-to-end (first l)
 (reverse (rest l)))]))
 (define (add-to-end x l)
 (cond
 [(empty? l) (list x)]
 [else (cons (first l)
 (add-to-end x (rest l))))]))
 (define (foldr l)
 (cond
 [(empty? l) e]
 [else (f (first l) (foldr (rest l))))]))
 (foldr (reverse l))))
```

We know that `reverse` has to traverse a list once for every item on the list, meaning `f*fldl` really performs  $n^2$  traversals for a list of length  $n$ . Fortunately, we know how to eliminate this

bottleneck with an accumulator:

```
; version 3
(define (f*ldl f e l)
 (local ((define (invert/a l a)
 (cond
 [(empty? l) a]
 [else (invert/a (rest l)
 (cons (first l) a))]))
 (define (foldr l)
 (cond
 [(empty? l) e]
 [else
 (f (first l) (foldr (rest l))))])
 (foldr (invert/a l '())))))
```

Once `reverse` uses an accumulator, we actually get the apparent performance of two traversals of the list. The question is whether we can improve on this by adding an accumulator to the locally defined `fold`:

```
; version 4
(define (f*ldl f e l0)
 (local ((define (fold/a a l)
 (cond
 [(empty? l) a]
 [else
 (fold/a (f (first l) a) (rest l))])))
 (fold/a e l0)))
```

Since equipping the function with an accumulator reverses the order in which the list is traversed, the initial reversal of the list is superfluous.

**Task 1** Recall the signature for `foldl`:

```
; [X Y] [X Y -> Y] Y [List-of X] -> Y
```

It is also the signature of `f*ldl`. Formulate the signature for `fold/a` and its accumulator invariant. **Hint** Assume that the difference between `l0` and `l` is `(list x1 x2 x3)`. What is `a`, then?

You may also be wondering why `fold/a` consumes its arguments in this unusual order, first the accumulator and then the list. To understand the reason for this ordering, imagine instead that `fold/a` also consumes `f`—as the first argument. At this point it becomes abundantly clear that `fold/a` is `foldl`:

```
; version 5
(define (f*ldl f i l)
 (cond
 [(empty? l) i]
 [else (f*ldl f (f (first l) i) (rest l))]))
```

**Task 2** Design `build-list` using an accumulator-style approach. The function must satisfy the following tests:

```
(check-expect (build-l*st n f) (build-list n f))
```

for any natural number  $n$  and function  $f$ .

---

## 32.4 A Graphical Editor, with Mouse

A [Graphical Editor](#) introduces the notion of a one-line editor and presents a number of exercises on creating a graphical editor. Recall that a graphical editor is an interactive program that interprets key events as editing actions on a string. In particular, when a user presses the left or right arrow keys, the cursor moves left or right; similarly, pressing the delete key removes a [1String](#) from the edited text. The editor program uses a data representation that combines two strings in a structure. [A Graphical Editor, Revisited](#) resumes these exercises and shows how the same program can greatly benefit from a different data structure, one that combines two strings.

Neither of these sections deals with mouse actions for navigation, even though all modern applications support this functionality. The basic difficulty with mouse events is to place the cursor at the appropriate spot. Since the program deals with a single line of text, a mouse click at  $(x,y)$  clearly aims to place the cursor between the letters that are visible at or around the  $x$  position. This section fills the gap.

Recall the relevant definitions from [A Graphical Editor, Revisited](#):

```
(define FONT-SIZE 11)
(define FONT-COLOR "black")

; [List-of 1String] -> Image
; renders a string as an image for the editor
(define (editor-text s)
 (text (implode s) FONT-SIZE FONT-COLOR))

(define-struct editor [pre post])
; An Editor is a structure:
; (make-editor [List-of 1String] [List-of 1String])
; interpretation if (make-editor p s) is the state of
; an interactive editor, (reverse p) corresponds to
; the text to the left of the cursor and s to the
; text on the right
```

**Exercise 508.** Design `split-structural` using the structural design recipe. The function consumes a list of [1Strings](#)  $ed$  and a natural number  $x$ ; the former represents the complete string in some [Editor](#) and the latter the  $x$ -coordinate of the mouse click. The function produces

```
(make-editor p s)
```

such that (1)  $p$  and  $s$  make up  $ed$  and (2)  $x$  is larger than the image of  $p$  and smaller than the image of  $p$  extended with the first [1String](#) on  $s$  (if any).

Here is the first condition expressed with an ISL+ expression:

```
(string=? (string-append p s) ed)
```

The second one is

```
(<= (image-width (editor-text p))
 x
 (image-width (editor-text (append p (first s))))))
```

assuming `(cons? s)`.

**Hints** (1) The x-coordinate measures the distance from the left. Hence the function must check whether smaller and smaller prefixes of ed fit into the given width. The first one that doesn't fit corresponds to the `pre` field of the desired [Editor](#), the remainder of ed to the `post` field.

(2) Designing this function calls for thoroughly developing examples and tests. See [Intervals, Enumerations, and Itemizations](#).

**Exercise 509.** Design the function `split`. Use the accumulator design recipe to improve on the result of [exercise 508](#). After all, the hints already point out that when the function discovers the correct split point, it needs both parts of the list, and one part is obviously lost due to recursion.

Once you have solved this exercise, equip the `main` function of [A Graphical Editor, Revisited](#) with a clause for mouse clicks. As you experiment with moving the cursor via mouse clicks, you will notice that it does not exactly behave like applications that you use on your other devices—even though `split` passes all its tests.

Graphical programs, like editors, call for experimentation to come up with the best “look and feel” experiences. In this case, your editor is too simplistic with its placement of the cursor. After the applications on your computer determine the split point, they also determine which letter division is closer to the x-coordinate and place the cursor there.

**Exercise 510.** Many operating systems come with the `fmt` program, which can rearrange the words in a file so that all lines in the resulting file have a maximal width. As a widely used program, `fmt` supports a range of related functions. This exercise focuses on its core functionality.

Design the program `fmt`. It consumes a natural number `w`, the name of an input file `in-f`, and the name of an output file `out-f`—in the same sense as `read-file` from the `2htdp/batch-io` library. Its purpose is to read all the words from the `in-f`, to arrange these words in the given order into lines of maximal width `w`, and to write these lines to `out-f`.

---

## 33 More Uses of Accumulation

This chapter presents three more uses of accumulators. The first section concerns the use of accumulators in conjunction with tree-processing functions. It uses the compilation of ISL+ as an illustrative example. The second section explains why we occasionally want accumulators inside of data representations and how to go about placing them there. The final section resumes the discussion of rendering fractals.

---

### 33.1 Accumulators and Trees

When you ask DrRacket to run an ISL+ program, it translates the program to commands for your specific computer. This process is called *compilation*, and the part of DrRacket that performs the task is called a *compiler*. Before the compiler translates the ISL+ program, it checks that every variable is declared via a `define`, a `define-struct`, or a `lambda`.

Stop! Enter `x`, `(lambda (y) x)`, and `(x 5)` as complete ISL+ programs into DrRacket and ask it to run each. What do you expect to see?

Let's phrase this idea as a sample problem:

**Sample Problem** You have been hired to re-create a part of the ISL+ compiler. Specifically, your task deals with the following language fragment, specified in the so-called grammar notation that many programming language manuals use:

We use the Greek letter  $\lambda$  instead of `lambda` to signal that this exercise deals with ISL+ as an object of study, not just a programming language.

```
expression = variable
| (λ (variable) expression)
| (expression expression)
```

Remember from intermezzo 1 that you can read the grammar aloud replacing `=` with “is one of” and `|` with “or.”

Recall that  $\lambda$  expressions are functions without names. They bind their parameter in their body. Conversely, a variable occurrence is declared by a surrounding  $\lambda$  that specifies the same name as a parameter. You may wish to revisit [Intermezzo 3: Scope and Abstraction](#) because it deals with the same issue from the perspective of a programmer. Look for the terms “binding occurrence,” “bound occurrence,” and “free.”

Develop a data representation for the above language fragment; use symbols to represent variables. Then design a function that replaces all undeclared variables with `'*undeclared`.

This problem is representative of many steps in the translation process and, at the same time, is a great case study for accumulator-style functions.

Before we dive into the problem, let's look at some examples in this mini-language, recalling what we know about `lambda`:

- `( $\lambda$  (x) x)` is the function that returns whatever it is given, also known as the identity function;
- `( $\lambda$  (x) y)` looks like a function that returns `y` whenever it is given an argument, except that `y` isn't declared;
- `( $\lambda$  (y) ( $\lambda$  (x) y))` is a function that, when given some value `v`, produces a function that always returns `v`;
- `(( $\lambda$  (x) x) ( $\lambda$  (x) x))` applies the identity function to itself;

- $((\lambda(x)(x x))(\lambda(x)(x x)))$  is a short infinite loop; and
- $((((\lambda(y)(\lambda(x)y))(\lambda(z)z))(\lambda(w)w))$  is a complex expression that is best run in ISL+ to find out whether it terminates.

Indeed, you can run all of the above ISL+ expressions in DrRacket to confirm what is written about them.

**Exercise 511.** Explain the scope of each binding occurrence in the above examples. Draw arrows from the bound to the binding occurrences.

Developing a data representation for the language is easy, especially because its description uses a grammar notation. Here is one possibility:

```
; A Lam is one of:
; - a Symbol
; - (list 'λ (list Symbol) Lam)
; - (list Lam Lam)
```

Because of [quote](#), this data representation makes it easy to create data representations for expressions in our subset of ISL+:

```
(define ex1 '(λ (x) x))
(define ex2 '(λ (x) y))
(define ex3 '(λ (y) (λ (x) y)))
(define ex4 '((λ (x) (x x)) (λ (x) (x x))))
```

These four data examples are representations of some of the above expressions. Stop! Create data representations for the remaining examples.

**Exercise 512.** Define `is-var?`, `is-λ?`, and `is-app?`, that is, predicates that distinguish variables from  $\lambda$  expressions and applications.

Also define

- $\lambda$ -para, which extracts the parameter from a  $\lambda$  expression;
- $\lambda$ -body, which extracts the body from a  $\lambda$  expression;
- app-fun, which extracts the function from an application; and
- app-arg, which extracts the argument from an application.

With these predicates and selectors, you basically can act as if you had defined a structure-oriented data representation.

Design `declareds`, which produces the list of all symbols used as  $\lambda$  parameters in a  $\lambda$  term. Don't worry about duplicate symbols.

**Exercise 513.** Develop a data representation for the same subset of ISL+ that uses structures instead of lists. Also provide data representations for `ex1`, `ex2`, and `ex3` following your data definition.

We follow the structural design recipe, and here is the product of steps two and three:

```
; Lam -> Lam
```

```

; replaces all symbols s in le with '*undeclared
; if they do not occur within the body of a λ
; expression whose parameter is s

(check-expect (undeclareds ex1) ex1)
(check-expect (undeclareds ex2) '(λ (x) *undeclared))
(check-expect (undeclareds ex3) ex3)
(check-expect (undeclareds ex4) ex4)

(define (undeclareds le0)
 le0)

```

Note how we expect undeclareds to process ex4 even though the expression loops forever when run; compilers don't run programs, they read them and create others.

A close look at the purpose statement directly suggests that the function needs an accumulator. This becomes even clearer when we inspect the template for undeclareds:

```

(define (undeclareds le)
 (cond
 [(is-var? le) ...]
 [(is-λ? le) (... (undeclareds (λ-body le)) ...)]
 [(is-app? le)
 (... (undeclareds (app-fun le))
 ... (undeclareds (app-arg le)) ...)]))

```

When undeclareds recurs on the body of (the representation of) a  $\lambda$  expression, it forgets ( $\lambda$ -para le), the declared variable.

So, let's start with an accumulator-style template:

```

(define (undeclareds le0)
 (local
 (; Lam ??? -> Lam
 ; accumulator a represents ...
 (define (undeclareds/a le a)
 (cond
 [(is-var? le) ...]
 [(is-λ? le)
 (... (undeclareds/a (λ-body le)
 ... a ...) ...)]
 [(is-app? le)
 (... (undeclareds/a (app-fun le)
 ... a ...)
 ... (undeclareds/a (app-arg le)
 ... a ...) ...)]))
 (undeclareds/a le0 ...)))

```

In this context, we can now formulate an accumulator invariant:

a represents the list of  $\lambda$  parameters encountered on the path from the top of le0 to the top of le.

For example, if `le0` is

```
'(((λ (y) (λ (x) y)) (λ (z) z)) (λ (w) w))
```

and `le` is the highlighted subtree, then `a` contains `y`. The left side of [figure 186](#) presents a graphical illustration of the same example. It shows a `Lam` expression as an upside-down tree; that is, the root is at the top. A `@` node represents an application with two descendants; the other nodes are self-explanatory. In this tree diagram, the bold path leads from `le0` to `le` through a single variable declaration.

Similarly, if we pick a different subtree of the same piece of data,

```
'(((λ (y) (λ (x) y)) (λ (z) z)) (λ (w) w))
```

we get an accumulator that contains both '`y`' and '`x`'. The right side of [figure 186](#) makes this point again. Here the bold path leads through two '`λ`' nodes to the boxed subtree, and the accumulator is the list of declared variables along the bold path.

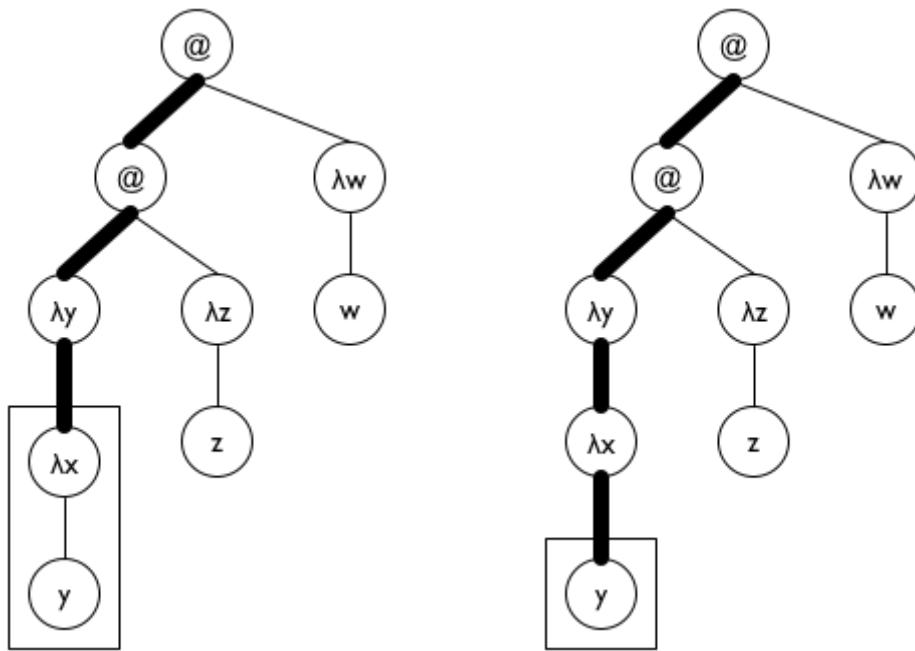


Figure 186: `Lam` terms as trees

Now that we have settled on the data representation of the accumulator and its invariant, we can resolve the remaining design questions:

- We pick an initial accumulator value of '`()`'.
- We use `cons` to add `(λ-para le)` to `a`.
- We exploit the accumulator for the clause where `undeclareds/a` deals with a variable. Specifically, the function uses the accumulator to check whether the variable is in the scope of a declaration.

[Figure 187](#) shows how to translate these ideas into a complete function definition. Note the name `declareds` for the accumulator; it brings across the key idea behind the accumulator invariant, helping the programmer understand the definition. The base case uses `member?` from `ISL+` to determine whether the variable `le` is in `declareds` and, if not, replaces it with

'\*undeclared. The second `cond` clause uses a `local` to introduce the extended accumulator `newd`. Because `para` is also used to rebuild the expression, it has its own local definition. Finally, the last clause concerns function applications, which do not declare variables and do not use any directly. As a result, it is by far the simplest of the three clauses.

```
; Lam -> Lam
(define (undeclareds le0)
 (local (; Lam [List-of Symbol] -> Lam
 ; accumulator declareds is a list of all λ
 ; parameters on the path from le0 to le
 (define (undeclareds/a le declareds)
 (cond
 [(is-var? le)
 (if (member? le declareds) le '*undeclared)]
 [(is-λ? le)
 (local ((define para (λ-para le))
 (define body (λ-body le))
 (define newd (cons para declareds)))
 (list 'λ (list para
 (undeclareds/a body newd))))]
 [(is-app? le)
 (local ((define fun (app-fun le))
 (define arg (app-arg le)))
 (list (undeclareds/a fun declareds)
 (undeclareds/a arg declareds))))])
 (undeclareds/a le0 '())))

```

Figure 187: Finding undeclared variables

**Exercise 514.** Make up an ISL+ expression in which `x` occurs both free and bound. Formulate it as an element of `Lam`. Does `undeclareds` work properly on your expression?

**Exercise 515.** Consider the following expression:

```
(λ (*undeclared) ((λ (x) (x *undeclared)) y))
```

Yes, it uses `*undeclared` as a variable. Represent it in `Lam` and check what `undeclareds` produces for this expression.

Modify `undeclareds` so that it replaces a free occurrence of '`x`' with

```
(list '*undeclared 'x)
```

and a bound one '`y`' with

```
(list '*declared 'y)
```

Doing so unambiguously identifies problem spots, which a program development environment such as DrRacket can use to highlight errors.

**Note** The trick of replacing a variable occurrence with the representation of an application feels awkward. If you dislike it, consider synthesizing the symbols '`*undeclared:x`' and '`declared:y`' instead.

**Exercise 516.** Redesign the `undeclare`s function for the structure-based data representation from [exercise 513](#).

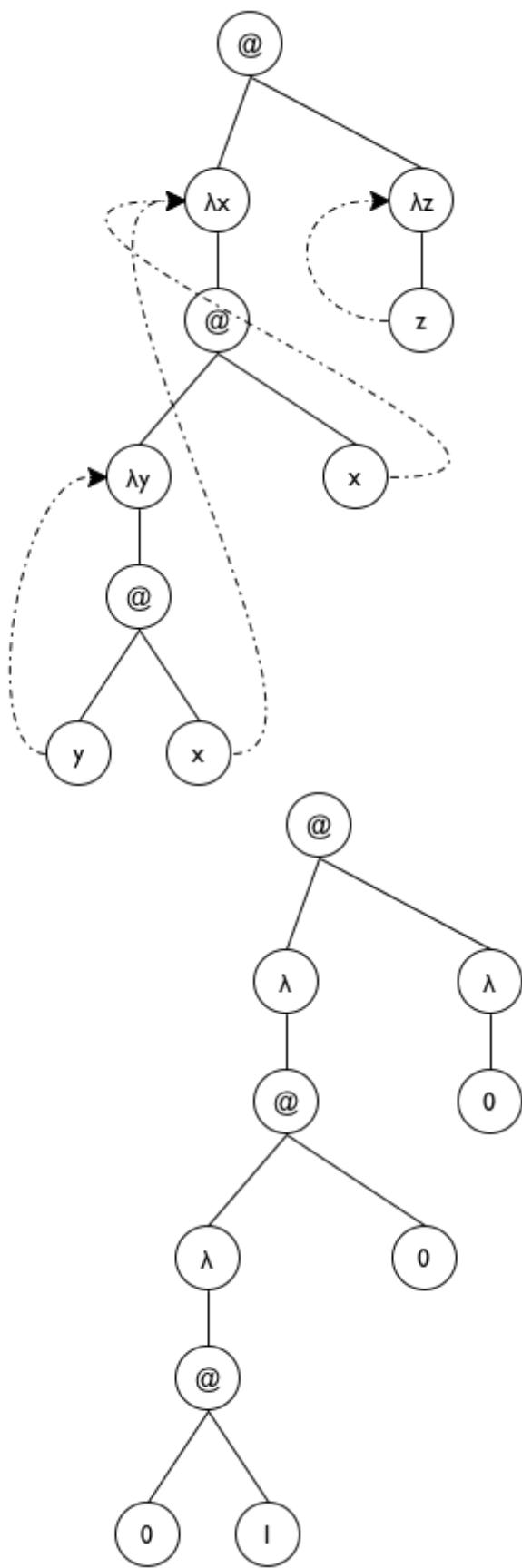


Figure 188: Static distances

**Exercise 517.** Design `static-distance`. The function replaces all occurrences of variables with a natural number that represents how far away the declaring  $\lambda$  is. [Figure 188](#) illustrates the idea for the term

```
'((λ (x) ((λ (y) (y x)) x)) (λ (z) z))
```

in graphical form. It includes dotted arrows that point from variable occurrences to the corresponding variable declarations. On the right, the figure shows a tree of the same shape, though without the arrows. The ' $\lambda$ ' nodes come without names, and variable occurrences have been replaced by natural numbers that specify which ' $\lambda$ ' declares the variable. Each natural number  $n$  says that the binding occurrence is  $n$  steps upward—toward the root of the [Lam](#) tree. A value of  $0$  denotes the first ' $\lambda$ ' on the path to the root,  $1$  the second one, and so on.

**Hint** The undeclareds accumulator of `undeclareds/a` is a list of all parameters on path from `le` to `le0` **in reverse order**—the last one seen is first on the list.

## 33.2 Data Representations with Accumulators

The end of [Intermezzo 5: The Cost of Computation](#) explains that \*SL measures the size of containers, say lists, by traversing them and hints that other programming languages use a different, less expensive way, to compute sizes. In this section, we show how to implement this idea with the **addition of an accumulator to data representations**.

See [Finite State Machines](#) for an early example of this idea.

Consider the ubiquitous lists in \*SL. All lists are constructed from `cons` and '`()`'; operations such as `quote` and `list`, for example, are merely abbreviations for these two. As [What Is '`\(\)`'](#), [What Is `cons`](#) shows, it is also possible to mimic lists in BSL with suitable structure type and function definitions.

```
(define-struct pair [left right])
; ConsOrEmpty is one of:
; - '()
; - (make-pair Any ConsOrEmpty)

; Any ConsOrEmpty -> ConsOrEmpty
(define (our-cons a-value a-list)
 (cond
 [(empty? a-list) (make-pair a-value a-list)]
 [(our-cons? a-list) (make-pair a-value a-list)]
 [else (error "our-cons: ...")])))

; ConsOrEmpty -> Any
; extracts the left part of the given pair
(define (our-first mimicked-list)
 (if (empty? mimicked-list)
 (error "our-first: ...")
 (pair-left mimicked-list)))
```

Figure 189: An implementation of lists in BSL

Figure 189 recalls the basic idea. Stop! Can you define `our-rest` now?

The key insight is that we can add a third field to the structure type definition of `pair`:

```
(define-struct cpair [count left right])
; A [MyList X] is one of:
; - '()
; - (make-cpair (tech "N") X [MyList X])
; accumulator the count field is the number of cpairs
```

As the accumulator statement says, the extra field is used to keep track of the number of `cpair` instances used to create the list. That is, it remembers a fact about the construction of the list. We call this kind of structure field a *data accumulator*.

Adding a field to the major list constructor does not come for free. To begin with, it requires a change to the checked version of the constructor, the one that is actually available to programs:

```
; data definitions, via a constructor-function
(define (our-cons f r)
 (cond
 [(empty? r) (make-cpair 1 f r)]
 [(cpair? r) (make-cpair (+ (cpair-count r) 1) f r)]
 [else (error "our-cons: ...")]))
```

If the extended list is '`()`', `count` is populated with `1`; otherwise, the function computes the length from the given `cpair`.

Now the function definition for `our-length` is obvious:

```
; Any -> N
; how many items does l contain
(define (our-length l)
 (cond
 [(empty? l) 0]
 [(cpair? l) (cpair-count l)]
 [else (error "my-length: ...")]))
```

The function consumes any kind of value. For '`()`' and instances of `cpair`, it produces natural numbers; otherwise it signals an error.

The second problem with the addition of a `count` field concerns performance. Indeed, there are two concerns. On the one hand, every single list construction comes with an extra field now, meaning a 33% increase in memory consumption. On the other hand, the addition of the field decreases how fast `our-cons` constructs a list. In addition to the check that the extended list is either '`()`' or an instance of a `cpair`, the constructor now computes the size of the list. Although this computation consumes a constant amount of time, it is imposed on every single use of `our-cons`—and just think how many times this book uses `cons` and does not ever compute how long the resulting list is!

**Exercise 518.** Argue that `our-cons` takes a constant amount of time to compute its result, regardless of the size of its input.

**Exercise 519.** Is it acceptable to impose the extra cost on `cons` for all programs to turn `length` into a constant-time function?

While the addition of a `count` field to lists is questionable, sometimes data accumulators play a crucial role in finding a solution. The next example is about adding so-called *artificial intelligence* to a board-game-playing program, and its data accumulator is an absolute necessity.

As you play board games or solve puzzles, you tend to think about your possible moves at every stage. As you get better, you may even imagine the possibilities after this first step. The result is a so-called *game tree*, which is a (part of the) tree of all possible moves that the rules allow. Let's start with a problem:

**Sample Problem** Your manager tells you the following story.

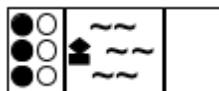
"Once upon a time, three cannibals were guiding three missionaries through a jungle. They were on their way to the nearest mission station. After some time, they arrived at a wide river, filled with deadly snakes and fish. There was no way to cross the river without a boat. Fortunately, they found a rowboat with two oars after a short search. Unfortunately, the boat was too small to carry all of them. It could barely carry two people at a time. Worse, because of the river's width someone had to row the boat back."

"Since the missionaries could not trust the cannibals, they had to figure out a plan to get all six of them safely across the river. The problem was that these cannibals would kill and eat missionaries as soon as there were more cannibals than missionaries in some place. Our missionaries had to devise a plan that guaranteed that there were never any missionaries in the minority on either side of the river. The cannibals, however, could be trusted to cooperate otherwise. Specifically, they would not abandon any potential food, just as the missionaries would not abandon any potential converts."

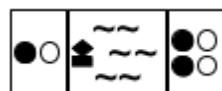
While your manager doesn't assign any specific design task, he wants to explore whether the company can design (and sell) programs that solve such puzzles.

While puzzles aren't board games, the program illustrates the idea of game trees in the most straightforward manner possible.

In principle, it is quite straightforward to solve such puzzles by hand. Here is the rough idea. Pick a graphical representation of the problem states. Ours consists of a three-part box: the left one represents the missionaries and the cannibals; the middle combines the river and the boat; and the third part is the right-hand side of the river. Take a look at the following representation of the initial state:



Black circles denote missionaries, white circles cannibals. All of them are on the left-hand river bank. The boat is also on the left side. Nobody is on the right. Here are two more states:



The first one is the final state, where all people and the boat are on the right bank of the river. The second one depicts some intermediate state where two people are on the left with the boat and four people are on the right.

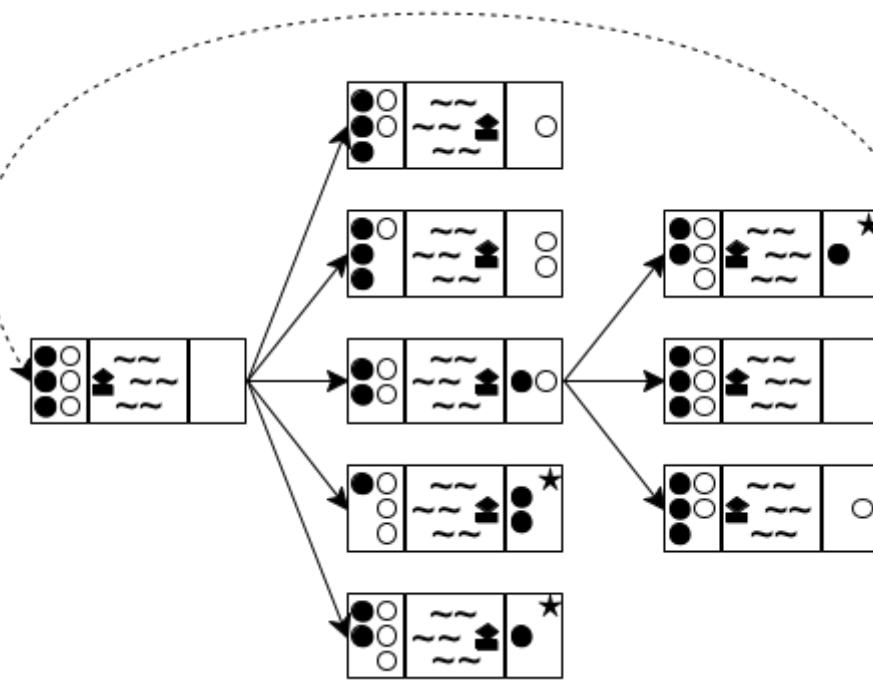


Figure 190: Creating a game tree

Now that you have a way to write down the state of the puzzle, you can think about the possibilities at each stage. Doing so yields a tree of possible moves. [Figure 190](#) sketches the first two and a half layers in such a tree. The left-most state is the initial one. Because the boat can transport at most two people and must be rowed by at least one, you have five possibilities to explore: one cannibal rows across; two cannibals row across; one missionary and one cannibal go; one missionary crosses; or two missionaries do. These possibilities are represented with five arrows going from the initial state to five intermediate states.

For each of these five intermediate states, you can play the same game again. In [figure 190](#) you see how the game continues for the middle (third) one of the new states. Because there are only two people on the right river bank, you see three possibilities: a cannibal goes back, a missionary goes back, or both do. Hence three arrows connect the middle state to the three states on the right side of the tree. If you keep drawing this tree of possibilities in a systematic manner, you eventually discover the final state.

A second look at [figure 190](#) reveals two problems with this naive approach to generating the tree of possibilities. The first one is the dashed arrow that connects the middle state on the right to the initial state. It indicates that rowing back the two people from the right to the left gets the puzzle back to its initial state, meaning you're starting over, which is obviously undesirable. The second problem concerns those states with a star in the top-right corner. In both cases, there are more white-circle cannibals than black-circle missionaries on the left river bank, meaning the cannibals would eat the missionaries. Again, the goal is to avoid such states, making these moves undesirable.

One way to turn this puzzle into a program is to design a function that determines whether some final state—here **the** final state—is reachable from some given state. Here is an

appropriate function definition:

```
; PuzzleState -> PuzzleState
; is the final state reachable from state0
; generative creates a tree of possible boat rides
; termination ???

(check-expect (solve initial-puzzle) final-puzzle)

(define (solve state0)
 (local (; [List-of PuzzleState] -> PuzzleState
 ; generative generates the successors of los
 (define (solve* los)
 (cond
 [(ormap final? los)
 (first (filter final? los))]
 [else
 (solve* (create-next-states los))]))
 (solve* (list state0))))
```

The auxiliary function uses generative recursion, generating all new possibilities given a list of possibilities. If one of the given possibilities is a final state, the function returns it.

Clearly, `solve` is quite generic. As long as you define a collection of *PuzzleStates*, a function for recognizing final states, and a function for creating all “successor” states, `solve` can work on your puzzle.

**Exercise 520.** The `solve*` function generates all states reachable with  $n$  boat trips before it looks at states that require  $n + 1$  boat trips, even if some of those boat trips return to previously encountered states. Because of this systematic way of traversing the tree, `solve*` cannot go into an infinite loop. Why? **Terminology** This way of searching a tree or a graph is dubbed *breadth-first search*.

**Exercise 521.** Develop a representation for the states of the missionary-and-cannibal puzzle. Like the graphical representation, a data representation must record the number of missionaries and cannibals on each side of the river plus the location of the boat.

The description of `PuzzleState` calls for a new structure type. Represent the above initial, intermediate, and final states in your representation.

Design the function `final?`, which detects whether in a given state all people are on the right river bank.

Design the function `render-mc`, which maps a state of the missionary-and-cannibal puzzle to an image.

The problem is that returning the final state says nothing about how the player can get from the initial state to the final one. In other words, `create-next-states` forgets how it gets to the returned states from the given ones. And this situation clearly calls for an accumulator, but at the same time, the accumulated knowledge is best associated with every individual `PuzzleState`, not `solve*` or any other function.

**Exercise 522.** Modify the representation from [exercise 521](#) so that a state records the sequence of states traversed to get there. Use a list of states.

Articulate and write down an accumulator statement with the data definition that explains the additional field.

Modify `final?` or `render-mc` for this representation as needed.

**Exercise 523.** Design the `create-next-states` function. It consumes lists of missionary-and-cannibal states and generates the list of all those states that a boat ride can reach.

Ignore the accumulator in the first draft of `create-next-states`, but make sure that the function does not generate states where the cannibals can eat the missionaries.

For the second design, update the accumulator field in the state structures and use it to rule out states that have been encountered on the way to the current state.

**Exercise 524.** Exploit the accumulator-oriented data representation to modify `solve`. The revised function produces the list of states that lead from the initial [PuzzleState](#) to the final one.

Also consider creating a movie from this list, using `render-mc` to generate the images. Use `run-movie` to display the movie.

---

### 33.3 Accumulators as Results

Take another look at [figure 157](#). It displays a Sierpinski triangle and a suggestion how to create it. Specifically, the images on the right explain one version of the generative idea behind the process:

The given problem is a triangle. When the triangle is too small to be subdivided any further, the algorithm does nothing; otherwise, it finds the midpoints of its three sides and deals with the three outer triangles recursively.

In contrast, [Fractals, a First Taste](#) shows how to compose Sierpinski triangles algebraically, a process that does not correspond to this description.

Most programmers expect “draw” to mean the action of adding a triangle to some canvas. The `scene+line` function from the `2htdp/image` library makes this idea concrete. The function consumes an image `s` and the coordinates of two points and adds a line through these two points to `s`. It is easy to generalize from `scene+line` to `add-triangle` and from there to `add-sierpinski`:

**Sample Problem** Design the `add-sierpinski` function. It consumes an image and three `Posns` describing a triangle. It uses the latter to add a Sierpinski triangle to this image.

Note how this problem implicitly refers to the above process description of how to draw a Sierpinski triangle. In other words, we are confronted with a classical generative-recursive problem, and we can start with the classic template of generative recursion and the four central design questions:

- The problem is trivial if the triangle is too small to be subdivided.

- In the trivial case, the function returns the given image.
- Otherwise the midpoints of the sides of the given triangle are determined to add another triangle. Each “outer” triangle is then processed recursively.
- Each of these recursive steps produces an image. The remaining question is how to combine these images.

```

; Image Posn Posn Posn -> Image
; generative adds the triangle (a, b, c) to s,
; subdivides it into three triangles by taking the
; midpoints of its sides; stop if (a, b, c) is too small
(define (add-sierpinski scene0 a b c)
 (cond
 [(too-small? a b c) scene0]
 [else
 (local
 ((define scene1 (add-triangle scene0 a b c))
 (define mid-a-b (mid-point a b))
 (define mid-b-c (mid-point b c))
 (define mid-c-a (mid-point c a)))
 (define scene2
 (add-sierpinski scene0 a mid-a-b mid-c-a))
 (define scene3
 (add-sierpinski scene0 b mid-b-c mid-a-b))
 (define scene4
 (add-sierpinski scene0 c mid-c-a mid-b-c)))
 ; -IN-
 (... scene1 ... scene2 ... scene3 ...)))]))

```

Figure 191: Accumulators as results of generative recursions, a skeleton

Figure 191 shows the result of translating these answers into a skeletal definition. Since each midpoint is used twice, the skeleton uses `local` to formulate the generative step in ISL+. The `local` expression introduces the three new midpoints plus three recursive applications of `add-sierpinski`. The dots in its body suggest a combination of the scenes.

**Exercise 525.** Tackle the wish list that the skeleton implies:

```

; Image Posn Posn Posn -> Image
; adds the black triangle a, b, c to scene
(define (add-triangle scene a b c) scene)

; Posn Posn Posn -> Boolean
; is the triangle a, b, c too small to be divided
(define (too-small? a b c)
 #false)

; Posn Posn -> Posn
; determines the midpoint between a and b
(define (mid-point a b)
 a)

```

Design the three functions.

**Domain Knowledge** (1) For the `too-small?` function it suffices to measure the distance between two points and to check whether it is below some chosen threshold, say, `10`. The distance between  $(x_0, y_0)$  and  $(x_1, y_1)$  is

$$\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

that is, the distance of  $(x_0 - x_1, y_0 - y_1)$  to the origin.

The midpoint between points  $(x_0, y_0)$  and  $(x_1, y_1)$  has as coordinates the midpoints between the respective x and y coordinates:

$$\left( \frac{1}{2} \cdot (x_0 + x_1), \frac{1}{2} \cdot (y_0 + y_1) \right).$$

Now that we have all the auxiliary functions, it is time to return to the problem of combining the three images that are created by the recursive calls. One obvious guess is to use the `overlay` or `underlay` function, but an evaluation in the interactions area of DrRacket shows that the functions hide the underlying triangles.

Specifically, imagine that the three recursive calls produce the following empty scenes, enriched with a single triangle in appropriate locations:

```
> scene1

> scene2

> scene3

```

A combination should look like this figure:



But, combining these shapes with `overlay` or `underlay` does not yield this desired shape:

```
> (overlay scene1 scene2 scene3)

> (underlay scene1 scene2 scene3)

```

Indeed, the image library of ISL+ does not support a function that combines these scenes in an appropriate manner.

Let's take a second look at these interactions. If `scene1` is the result of adding the upper triangle to the given scene and `scene2` is the result of adding a triangle on the lower left, perhaps the second recursive call should add triangles to the result of the first call. Doing so would yield



and handing over this scene to the third recursive call produces exactly what is wanted:



```
; Image Posn Posn Posn -> Image
; generative adds the triangle (a, b, c) to s,
; subdivides it into three triangles by taking the
; midpoints of its sides; stop if (a, b, c) is too small
; accumulator the function accumulates the triangles scene0
(define (add-sierpinski scene0 a b c)
 (cond
 [(too-small? a b c) scene0]
 [else
 (local
 ((define scene1 (add-triangle scene0 a b c))
 (define mid-a-b (mid-point a b))
 (define mid-b-c (mid-point b c))
 (define mid-c-a (mid-point c a)))
 (define scene2
 (add-sierpinski scene1 a mid-a-b mid-c-a))
 (define scene3
 (add-sierpinski scene2 b mid-b-c mid-a-b)))
 ; -IN-
 (add-sierpinski scene3 c mid-c-a mid-b-c))))
```

Figure 192: Accumulators as results of generative recursion, the function

Figure 192 shows the reformulation based on this insight. The three highlights pinpoint the key design idea. All concern the case when the triangle is sufficiently large and it is added to the given scene. Once its sides are subdivided, the first outer triangle is recursively processed using scene1, the result of adding the given triangle. Similarly, the result of this first recursion, dubbed scene2, is used for the second recursion, which is about processing the second triangle. Finally, scene3 flows into the third recursive call. In sum, the novelty is that the accumulator is simultaneously an argument, a tool for collecting knowledge, and the result of the function.

To explore add-sierpinski it is best to start from an equilateral triangle and an image that leaves a sufficiently large border. Here are definitions that meet these two criteria:

```
(define MT (empty-scene 400 400))
(define A (make-posn 200 50))
(define B (make-posn 27 350))
(define C (make-posn 373 350))

(add-sierpinski MT A B C)
```

Check what kind of Sierpinski fractal this code fragment delivers. Experiment with the definitions from [exercise 525](#) to create sparser and denser Sierpinski triangles than the first one.

**Exercise 526.** To compute the endpoints of an equilateral Sierpinski triangle, draw a circle and pick three points on the circle that are 120 degrees apart, for example, 120, 240, and 360.

Design the function `circle-pt`:

```
(define CENTER (make-posn 200 200))
(define RADIUS 200) ; the radius in pixels

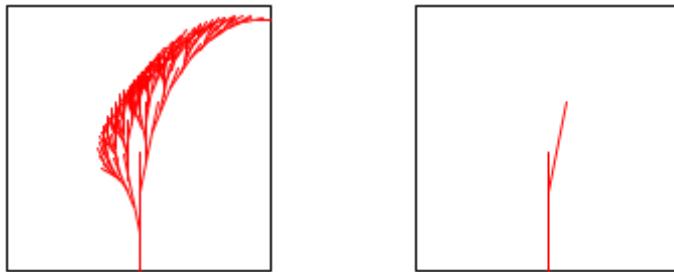
; Number -> Posn
; determines the point on the circle with CENTER
; and RADIUS whose angle is

; examples
; what are the x and y coordinates of the desired
; point, when given: 120/360, 240/360, 360/360

(define (circle-pt factor)
 (make-posn 0 0))
```

**Domain Knowledge** This design problem calls on knowledge from mathematics. One way to view the problem is as a conversion of a complex number from the polar-coordinate representation to the `Posn` representation. Read up on `make-polar`, `real-part`, and `imag-part` in ISL+. Another way is to use trigonometry, `sin` and `cos`, to determine the coordinates. If you choose this route, recall that these trigonometry functions compute the sine and cosine in terms of radians, not degrees. Also keep in mind that on-screen positions grow downward, not upward.

**Exercise 527.** Take a look at the following two images:



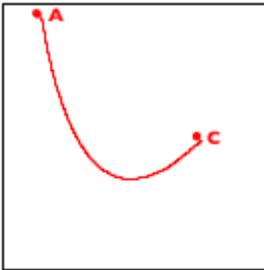
They demonstrate how to generate a fractal Savannah tree in the same way that [figure 156](#) shows how to draw a Sierpinski triangle. The image on the left shows what a fractal Savannah tree looks like. The right one explains the generative construction step.

Design the function `add-savannah`. The function consumes an image and four numbers: (1) the x-coordinate of a line's base point, (2) the y-coordinate of a line's base point, (3) the length of the line, and (4) the angle of the line. It adds a fractal Savannah tree to the given image.

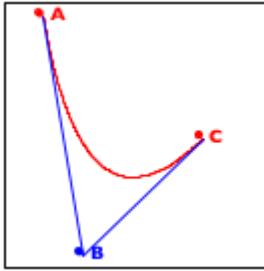
Unless the line is too short, the function adds the specified line to the image. It then divides the line into three segments. It recursively uses the two intermediate points as the new starting points for two lines. The lengths and the angles of the two branches change in a fixed manner, but independently of each other. Use constants to define these changes and work with them until you like your tree well enough.

**Hint** Experiment with shortening each left branch by at least one third and rotating it left by at least `0.15` degrees. For each right branch, shorten it by at least 20% and rotate it by `0.2` degrees in the opposite direction.

**Exercise 528.** Graphics programmers often need to connect two points with a smooth curve where “smooth” is relative to some perspective. Here are two sketches:

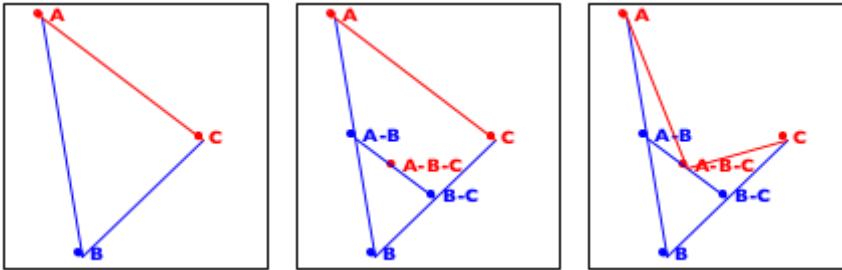


Géraldine Morin suggested this exercise.



The left one shows a smooth curve, connecting points  $A$  and  $C$ ; the right one supplies the perspective point,  $B$ , and the angle of an observer.

One method for drawing such curves is due to Bézier. It is a prime example of generative recursion, and the following sequence explains the *eureka!* behind the algorithm:



Consider the image on the left. It reminds you that the three given points determine a triangle and that the connection from  $A$  to  $C$  is the focal point of the algorithm. The goal is to pull the line from  $A$  to  $C$  toward  $B$  so that it turns into a smooth curve.

Now turn to the image in the middle. It explains the essential idea of the generative step. The algorithm determines the midpoint on the two observer lines,  $A-B$  and  $B-C$ , as well as their midpoint,  $A-B-C$ .

Finally, the right-most image shows how these three new points generate two distinct recursive calls: one deals with the new triangle on the left and the other one with the triangle on the right. More precisely,  $A-B$  and  $B-C$  become the new observer points and the lines from  $A$  to  $A-B-C$  and from  $C$  to  $A-B-C$  become the foci of the two recursive calls.

When the triangle is small enough, we have a trivially solvable case. The algorithm just draws the triangle, and it appears as a point on the given image. As you implement this algorithm, you need to experiment with the notion of “small enough” to make the curve look smooth.

This last part is about designing with accumulators, a mechanism for collecting knowledge during a data structure traversal. Adding an accumulator can fix performance flaws and eliminate termination problems. Your take away from this part are two and a half design lessons:

1. The first step is to recognize the need for introducing an accumulator. Traversals “forget” pieces of the argument when they step from one piece to the next. If you discover that such knowledge could simplify the function’s design, consider introducing an accumulator. The first step is to switch to the **accumulator template**.
2. The key step is to formulate an accumulator statement. The latter must express **what knowledge** the accumulator gathers **as what kind of data**. In most cases, the accumulator statement describes the difference between the original argument and the current one.
3. The third step, a minor one, is to deduce from the accumulator statement (a) what the initial accumulator value is, (b) how to maintain it during traversal steps, and (c) how to exploit its knowledge.

The idea of accumulating knowledge is ubiquitous, and it appears in many different forms and shapes. It is widely used in so-called functional languages like ISL+. Programmers using imperative languages encounter accumulators in a different way, mostly via assignment statements in primitive looping constructs because the latter cannot return values. Designing such imperative accumulator programs proceeds just like the design of accumulator functions here, but the details are beyond the scope of this first book on systematic program design.

---

## Epilogue: Moving On

You have reached the end of this introduction to computing and programming, or program design, as we say here. While there is more to learn about both subjects, this is a good point to stop, summarize, and 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; then came fractions. Eventually, you found out about variables and functions, which your teachers called **algebra**. Variables represented numbers, and functions related numbers to numbers.

Because you used numbers throughout this process, 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 step from information to data and back becomes central. Nowadays, programs process representations of music, videos, molecules, chemical compounds, business case studies, electrical diagrams, and blueprints. Fortunately, you don't need to encode all this information with numbers or, worse, just `0` and `1`; if you had to, life would be unimaginably tedious. Instead, computing generalizes arithmetic and algebra so that when you program, you can code—and your programs can compute—with strings, Booleans, characters, structures, lists, functions, and many more kinds of data.

Classes of data and their functions come with equational laws that explain their meaning, just like the laws for numbers and their functions. While these equational laws are as simple as “`( + 1 1)` evaluates to `2`” and “`(not #true)` equals `#false`,” you can use them to predict the behavior of entire programs. When you run a program, you actually just apply one of its many functions, an act that you can explain with the beta rule first mentioned in [Intermezzo 1: Beginning Student Language](#). Once the variables are replaced with values, the laws of data take over until you have either only a value or another function application. But yes, that's all there is to computing.

---

## Program Design

A typical software development project requires the collaboration of many programmers, and the result consists of thousands of functions. Over the life span of such a project, programmers come and go. Hence, the design structure of programs is really a means of communication among programmers across time. When you approach code that someone else wrote some time ago, the program ought to express its purpose and its relationships to other pieces—because that other person might not be around anymore.

In such a dynamic context, programmers must create programs in a disciplined manner if they wish to work reasonable numbers of hours or produce high-quality products. Following a systematic design method guarantees that the program organization is comprehensible. Others can then easily understand the pieces and the whole, and then fix bugs or add new pieces of functionality.

The design process of this book is one of these methods, and you ought to follow it whenever you create programs you might care about. You start with an analysis of the world of information and a description of the data that represents the information. Then you make a plan, a work list of functions needed. If this list is large, you refine the process in an iterative manner. You start with a subset of functions that quickly yields a product with which a client can interact. As you observe these interactions, you will quickly figure out which elements of your work list to tackle next.

Designing a program, or only 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. Make up, and work through, examples. 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 rerun these tests and reconfirm that the program still works for the basic examples.

Eventually your program will also fail. Other programmers may use it in an unanticipated manner. Real-world users may find differences between expected and actual behavior. Because you have designed the code in a systematic manner, you will know what to do. You will formulate a failing test case for your program's main function. From this one test, you will derive a test case for each function that the main function mentions. Those functions that pass their new tests do not contribute to the failure. One of the others does; on occasion, several might collude to create a bug. If the broken function composes others, resume the test creation; otherwise you have found the source of the problem. You will know that you have fixed the problem when the program as a whole passes all its tests.

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 design flaws and repetitions of designs. If you find any design patterns, form new abstractions or use existing abstractions to eliminate these patterns.

If you respect these guidelines, you will produce solid software with reasonable effort. It will work because you understand why and how it works. Others who must modify or enhance your software will understand it quickly because the code communicates its process and its purpose. Working through this book got you started. Now 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.

---

## Onward, Developers and Computer Scientists

Right now, you might be wondering what to study next. The answer is both more programming and more computing.

As a student of program design, your next task is to learn how the design process applies in the setting of a full-fledged programming language. Some of these languages are like the teaching languages, and the transition will be easy. Others require a different mind-set because they offer 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 process to the use and production of so-called frameworks (“stacks”) and components. Roughly speaking, frameworks abstract pieces of functionality—for example, graphical user interfaces, database connections, and web connectivity—that are common to many software systems. You need to learn to instantiate these abstractions, and your programs will compose these instances to create coherent systems. Similarly, learning to create new system components is also 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 the computational process. This book has focused on the laws that describe the process itself. In order to function as a real software engineer, you need to learn what the process costs, at both a theoretical level and a practical one. Studying the concept of big-O in some more depth is a first, small step in this direction; learning to measure and analyze a program’s performance is the real goal because you will need this skill as a developer on a regular basis. Above and beyond these basic ideas, you will also need knowledge about hardware, networking, layering of software, and specialized algorithms in various disciplines.

---

## Onward, Accountants, Journalists, Surgeons, and Everyone Else

Some of you wanted to see what computing and programming are all about. You now know that computing is merely a generalization of calculating, and you may sense how useful program design is to you. Even if you never develop programs again, you know what distinguishes a garage programmer from a serious software developer. When you interact with developers as a professional, you know that systematic design matters because it affects your quality of life and the bottom line of your business.

In reality, though, you are likely to “program” again, on a regular basis; you may just fail to see your activities in this light. Imagine a journalist for a moment. His story starts with the collection of information and data, laying it out, organizing it, and adding anecdotes. If you squint, you’ll see that this is only step one of the design process. Let’s turn to a family doctor who, after checking up on your symptoms, formulates a hypothesis of what might affect you. Do you see step two? Or, think of a lawyer who illustrates the point of an argument with a number of examples—an instance of step three. Finally, a civil engineer cross-checks the bridge as it is built to make sure it lives up to the blueprint and the underlying static calculations. Cross-checking is a form of testing—step six of the process; it compares actual measurements with expected values from the predictive calculations. Each of these professionals develops a system to work effectively and efficiently; and deep down, this system is likely to resemble the design process employed in this book.

Now, once you accept that many activities are a form of programming, you can transfer additional ideas from the design process to your own life. For example, if you recognize patterns, you may take the little additional time it takes to create an “abstraction”—a single point of control—to simplify your future work. So, regardless of whether you become an accountant or a doctor or something else, remember the design processes wherever you go and whatever you do.

**Exercise** Write a short essay on how the design process may help you with your chosen profession.