Submitted to the 2014 ACM SIGPLAN International Conference on Functional Programming (ICFP'14)

On Teaching *How to Design Programs*: Observations from a Newcomer

Norman Ramsey
Department of Computer Science, Tufts University
nr@cs.tufts.edu

March 2014

Abstract

This paper presents a personal, qualitative case study of a first course using *How to Design Programs* and its functional teaching languages. The paper reconceptualizes the book's six-step design process as an eight-step design process ending in a new "review and refactor" step. It recommends specific approaches to students' difficulties with function descriptions, function templates, data examples, and other parts of the process. It connects the design process to interactive "world programs." It recounts significant, informative missteps in course design and delivery. Finally, it identifies some unsolved teaching problems and some potential solutions.

1 Introduction

This paper is about teaching introductory programming using the method called *Program by Design*, which is explained in the book called *How to Design Programs* (Felleisen et al. 2001). The method uses functional-programming principles, and the book uses functional languages derived from Scheme. The method has proven effective in different educational contexts at many levels (Felleisen et al. 2004b, 2009; Bieniusa et al. 2008; Bloch 2010; Schanzer, Fisler, and Krishnamurthi 2013).

How to Design Programs argues eloquently that everyone should learn to program. And the book keeps its implied promise: my students really did

learn to design programs. But the idea that students learned is not enough; a teacher needs to know what "learning to program" means. What exactly did my students learn to do? How did they learn it? In the jargon of the educator, what were the learning outcomes? While I have come to love lambdas and round parentheses and `cond` expressions, these are not the kinds of learning outcomes that teachers need to know about in order to ensure students' success in a *second* course. And although some valuable information is available from Bieniusa et al. (2008), from Crestani and Sperber (2010), and from Sperber and Crestani (2012), teachers still need more. This paper provides some.

The contributions of this paper are

- To articulate a refined, extended version of the design method presented in *How to Design Programs*, and to develop a view of the method, from a newcomer's perspective, that can help a teacher deliver a class (Section 2)
- To identify, from direct observation, where students struggle with the method and what points can be emphasized to help them succeed (Section 3)
- To communicate what it's like for a functional programmer with no Scheme experience to work with the languages and tools (Section 4)
- To identify and learn from one beginner's mistakes (Section 5)
- To identify some open problems and sketch potential solutions (Section 6)

I have written the paper for people who wish to use functional programming to teach an introductory course. I assume experience with typeful functional programming at the level of Haskell, ML, or System F; I assume no experience with LISP, Scheme, or Racket. I address questions like those we ask graduating PhD students: what parts were hard, and when I do it again, what I will do differently.

I address these questions using the "humanities" approach to educational research (Burkhardt and Schoenfeld 2003), in which authors reflect on their experience. Burkhardt and Schoenfeld write that "the test of quality is critical appraisal concerning plausibility, internal consistency and fit to prevailing wisdom. The key product of this approach is critical commentary." My reflections and commentary are informed by empirical observations in the classroom, but the paper is purely reflective, with no controlled ex-

periments or quantitative measurements. Information about my teaching experience appears in Appendix A.

2 What is *Program by Design*?

If you teach a course in *Program by Design*, using *How to Design Programs*, you can expect these outcomes:

1. Your students will learn a step-by-step *design process*. The process is presented in six steps, but as explained below, I found it helpful to articulate eight steps.
2. Your students will be able to apply the design process to build *functions* that consume increasingly more sophisticated forms of data: strings, images, numbers and numeric intervals; enumerations; products; general sums (including sums of products); and lists, trees, or other sums of products whose definitions incorporate self-reference or mutual reference. Each form of data engenders a specialized instance of the design process called a *design recipe*.
3. With additional guidance, your students will be able to design interactive *programs* that are composed of many functions.

Your students can also pick up a few other techniques that don't fit neatly into the model of "process plus data equals recipe." Possibilities include using abstraction to eliminate duplicate or near-duplicate code; writing "generative" recursive functions; using higher-order functions on lists; using accumulating parameters; reasoning about costs; and programming with mutable state.

Your students can achieve these outcomes using either the complete, first edition of *How to Design Programs* or the incomplete second edition. (The choice is discussed in Appendix D.) With either edition, the key learning outcome is mastery of design recipes, and the distinctive aspect of the recipes is the design process.

2.1 Introduction to the (refined) design process

How to Design Programs presents the design process for functions in six steps:

1. Describe the data used by the function

2. Using a *signature*, *purpose statement*, and *header*, describe what the function does[1]
3. Show examples of what the function does
4. Write a template for the function
5. Fill in the template with code, making a complete function
6. Test the function

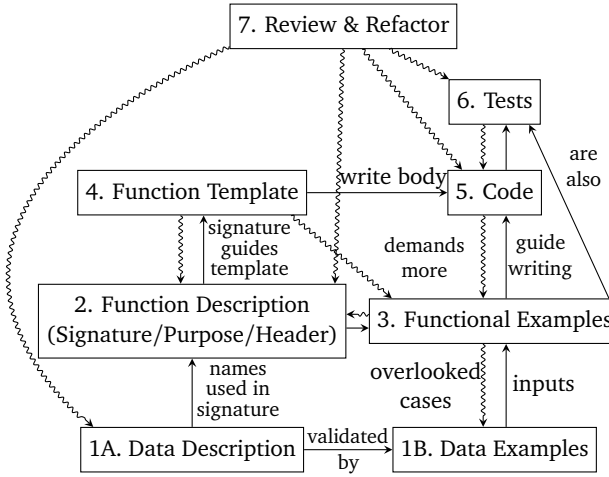
But this six-step process supports only some of the skills the book teaches. The other skills primarily involve eliminating repetition and establishing a single point of truth, e.g., reducing multiple function definitions to a single function definition by abstracting over additional parameters, or eliminating repetitive case analysis and recursion by using higher-order functions. At first I found these skills hard to motivate, but after teaching them I realized they could fit into a new, seventh step of the design process:

7. Review and refactor the code

To call this final step "new" is not really fair; ideas that bear on reviewing and refactoring appear everywhere in *How to Design Programs*. What I have done is to *articulate* this step, which had been implicit and hidden.

The steps are presented sequentially, but in practice, they are richly interrelated. Early steps support multiple later steps, and later steps can trigger revisions in earlier steps. To help students use the design process mindfully, I taught them about the relationships shown in the diagram below. Initial design flows through solid arrows; feedback that triggers revisions flows through squiggly arrows.

[1] In the first edition, the signature is called a "contract."



In the diagram, as in class, I treat data description and data examples as separate steps, making eight steps in all. Separating data examples makes them harder for students to forget, and it brings out ways in which the description and development of functions parallel the description and exemplification of data.[2]

Students of *Program by Design* learn all steps of the design process immediately, using simple atomic data. They then learn to specialize the process for products, sums, sums of products, self-referential and mutually referential data (a.k.a. recursive types), functions as data, and finally mutable data.

2.2 Relating *Program by Design* to functional programming

Program by Design is not just a paper-and-pencil design method; it is supported by the DrRacket programming environment and by the Racket teaching languages: Beginning Student Language, Intermediate Student Language, and Advanced Student Language. The software and languages are described in detail elsewhere (Findler et al. 2002; Felleisen et al.

[2]Sperber and Crestani (2012) also expand the design process into eight steps: they split function description into two steps (purpose statement and signature) and the function template into two steps ("skeleton" and template). They do not mention anything like "review and refactor."

2004a), but the languages are worth summarizing here: Beginning Student Language is a pure, eager, first-order, functional language that has global definitions of functions and variables, structure (record) definitions, LISP's multiway conditionals, and language constructs for expressing unit tests. Intermediate Student Language adds nested definitions and `lambda`, making functions higher-order and first-class. Advanced Student Language adds mutation and imperative I/O.

Is it functional programming? Well, the design method does not use equational reasoning or algebraic laws, ideas that some functional programmers deem essential (Bird and Wadler 1988). There are (at least at first) no higher-order functions, and the language is not lazy, which rules out the kinds of modularity that may make functional programming matter (Hughes 1989). And while testable equations are central, equational *properties* can be tested only if you can read documentation in German (Crestani and Sperber 2010).

On the other hand, data are immutable. As a result, specifications of functions are simple and equational, and unit tests are simply equations. The first two languages are pure, so their evaluation can be explained (and debugged) using DrRacket's algebraic stepper. Function composition is encouraged. Control flow is expressed exclusively through a combination of function calls and conditionals. In particular, there are no loops; there are only recursive functions and higher-order list functions. And although there is no static type checking, your students will nevertheless learn to write parametric type definitions and polymorphic functions.

If your goal is to teach functional programming, this summary may tell you if *Program by Design* will meet your needs. Our institutional need, as detailed in Appendix C, was to teach beginning students how to solve problems using the computer; for us, functional programming is a means, not an end.

2.3 Understanding and teaching the design method

This section highlights aspects of *Program by Design* to which, in my first time teaching the method, I had to pay extra attention.

Types guide code As noted by Felleisen et al. (2004a), the syllabus is driven by *data definitions*. Data definitions are informal; a data definition is a comment that introduces a type and gives it a name. The types are

familiar: there are (immutable) base types, and new types can be formed using products, sums, and arrows. As suggested by Brooks (1975) and Jackson (1975), the types of input data guide the shape of code. If a function f consumes a value of type τ, then f's body is typically designed around the elimination form(s) for values of type τ. In *How to Design Programs*, the body design is called a *template*. The template for a sum uses the elimination form for sums: the conditional. The template for a product uses the elimination forms for products: projection functions, which are called *selectors*. (Values projected from a product are then "combined" by applying a function.)

Teaching types and code Words like "sum type" and "product type" are too mathematical for our beginning students, many of whom think they can't do math. To keep students comfortable so they can learn, I use terminology from Stephen Bloch: a definition of a product type is a *definition by parts*, and a definition of a sum type is a *definition by choices*. To avoid getting into the distinction between defining a name and using a type-formation rule, I also abuse terminology and talk about types that are "defined by name."

How to Design Programs made it easy for me to teach students to use the elimination forms for sums and products. It was not so easy to teach students to use named types. The "elimination form" for a named type is a function call, but what are the right functions to call? It depends on where the name comes from and what manner of type it refers to.

- If a type name refers to a base type (*atomic data*), then the right calls are to library functions or to helper functions written by the student. The student should look for a function whose signature says it consumes a value of the named type.
- If a type name refers to the data definition in which it appears, that definition is self-referential, and the right call is a *naturally recursive* call to the function being defined.
- If a type name refers to a data definition in a group of mutually referential definitions, then the design recipe calls for parallel development of a group of similar functions, one per data definition. The right call is to the function within the group that consumes data of the type referred to.
- Finally, if the type name refers to a data definition written by the student (or the instructor), then the right calls are to functions written by the student (or possibly the instructor). Again, the student should look for a

function whose signature says it consumes a value of the named type. If no suitable function is available, I ask my students to create a *work order* for a new function and to put it on an *order list* (see the discussion of "wish lists" in Section 3.2 below).

This analysis suggests when and how to write a helper function and when and how to look for a library function.

The book pushed me, implicitly, to teach elimination forms. But types can guide code in two other ways:

- In any function you have inputs, possibly structure elements, and the results of any natural recursions, which you put together to compute the answer you want. But how? You could use the method of tables and examples described under Step 5 in Section 3.1 below, which relies on insight. Or you could use types.

  If you are designing a function f that is obligated to produce a result of type τ_r, you can treat τ_r as a "goal type," and you can ask if there is a function available, either defined or on your order list, that produces a result of that goal type τ_r. If you find such a function g of type $\tau_a \to \tau_r$, perhaps you already have a value of type τ_a, or perhaps you repeat the exercise with τ_a as the new goal type. Your goal-directed search produces, as candidate expressions, well-typed compositions of functions. It helps you by limiting what compositions you consider.
- Last, and rarely, you could design a function's template around the introduction form for the result type.

When I teach *Program by Design* again, I will make my students aware of this decision point in the construction of a function's template: should they use elimination forms, function composition, or an introduction form? They should use elimination forms usually, function composition sometimes, and an introduction form rarely.

A data-description pitfall Because a data definition is informal English, it can include invariants and other properties, like the order invariant on a binary-search tree, which are difficult to express in simple type systems based on System F. This power is useful, but it also represents a potential pitfall. Because the role of a data definition is to guide the shape of code, programmers should use the power of informal English only as a last resort. For example, I would never say informally that a nonempty list of numbers

is a list of numbers that contains at least one number. I avoid this definition because it leaves students with no guide to the structure of a function that consumes nonempty lists of numbers. A property such as "a nonempty list of numbers" or "a list of an even number of strings" should be expressed inductively as part of the structure of a data definition. For example,

```
A *nonempty list of numbers* (lon+) is one of
   (cons n empty),   where n is a number
   (cons n ns),      where n is a number
                       and ns is a lon+
```

This data definition is the one to use for such functions as "minimum" or "maximum," which are defined only on nonempty lists—it tells a student exactly how such a function should be structured.

Creativity and constraint The design process has a lot of steps, and the textbook has a lot of rules and prescriptions. Some steps call for students to get creative; others call for them to respect the rules and prescriptions. To help students succeed, I tried to be explicit about which were which.

- To look at the world or at a problem, and to capture its essential aspects in a data definition, is in my opinion the design step that requires the most creativity. It is also the most challenging. Because systematic design begins with data definition, and because I wanted my students to build on solid definitions, I rarely had my students create data definitions; instead, I provided most data definitions. To give students, safe, relatively easy opportunities to create their own data definitions, I recommend using "world programs" (Section 4.2 below).
- The other design step that requires creative problem solving is turning a function template into code. This step requires a less difficult, puzzle-solving style of creativity, and I had my students do it all the time. To help them, I taught the method of tables and examples discussed under Step 5 in Section 3.1 below.
- The remaining steps of the design process reward order and method over creativity. Function signatures should mention only defined data and should be connected to words or relationships in the problem. Purpose statements must be written methodically and checked to be sure they are complete and comprehensive. Data examples should enumerate all possible shapes of data, and functional examples should also include

examples of all shapes. Templates should be developed systematically using one of the three methods listed above. Tests should come from functional examples; additional tests should be introduced only to help clarify function descriptions, to isolate bugs, or to prevent regressions.

Presenting functional abstraction and higher-order functions Both editions of *How to Design Programs* include sections on simplifying and generalizing code. In particular, the book shows how to combine two similar functions into one by abstracting over the parts that are different. Both the desire to simplify and the ability to abstract are essential for any working programmer, but they don't correspond to any step of the design process in the book, so I found them difficult to motivate. This difficulty will be resolved by making "review and refactor" an explicit, final step in the design process, as suggested above.

What about standard higher-order functions on lists? Functions like `ormap` (a.k.a. `any` or `exists`), `andmap` (a.k.a. `all`), `map`, `filter`, and folds? I debated whether to teach them; in part, I feared that identifying common patterns of recursion would be too difficult, or that I could not offer enough practice time. Eventually, I decided to teach these functions because they are prominent in the book, and they are a functional programmer's power tools. To justify this decision, I concocted, with help from colleagues, a story about preparing for the future:

> Processing data in sequence is very common, and most languages provide features that help. You will see "loops" with keywords like "for", "while", or "repeat"; you may see "iterators"; and if you use a fashionable language like Python, you might even see fancy "list comprehensions."
>
> Why do such features matter? Because, if you are a principled software designer and you use a language that provides its own bricks, you use the bricks that are provided—you don't bake your own funny-shaped bricks out of raw clay. In other words, you must learn when and how you can solve your problems using built-in looping features.

I then explained that Intermediate Student Language provides these bricks in the form of general-purpose functions that implement "loops" for search, selection, and transformation. It even provides two very general-purpose

functions that amount to "do something with each element": `foldl` works left-to-right, and `foldr` works right-to-left. My students learned to use these functions well enough, but I still don't understand how the functions fit into the steps of the design process.

3 Outcomes in the classroom; delivering the course

This section presents lessons I learned from teaching *Program by Design* at Tufts. Tufts is a private, Carnegie Research I university with very selective admissions. My course substituted for our usual first course in computing, which is required of all majors. Two-thirds of the students were in their first semester at university; most of the others were starting their third year. Most reported little or no prior experience with computer programming. Those who completed my course were eligible to continue to the second course in computing for majors, and most elected to do so.

My conclusions are drawn from observations in the classroom, in the laboratory, and of students' written work. I observed my students, my staff, and myself. For students, I address their learning about the design process and about some advanced topics. For myself, I confirm that my experiences are consistent with those published in the literature. Because my conclusions come from just a single case study, they are in no way definitive. But they should be informative, helping both you and your students.

3.1 Where students struggle & where they don't (design steps)

Not all steps of the design process are equally easy to learn. Here I report on students' experience with seven of the eight design steps. (I conceived of the "review and refactor" step too late to teach it.)

Step 1A: Data definitions My students had little trouble learning to write monomorphic data definitions. What trouble they did have arose from compositionality: although the elements of a sum or product type can themselves be sum or product types, some students thought at first that the elements of a sum or product type had to be base types. Had I chosen better examples, my students could have avoided this misconception.

A few students had a more subtle problem: they wanted to nest sums and products more deeply than is wise. As is implicit in the examples in the textbook, defining a sum of products of named types is a good strategy and

works well with the rest of the design method. But putting an additional sum or product under one of the nested products creates definitions that are harder to understand, and it militates against the effective use of helper functions.

Finally, late in the term, many students allowed their data definitions to get sloppy. Their most common fault was to conflate the name of a structure element with the name of its type.

Step 1B: Data examples A data definition is a kind of specification; like other specifications, it expresses what a programmer intends to model. So how do we tell if a data definition expresses the right intent? By writing *data examples*. Data examples also support problem-specific case analysis, e.g., which temperatures support deciduous trees? Or on a map, which hospitals lie in the local jurisdiction?

My students often forgot to write data examples. And initially, most of them struggled to write data examples that DrRacket would accept. The forgetting can be addressed, as suggested above, by teaching the construction of data examples as a discrete design step with its own number. The struggles with DrRacket should be addressed, in my opinion, by improving DrRacket so that it supports data examples as well as it currently supports *functional* examples. Let's look at that support.

For functional examples, each teaching language provides a syntactic form called `check-expect`. A `check-expect` may appear anywhere at top level; in particular, a `check-expect` that calls a function may appear before the definition of that function. Here is an example using pairs:

```
(check-expect
   (swap (make-pair 'fish 'fowl))
   (make-pair 'fowl 'fish))

;; swap : (pairof X Y) -> (pairof Y X)
;; return a pair that is equal to the given
;;   pair with the elements swapped
(define (swap p)
  (make-pair (pair-snd p) (pair-fst p)))

;; DATA DEFINITION of (pairof X Y) is elided ...
(define-struct pair (fst snd))
```

DrRacket accumulates uses of `check-expect`; waits until all functions, values, and structures have been defined; runs the accumulated uses in the context of the definitions; and finally rewards students by saying something like "All 16 tests passed!"

Data examples enjoy no comparable support:

- Data examples lack their own syntactic form; they are written either as top-level expressions or as right-hand sides of `define`.
- Data examples are not accumulated and summarized.
- Data examples are not rewarded by DrRacket. In fact, if data examples are written as top-level expressions, they are lightly punished: before reporting about tests, DrRacket sprays the values of top-level expressions to standard output.
- A data example that incorporates a structure must appear *after* the relevant `define-struct`. This requirement confused and frustrated my students, who tripped over it repeatedly. Students expected data examples to be like function definitions and `check-expects`, both of which can refer to functions and structures before they are defined.

My students eventually learned to place their data examples *after* all relevant definitions, but they deserve better. An idea of what "better" might look like appears in Section 6.1 below.

Step 2: Function descriptions A function description comprises a type signature, a *purpose statement*, and a *header*. The purpose statement is essentially Meyer's (1997) *contract* (precondition and postcondition), but it is informal and therefore not checkable by automated tools. The header names the parameters. Headers are easy, but for my students, learning to write good signatures and purpose statements was hard.

Students quickly learned the idea of signatures, but some students suffered from a misconception similar to their misconception about data definitions: that signatures could refer only to base types. With that misconception cleared up, students wrote signatures without difficulty, but like Crestani and Sperber's (2010) students, they wrote a lot of bad ones.

The worst kind of bad signature was imprecise because it contained ill-kinded types. Most often I saw a bare `list` (with no type parameter) or a bare `structure` (not identifying *which* structure). A less bad signature had a precise meaning that was inconsistent with the function it described,

usually because it had the wrong number of parameters. Both kinds of bad signature could be ruled out by making signatures linguistic, as described by Crestani and Sperber. Unfortunately, because I could not find documentation on using Crestani and Sperber's signatures with the standard teaching languages, I can't confirm their experience.

Writing good purpose statements is very hard, even for third- and fourth-semester students. I don't know of a royal road to writing purpose statements, but I do issue the following instructions, which relate purpose statements and function headers:

> Why is a function's header grouped with its signature and purpose statement? So you can use the *names* of the parameters in your purpose statement. Therefore, please make sure that your purpose statement refers to *each* parameter *by name*—and that it mentions the result.

The idea can be found in the textbook, but I needed to emphasize it. And checking purpose statements in a final "review and refactor" step of the design process is something even a raw beginner can do.

Steps 3 and 6: Functional examples and unit tests As noted above under Step 1B, functional examples are written using a special form called `check-expect`. This form, which is explained in the second edition of the textbook, shows two expressions that are expected to evaluate to equal values; a `check-expect` serves as both example and test. Even though we were using the first edition, I used `check-expect` for every functional example I presented.

My colleagues, my course staff, and I had worried about mandating a testing step for each function; students in our existing classes almost never write unit tests, and tests are seldom included in the work on which students are graded. We needn't have worried. My students quickly learned to write functional examples and to reuse them as tests. All students learned to use tests, and most students grew to value them highly. My most vivid example comes from a hallway conversation about the differences between *Program by Design* and our standard first course, which is taught using C++. When I explained some of the limitations imposed by C++, one student was dumbfounded: "You mean they don't have `check-expect`?"

Although my students learned to use examples and tests routinely, not all students learned to use them well. For example, students were slow to learn that if a function consumes a value that is defined by choices (a sum type), they should write a functional example for each choice. (If there is no test for a given choice, DrRacket reports in Step 6 that code written for the choice is not tested, but by then it is too late for the missing functional example to play its role in guiding the construction of the code, as explained in Step 5 below.)

More subtle, and harder to learn, was the idea that functional examples should include a representative variety of *results*. The easiest context in which to introduce this idea is a function that returns a Boolean; there should be examples that return both `true` and `false`. Had I known that such functions might present difficulties, I could have forestalled a few instances in which students mistakenly wrote predicates that always returned `true`, for example.

Step 4: Templates Function templates were the most difficult part of the design process for my students to apply. The students weren't confused, and they didn't ask questions, but most of them consistently turned in code that, to an instructor, was obviously not derived from a legitimate template. Worse, students could not see for themselves that they had deviated from the template approach. Several deviations recurred frequently; I call them *confused conditionals* and *stubborn sums and structures*. I also describe *false choices*. To illustrate these deviations, I describe two students' implementations of `insert`, a function that inserts a key and value into a binary search tree. Such a tree is either `false` (the empty tree) or a `node` containing a key, a value, and left and right subtrees.

The good implementation follows the template approach. Function `insert` begins with the elimination construct for the sum: a conditional that asks `false?` and `node?`. The case for `node` selects the node's key and continues with a three-way conditional that compares the node's key with the input key. Two branches of this conditional contain naturally recursive calls to `insert`, passing the node's left and right subtrees, respectively.

The bad implementation also begins with the correct conditional. But its `node` case deviates from the template approach. It passes all inputs (including the node) to a helper function `change-value`, whose type is too general: `change-value` expects not a `node` but an arbitrary tree. And `change-value` doesn't use a template based on input data; instead,

it calls a helper function `has-key?`, which tells if any node of the given tree contains the given key. Then `change-value` calls another helper function, either `update-value` or `add-node`. And `add-node` uses a three-way conditional to ask if the input tree is `false` or if the given key is smaller or larger than the key in the node. In this implementation, many things have gone wrong. Let's look first at conditionals.

A multiway conditional should either distinguish among alternatives in a sum type, like a `case` expression in Haskell or ML, or it should make some other single decision, like an `if` expression in Haskell or ML. The initial conditional in both implementations of `insert` acts like a `case` expression, distinguishing a node from an empty tree. Such conditionals are prescribed by the design recipe for sum types, and once that recipe is understood, they can be written quickly and easily. The second conditional in the good implementation of `insert` acts like nested `if` expressions, deciding how the input key relates to a node's key. It, like many similar conditionals, is prescribed by a design recipe for ordered data. The conditional in `add-node` is a *confused conditional*: it mixes the discrimination of alternatives in the sum with discrimination among keys. Such conditionals are not prescribed by any design recipe, and they are usually hard to understand.

Now let's consider the types of the bad implementation's functions. Functions `insert` discriminates between `node` and `false`, and it passes only a node to `change-value`. But `change-value` expects a tree, not a node. This tree is a *stubborn sum*—one that won't go away and has to be scrutinized repeatedly. Sure enough, trees are scrutinized by `insert`, `has-key?`, `update-value`, and `add-node`. That's four times as much scrutiny as the template (or the problem) calls for.

I've also seen *stubborn structures*. One example was in a function that takes a binary search tree keyed by number and returns the leaf whose key is closest to a given number. The template for a node structure should combine the node's key with computations on the node's left and right subtrees. But the example code examines the node's key, then calls a helper function with one subtree *and the node*. This node is a stubborn structure—instead of being abandoned once its elements have been selected, it is passed around, and multiple helper functions select elements from it, repeatedly.

Finally, a *false choice* is a conditional decision that ought not to be made, because one of the right-hand sides subsumes the others. One example is

the conditional in the bad implementation of `insert`: because the helper function `change-value` can handle *any* tree, not just a node, `insert` needs no conditional. I observed many other examples.

You can watch for these template problems, and you can warn students about them, but if students are to identify bad templates for themselves, I believe they need more support from DrRacket. Some ideas appear in Section 6.2 below.

Step 5: Coding As noted in Section 2.3 above, going from template to code sometimes requires creative puzzle solving. To help stimulate students' creativity, I have generalized a technique that is explained in the second edition of *How to Design Programs*, in the section on designing with self-referential data definitions.

The problem is to turn a template into code. The technique is to create one table of examples for each nontrivial case in the template's main conditional, or if the template does not begin with a conditional, one table for the whole template. The first column of the table is labeled *Wanted*, and it shows the value the function should return, which is taken from a functional example. Another column is needed for each application of a selector function and for each natural recursion. A labeled column for each input may also help, as may labeled columns for calls to helper functions.

Once the columns are set up, the student fills in a row for each functional example that meets the condition associated with the table. The *Wanted* column, the inputs, and the results of applying selector functions are filled in mechanically using the inputs from the functional example. Columns for natural recursions or for calls to helper functions are filled in using each called function's purpose statement. Here is an example of such a table for the induction step of a recursive function that sums the first `n` natural numbers:

Wanted	`n`	`(sub1 n)`	`(sum-to (sub1 n))`
1	1	0	0
10	4	3	6
15	5	4	10

We hope the student sees that *Wanted* is `(+ n (sum-to (sub1 n)))`.

I found the table-of-examples technique so valuable that after two-thirds of the course, I devoted a full homework assignment to it and to remedial template writing. The next time I teach the course, I will ask students, on the very first assignment, to fill in tables of examples, using images and perhaps a few numbers.

3.2 Where students do & don't struggle (advanced topics)

Self-referential data and natural recursion I was wisely advised to start teaching recursion not with lists but with a richer recursive type. As my first example, I defined a particular binary tree: a *conspiracy* is either an empty conspiracy or a cell headed by a person and containing two recruits, each of which is also a conspiracy. To create a running example, I claimed a position at the root of a class-wide conspiracy, and I sent emails to two students asking them to recruit two classmates each, and to "give your [recruits] these instructions and ask them to recruit two more classmates into the conspiracy." Students seemed to enjoy the play-acting, and when the conspiracy was revealed at the blackboard, students were able to help me evaluate and then define such functions as the number of people in a conspiracy (population) or the number of steps needed to get a message to every conspirator (depth). Also, when the time came to explain the function template for self-referential data, I was able to draw an analogy between a recursive function call and the self-reference in my informal recruiting instructions.

After this successful introduction to recursion, I thought we were home free, but as I watched students tackle more ambitious problems, I observed a dispiriting phenomenon: many students tried to understand recursion by mentally inlining recursive calls, arbitrarily many times. Thinking about sequences of recursive calls abandons the template approach, makes students' heads hurt, and leads to hideous, broken code. In the future, I will insist even more often that when you call a function, you must *not* look at its definition, but only at its description—and in particular, at its purpose statement. It is only by trusting purpose statements that a programmer can build things that are big or recursive. I don't know if my students picked up bad habits elsewhere or if the desire to inline functions is innate, but now I do know I have to fight against it.

Generative recursion One of the contributions of *Program by Design* is to distinguish *natural* recursion, which amounts to structural induction,

from *generative* recursion, which describes all other methods of dividing a problem into smaller subproblems (Felleisen et al. 2004a). Generative recursion provides opportunities for great homework problems, but I needed to give remedial homework on templates, and I had promised that students would do the language-classification project described in Section 5.1 below. So I assigned no generative-recursion homework. I was able to assess students' mastery of generative recursion only by observing them in lecture. There, we tackled the construction of a 2D-tree from a list of points, and I set sorting a list of numbers as a quiz problem. In both problems, at least some students were able to identify which approaches were structural and which were generative. For example, given the standard structure in which a list is either `empty` or is made with `cons`, students correctly identified insertion sort as structural and selection sort as generative.

Local definitions and lambda expressions My students easily made the transition from Beginning Student Language to Intermediate Student Language, which adds local definitions and first-class, nested functions. They also easily absorbed `lambda`. Although `lambda` is officially an add-on, we went straight from Beginning Student Language to Intermediate Student Language *with* `lambda`; we never used Intermediate Student Language without `lambda`. And because my mandate was to teach programming and problem-solving, not functional programming, I did not dedicate any class time to `lambda`; I simply used it in examples in which I called higher-order list functions. To my surprise, in lab exercises designed to reinforce skills with higher-order functions, at least half the students chose to use `lambda`, without encouragement or instruction beyond what they had seen in class and read in the book.

Functional abstraction and higher-order functions Abstraction over differences includes abstraction over different *functions*, and abstracting over a function produces a higher-order function. I taught higher-order functions using a suggestion from Viera Proulx: present purpose statements for functions that answer similar questions. For example, "How many students in this class carry a MacBook?", "How many students in this class are freshmen?", and so on. My students swallowed the idea whole and were able to design, during class, a higher-order `how-many` function. And they sailed through a homework assignment in which they used abstraction to combine functions they had written previously.

My students also had little difficulty using standard higher-order functions on lists. They did so not only on small problems that emphasized standard list functions, but also on a large project for which they were instructed to avoid recursion when possible. A number of students were comfortable enough to complain about the names of `ormap` and `andmap`, which Haskell programmers know as `any` and `all`. Students did only one thing that disappointed me: many of them used a fold where a map or filter would be better. Correcting this fault would be an appropriate refactoring.

Parametric polymorphism As part of its story about abstraction, *How to Design Programs* introduces parametric polymorphism. Data definitions can acquire type parameters, and type signatures can use universally quantified type variables. My students' written work showed that most of them got the *idea* of a parametric data definition, but none of them learned to use the notation properly. And even granting some idiosyncratic notation, few of them wrote definitions that were clear and unambiguous.

To correct these problems, I would follow Crestani and Sperber (2010) in introducing a formal language for data definitions. And although I do not advocate static type checking, I do think my students would benefit from a static check that type expressions are well kinded (each type constructor should receive the expected number of type parameters).

My students had more problems with polymorphic type signatures. Most of them sometimes wrote signatures that were less polymorphic than their code, using a named type where a type variable would be permitted. Many of them also sometimes wrote signatures that were more polymorphic than their code, using a type variable where a named type was required. These problems would not have been detected by Crestani and Sperber's dynamic signature checker: the functions whose signatures aren't polymorphic enough aren't *used* at non-conforming types, and the functions whose signatures are too polymorphic won't be detected because a type variable does not trigger any dynamic checks.

Wish lists *How to Design Programs*, especially the first edition, emphasizes the design of functions over programs. But it does present one key tool for designing programs, which it calls the *wish list*. The wish list is a list of descriptions, each including a name, signature, and purpose statement, of functions that need to be written for the program to be complete.

Unfortunately, I never saw a student use a wish list effectively. And I often saw students use wish lists *ineffectively*: instead of being demanded by demonstrated needs, functions appeared on the wish list after a quick reading of a problem, without thought. The wish list turned into a fantasy list, containing anything a student might possibly wish for. Such lists result from muddy, wishful thinking about problems, not from systematic design.

To fight against muddy, wishful thinking, in future courses I will avoid the term "wish list." I will instead refer to an "order list" and to "work orders." I will tell students that issuing a work order costs something, and they had better not order a function unless they're willing to pay for it. I look forward to seeing if the new words help.

3.3 Replicating others' experience

My classroom experience confirms what others have written about dynamic types, about lecturing, and about laboratories.

Dynamic types Findler et al. (2002) argue that the type systems of Haskell and ML are too sophisticated for beginning students, but that a first-order, monomorphic type system might be helpful for beginning students. Felleisen et al. (2004a) argue that dynamic typing is a benefit because students and teachers need not spend energy finding and explaining type errors. My students' difficulty writing well-formed templates (Section 3.1) suggests that writing statically well-typed code might also be a challenge.

Live coding in the lecture theater Sperber and Crestani (2012) recommend that instructors teach design by solving problems using the full design process, with DrRacket, before a live audience of students. They caution against taking shortcuts. I found this method of teaching most effective during the second half of the course. I also found that a 75-minute lecture is too short for complete, correct solution of such problems as designing higher-order functions proposed by the students, or building a 2D-tree. I had to choose between dropping examples and taking shortcuts, and I took shortcuts. (When taking a shortcut, I identified each design step I wished to skip, and I asked students' permission to skip it.) You may need to make similar compromises.

Laboratory experiences and assisted programming At Tufts, instruction is limited to 150 minutes of lecture per week, plus a 75-minute lab.

A lab accommodates up to 22 students and is supervised by a staff of two or three undergraduate assistants, plus a "lab runner," who is typically a doctoral student. My class was limited to 40 students, so I needed only two labs, which I ran myself—primarily so I could observe students at work.

In lab, I tried to replicate the *assisted programming* model described by Bieniusa et al. (2008): students are given a set of small programming exercises, of which they are expected to finish half. Students worked in pairs, and I asked them, at the end of each lab, to write what they had done and what they learned. Although personal observation told me more, the self-assessments helped me judge students' learning and address issues in subsequent lectures. And self-assessments scale in a way that personal observation doesn't.

My labs presented many of the same issues described by Bieniusa et al., especially the construction of exercises with a suitable number of problems of suitable difficulty. My most popular labs were those that posed many small problems. Examples included a list lab that asked for one data definition and ten functions, and a higher-order functions lab that asked for ten functions and the results of several function applications. My least popular labs were those that posed a single problem broken down into many pieces. Examples included a lab to convert any S-expression into a sequence of atoms (and back again); a lab to build a game of whack-a-mole; and a lab to build an interactive map of the northeast United States, highlighting the hospital nearest the mouse cursor. No student completed any of these labs, so students did not enjoy the early successes that so help their motivation and learning (Ambrose et al. 2010, Chapter 3).

Other instructors report being challenged to develop good labs that work in 90 or 120 minutes. A 75-minute lab is even more challenging. If possible, arrange for a longer lab.

4 Working with the languages, libraries and tools

In this section I explain what I learned about DrRacket, the teaching languages, and the teaching libraries.

4.1 Using the teaching languages with DrRacket

For over fifteen years I have taught programming languages using little languages (Kamin 1990; Ramsey 2016). With this experience as back-

ground, I cannot praise the Racket teaching languages highly enough. The language design is lapidary. I was especially impressed that Beginning Student Language functions may not have local variables. At first I thought this restriction was crazy, but after observing students at work, I see that not only is the language simplified, but without local variables, students are nudged to create helper functions—a notorious point of difficulty for beginning students.

Including `check-expect` is a masterstroke. Even if you use only the first edition of the textbook, you must teach `check-expect`, because it is so beautifully integrated with DrRacket. Clicking Run runs all tests, and DrRacket shows untested code in red on a reverse-video background. After seeing this feature demonstrated in one or two early lectures, almost all students routinely submitted code with complete "statement" coverage. They submitted untested code only in assignments that were substantially incomplete. Why? Probably because *every time you compile, DrRacket runs the tests and tells you about coverage*. When my fellow instructors and I compare programming environments, we agree that easy, routine, automatic testing and coverage analysis is DrRacket's most important benefit.

4.2 Teaching with world programs and the universe library

The teaching languages come with purely functional `image` and `universe` libraries, which can be used to create interactive graphical applications as well as distributed applications (Felleisen et al. 2009). Interactive applications are called "world programs," and my students wrote lots of them. (We did no distributed computing.) A world program is built around a single higher-order function, `big-bang`,[3] which has a polymorphic type. The unspecified, universally quantified type is called the *world state*.[4] Client code provides a function to render a world state as an image, as well as pure functions that respond to mouse and keyboard events, or even to the passage of time, by mapping world states to world states. The design of world programs is discussed briefly by Felleisen et al. (2009) and at length in the second edition of *How to Design Programs*. My summary guide is reproduced in Appendix E.

[3] Actually, `big-bang` is a syntactic form, but you don't need to know this.

[4] Felleisen et al. explain world programs using units (modules), but an explanation using polymorphism and type variables also works.

World programs impressed me very favorably: `big-bang` is both powerful and simple, and creating satisfying interactive programs is easy. But world programs have more intellectual depth than I realized, and I made some mistakes (Section 5.4 below). I trace my mistakes to a shallow understanding of the `universe` library; I was too willing to take at face value the idea that the purpose of the library is to enable students to "construct a program that is like the applications they use on their computers" (Felleisen et al. 2009). I now believe the library serves broader and deeper purposes:

- The library provides a simple space in which students can develop and practice the skill of "look at the world; see data; define a representation in the computer."
- The library provides a safe, guided environment in which students can design *programs*, not just functions.
- The library exposes students to the power of data abstraction (over the world state).
- The library provides flexibility for students to choose different representations of a world state and to design the event handlers required by `big-bang`. This kind of flexibility, and the control students have over their choices, enhance motivation and learning (Ambrose et al. 2010, Chapter 3).

Pleasing students with lifelike applications is all very well, but world programs are important because of their other purposes. In the future, the aspects I will emphasize most are the skill of modeling the world in the computer, and the practice in designing programs, not just functions.

4.3 What to expect from the programming environment

Except for the help and menu system, I found the student-facing part of DrRacket as good as advertised. Almost all of my students were instantly productive using the Beginning Student Language. The help and menu system does present a problem: as far as I can tell, students are expected to deal with the same help and menu system that fully fledged Racket programmers use. Most students were willing to ignore menu items they didn't understand, but almost all of my students tried to use the help system and found themselves reading documentation for full Racket—especially library documentation. This documentation, with its idiosyncratic notation for function signatures, was difficult even for my teaching assistants. Unless this problem is corrected, you will want to prepare for it.

The instructor-facing part of DrRacket surprised me. I was expecting mature, well-documented, stable production software. I got mature, well-documented, evolving *research* software. Once I adjusted my expectations, I got along fine, but I hit a couple of pain points worth knowing about. Because I hope these pain points will soon disappear, I have relegated the details to Appendix D.3.

One pain point is not going to disappear: if you write libraries, you are expected to use full Racket. If, like me, you've learned only up to Intermediate Student Language, full Racket presents some problems: it's not just bigger; it's different. I tripped over differences in the definitions of structure types and in the meanings of numeric literals. With help from Matthew Flatt, I found a workable compromise: I write my libraries in Intermediate Student Language, and in full Racket I write only a small file that exports the `provide` form. By importing this file into my library, I extend Intermediate Student Language with `provide`, and this extended language is sufficient to create a usable library.

One final caution: it is all too easy for a student to use DrRacket's menus to import the wrong library ("teachpack") by mistake—a mistake that both students and teaching assistants found hard to diagnose. Insist that your students import libraries only by using `require` in their source code. Using `require` makes manifest what libraries have been imported, and as a bonus, it puts your libraries on the same footing as built-in libraries.

5 Rookie mistakes and what I learned from them

It's great to do things well, but we learn the most from our mistakes. I asked other instructors to help me learn from their mistakes, but those who made beginner's mistakes did not share them. Shriram Krishnamurthi did identify two common mistakes: failing to get complete buy-in from teaching assistants, and allowing experienced students to disrupt or undermine the class. What follows is an account of my own most significant mistakes—the ones from which I learned the most, and the ones I most wish I had avoided.

5.1 Misdirected effort in preparation and planning

I began preparing my course by trying to identify learning outcomes, in more detail than I present in Section 2 above. I read the textbook painstakingly and took detailed notes. This work turned out to have been

a poor use of my time. I later skimmed the book *quickly* and made a high-level summary. The summary, which splits the material into six broad tiers and articulates a simple learning goal for each tier, helped me far more than my detailed notes. It still does not contain what my colleagues in education would consider proper learning objectives, but in hopes that it may also help you, I have reproduced it as Appendix B.

I worked on the course with seven students who had studied functional programming with me. None of us had used Racket or its teaching languages; we had used a dialect of Scheme called μScheme, which is a bit smaller than Racket's Intermediate Student Language. We did not try to learn the teaching languages in advance, which was a good decision: we were all able to draw on prior experience to pick them up quickly and easily.

We spent our preparation time on potential homework assignments. Because our departmental culture encourages "projects," which are big, open-ended assignments intended to provide scope for significant design choices (Appendix C), we focused almost exclusively on project ideas. We especially wanted projects that would meet our departmental goals of establishing connections to real-world technology, to real-world data, or to students' interests outside of computer science. In the light of experience, our focus was misdirected.

- In almost every week of a *Program by Design* course, students learn a new way to organize data. But coming up with projects that organize data in sufficiently diverse ways was beyond our abilities. Almost every one of our project ideas required a list of structures, and for many ideas, a list of structures was sufficient. The fit with *Program by Design* was bad: before students are ready to work with a list of structures, they have to spend a month learning simpler forms of data. They then have only a week or two in which lists of structures are on topic, after which they move on to other forms of data.
- In *Program by Design*, students learn so much technique that there isn't *room* for a lot of projects. In a 13-week course, even though I chose not to teach mutation, I felt that I had only about $3\frac{1}{2}$ weeks in which I could give students a project that was not driven by a technical learning objective. I was able to use only one of those weeks for a project. (I used another week to remediate difficulties with templates, and I used the

remaining week and a half to help my students prepare a learning portfolio, which served them in lieu of a final examination.)

My staff and I also looked for problem domains that could serve as unifying themes for multiple labs and homeworks. We settled on two themes: probability and GPS navigation. I knew probability was a reach, but I wanted to deliver a project that has been popular in our first course for several years: write a naïve Bayesian classifier that identifies the natural language in which a web page is written. I was more confident in GPS navigation: I felt that it would provide a more interesting introduction to numeric computation than the ancient, boring Fahrenheit/Centigrade conversions, and I felt it would lead up to interactive mapping applications. But neither of the two themes worked out as well as I had hoped.

- We didn't have time to take probability seriously. We started well enough by having students estimate and measure some real-world probabilities, using log odds. We then ignored probability for ten weeks, and in the eleventh week, I bombed students with a few dense pages of probabilistic notation and Bayesian reasoning, so they could build classifiers. I don't fool myself that they retained anything.
- We did better with GPS navigation, but I underestimated my students' discomfort with sines and cosines. Not only did my students find sines and cosines intimidating, but sines and cosines use "inexact" (floating-point) arithmetic, which I could otherwise have delayed or avoided. Many students struggled to write simple functions on GPS coordinates, which took them far more time and effort than I ever imagined.

These two problem domains may or may not have been poor choices, but my real blunder was more fundamental. Only a person whose early education was warped by experience with Pascal and C would think that examples in the first course should begin with *numbers*. Beginning Student Language includes a first-class *image* type. This type comes with a lovely algebra of operations, and it even enjoys special support in DrRacket's read-eval-print loop! Or if I hadn't thought of images, as a longtime Haskell and ML programmer I should definitely have thought of *strings*. I promise all future students that their education in computing will begin with examples that draw pictures and say things, not with boring or intimidating numerical calculations. Bloch (2010) agrees.

What else did I learn from my mistakes in course planning?

- I found room for only one or two things beyond basic functional programming. The textbook suggests mutation, but I chose instead to take an extended look at tree structures (1D- and 2D-trees) and to use a novel final assessment (learning portfolios).
- I believe in projects, but when I teach the course again, I will identify *one* project and have the students build it in pieces throughout the term. I might try a simple web browser or perhaps a browser for some other kind of database. I would consider a game like Scrabble, which would provide practice in data structures and in designing world programs, but my department is cautious about games (Appendix C).

5.2 Miscalibrated homework

My most embarrassing mistake was to assign a problem I thought was simple without first having completed the *entire* design process myself. I asked students to write three functions on GPS positions: distance, bearing, and projection. I had previously implemented the functions, and I knew that the function descriptions and codes were simple. I also knew that some of the trigonometry was subtle, so I prepared my students thoroughly for the trigonometric calculations. I thought that was enough.

I was wrong: I badly misjudged the cost of developing functional examples and unit tests. When I finally finished my reference solution, the code itself took only 24 lines of Beginning Student Language, even with liberal use of helper functions. But to test it properly, I had to define at least another half a dozen functions, and the full solution contained 226 nonblank lines of code, tests, and documentation. The assignment turned out to be about three times as much work as I had meant to ask for, and I was lucky my students did not desert *en masse*.

5.3 Misunderstood templates

As a beginner, I was a little too eager to construct function templates by leaping at the elimination form for one of the argument types, as described in Section 2.3 above. I had learned from the book that when you get a value of sum or product type, you take it apart using a conditional or a set of selector functions. But there's always another choice: you can leave an argument alone, not inspect it or take it apart, but simply pass it to another function. I didn't teach my students this choice early enough.

The possibility, however, can be taught from the very beginning; indeed, values of atomic type can *only* be passed to other functions.

Midway through the term, I tried to correct my mistake by introducing a new word for an uninspected value: *sealed*. The decision about whether to leave arguments sealed comes into play in the book's section on processing multiple pieces of complex data, but I wish I had introduced it earlier. Delaying may have contributed to my students' difficulties with templates and to my own difficulties in teaching function composition.

5.4 My world-state disaster

When I introduced world programs to my students, I made my biggest mistake of the term. I wanted to show them an interactive graphics program that did something interesting, and they had learned about structures but not yet about lists. I somehow got the idea of a program that would drop a disk on the screen at every mouse click, potentially filling the screen with disks. No lists? No problem! I chose as my world state an *image* containing all the dropped disks. Had I been *trying* to sabotage myself, I could not have chosen a worse example. For weeks, my students conflated world states with images, and when asked to write new world programs, they struggled mightily. To get everybody sorted out on the difference between an image and a world state took my teaching assistants a month of hard work. Next time I introduce world programs, I will begin with a simple state containing just one disk which can change position.

6 Open problems

During the semester, I identified a number of teaching problems that I have not yet solved. Some problems require Racket programming that is beyond my skills; some require a depth of understanding that I have not yet developed; and some require time, effort, and in-class experimentation that I have not yet been able to invest. I begin with easier problems and move to more difficult ones.

6.1 Making data examples first-class

In Section 3.1 above, in Step 1B (data examples), I enumerate the ways in which data examples are second-class citizens, not supported by the teaching languages or by DrRacket. All the problems my students had could be

addressed by adding a syntactic form for accumulating and then evaluating data examples, which would enable DrRacket to report something like "All 7 data examples built!" before reporting the results of `check-expects`.

6.2 Enabling templates to persist and be reviewed

DrRacket does not provide enough support for function templates. The teaching languages do include forms such as `...` and `.....`, which can be used to *write* templates. But DrRacket does not recognize these forms as special, and it complains that they are untested code. Untested code is anathema, and DrRacket's complaints push students to turn templates into code as soon as possible. And once a template has been turned into code, it is gone forever.

Because templates disappear, a student cannot review a template to see if it makes sense in the context of a given signature and data definition, and a student cannot compare a template with a function definition to see if the two are consistent. Bad templates account for almost all the times my students wrote horrible code or went off the rails entirely. And students don't see them! As an experienced functional programmer, I can look at a function and imagine the template from which it was derived, and I can identify problems that stem from the template in my mind. But such acts of critical imagination are too much to expect of beginning students.

Every other step of the design process (data description, data examples, function description, functional examples, code, and tests) leaves behind a visible artifact that can be assessed. Templates should be able to leave footprints, too. It might be enough to extend DrRacket with a new syntactic form, perhaps called `define-template`, which would define a new species of function. Such a "template function" would undergo the same static checks as a regular function, would be required to contain the `...` form or related forms, would be expected not to be tested, and could coexist with a true function of the same name. If I could change only one thing about DrRacket, making templates explicit and persistent would have the biggest effect on my students' learning.

6.3 Developing the "review and refactor" step

I plan to teach an explicit "review and refactor" step not only to unify some disparate instructions and activities that are distributed throughout *How to Design Programs*, but also to show students that mature designers

don't just write good code; they improve code by refactoring. To identify review and refactoring activities and to match them to levels of learning and development, much work remains to be done. As a first step, here are some suggested activities, starting with those suitable for very beginning students:

- Check signatures for arity problems, references to unqualified "lists" or "structures," and other faults. (Crestani and Sperber (2010) observe that this activity can be profitably automated by incorporating formal signatures into a teaching language.)
- Check functional examples to be sure every choice of input is represented.
- Check functional examples to be sure every possible sort of *output* is represented. This activity is especially valuable for functions returning Booleans.
- Examine code for violations of the template approach, especially the "confused conditionals" and the "stubborn" sums or structures described in Section 3.1.
- Look for duplicate or near-duplicate codes; identify parameters to abstract over.
- Scrutinize functions that have similar purpose statements (specifications) and consume the same kind of data. Identify and eliminate redundancies.
- Look for functions that take one or more arguments of sum, product, and arrow types. Identify which arguments are "inspected" (by `cond` or selectors) and which are "sealed" (ignored or passed to other functions). Decide if the decision to inspect or seal makes sense or if the code would be improved by deciding differently. Especially, look for arguments that are inspected but could be sealed.
- Eliminate conditionals in which one case can subsume the others.
- Look for recursive functions that consume lists and can be expressed using standard higher-order list functions.
- Look for uses of `foldl` and `foldr` that can be rewritten using `map` or `filter`.
- Look for recursive functions with similar structures, and replace them with new higher-order functions.
- Review type signatures of polymorphic functions. For each type variable, try substituting different actual types, such as image, Boolean, and list of number. Verify that *after* substitution, each signature accurately

describes the types of data that you expect to flow into and out of the function.

6.4 Developing better guidance for conditionals

How should students design conditionals? What role(s) should `else` play? When we review a conditional expression, how do we tell if it's good or bad? How can a *student* tell if a conditional expression is good or bad? I can answer only in two situations:

- An experienced Haskell or ML programmer knows that pattern matching in `case` expressions is most easily understood when patterns are non-overlapping, so the behavior of the program is independent of the order in which the cases appear. Each case can be understood in isolation, without considering the others. The corresponding principle in the Racket teaching languages is that when `cond` is used to choose among alternatives in a sum type, each alternative should be identified by an appropriate predicate. For example, a function that consumes a list `xs` should use the predicates `(empty? xs)` and `(cons? xs)`; it should not use `else`. However, in the first edition of *How to Design Programs*, students will see `else` used more often than "`(cons? xs)`." In the second edition, "`(cons? xs)`" is used more often.
- When a `cond` uses just two predicates, they are nontrivial, and they are complements, use `else`. For example, this code from a student would be clearer with `else`:

```
(cond [(look-across? tree close x y) ...]
      [(not (look-across? tree close x y)) ...])
```

Beyond these two situations, I don't know what to tell my students. And while I myself can usually look at a conditional and distinguish good from bad, I don't know how to teach graders to do it.

6.5 Assessing students' programs

The open problem that most affected my students' learning was that I found no clear, principled basis on which to grade students' programs. To learn, students need timely feedback on their work, but because my staff and I got bogged down in grading, they didn't get it. The big grading question is clear: we want to know if our students are practicing systematic design. What is unclear is how to evaluate systematic design, and on what scale.

Many instructors use a system of points. For example, Mitch Wand uses a detailed rubric graded on a 50-point scale and containing over 65 potential deductions. Unfortunately, this rubric is designed for beginning master's students, and it assumes an in-person code review. I did not understand the principles used to create the rubric and so could not adapt it for my situation. Also, just as instructors in Germany have special concerns about plagiarism (Bieniusa et al. 2008), I, like many other instructors in America, have concerns about wrangling with students over points.

As a promising alternative to a points system, the education literature recommends that we identify *primary traits* to look for in students' work, and that we evaluate each trait on a scale with three to five choices (Stevens and Levi 2005; Walvoord and Anderson 2011). I have used this alternative successfully in our third and fourth courses. In principle, these courses use the same five-point scale that the NSF uses to grade proposals: Excellent, Very Good, Good, Fair, and Poor. In practice, the two extreme grades are rarely used and are easy to identify;[5] the "normal" grades are Very Good (meets all expectations, equivalent to an American A), Good (does not meet all expectations but shows evidence of quality and significant learning), and Fair (the lowest passing grade).

To apply this scale to a course, the instructor must identify characteristics of work that is Very Good, Good, or Fair. But my staff and I were able to identify only Very Good and Fair characteristics:

- Very Good work may contain flaws, but it shows evidence throughout of having been developed using the design process.
- Fair work shows a *systemic* failure to apply the appropriate design recipe. For example, a solution would be graded Fair if every function's purpose statement merely restated the information given the function's signature.

We were not able to identify criteria by which to place students' work *between* Very Good (developed according to a design recipe) and Fair (systemic failure of design). And we are not comfortable grading on a two-point scale.

An ideal analysis of primary traits characterizes what is observed about each trait for each level of performance. But a partial analysis, in which

[5] Excellent work exceeds expectations and impresses the course staff (an American A-plus), and Poor work shows evidence of serious deficiencies, typically by being substantially incomplete (a failing grade).

only the highest levels of performance are characterized, can also be useful. For instance, Jordan Johnson has developed a list of 27 characteristics of exemplary work in *Program by Design*. To assign a grade, Johnson counts how often these characteristics appear, on a scale of Always, Usually, Sometimes, Seldom, and Never. He reports good results, but his classes are small—at most 14 students each. In my class of 40 students, we tried to replicate the "counting" approach for a just a few characteristics on a couple of homework assignments, with a coarser scale. But our graders reported that even this small amount of counting was time-consuming and stressful, and I felt that the results did not value students' work very accurately. For students, I expect Johnson's characteristics would make a fine checklist, but for graders, the counting approach is too expensive and does not lead to an obvious grade.

Another alternative is to base grades on a program's functional correctness, perhaps as determined by testing. In *Program by Design*, test results are less important than systematic design, but students do wish to be rewarded for producing "working" code. Automated testing finds bugs effectively (Claessen and Hughes 2000; Crestani and Sperber 2010). But automated tests require well-specified interfaces, and an essential aspect of *Program by Design* is that the interfaces are designed by the *students*, not the instructor. Were I to specify interfaces for students to implement, I would be doing much of the design work that I want them to learn to do.[6]

Bieniusa et al. (2008) use a "semi-automatic" tool that checks a student's program and assigns a preliminary "score." But the tool appears to require interfaces to be specified. And unfortunately, Bieniusa et al. do not discuss the set of possible scores, algorithms by which scores are assigned, principles on which such algorithms are based, or instructions given to the teaching assistant who converts the preliminary score to a final score.

A principled grading method that lies outside the context of *Program by Design* is described by Edwards (2003): students submit both code and tests, and the submission is scored by multiplying three fractions: the fraction of the student's tests that are consistent with the problem statement, the fraction of the student's tests that the student's code passes, and the fraction of the *instructor's* code *covered* by the student's tests. Like other testing approaches, this approach limits students' freedom to design.

[6] I have written a prototype that *discovers* students' interface designs by probing their code, but it relies on compile-time type checking.

Program by Design's method enables yet another approach, with which we can assess functional correctness without limiting students' freedom to design: we assess correctness by reading purpose statements and unit tests (functional examples). DrRacket tells us which code has actually been executed. If a function's purpose statement is clear, the code has been tested, and the tests seem sufficient to validate the purpose statement, the function is deemed correct. This approach gives students the freedom to design interfaces, but compared with automated approaches, it is significantly more expensive.

7 Conclusion

Principled course design focuses not on material but on students: what they can do, and how we know they can do it (Wiggins and McTighe 2005). *How to Design Programs* is a great source of material, and prior work (Bieniusa et al. 2008; Crestani and Sperber 2010; Sperber and Crestani 2012) tells us a great deal about how to teach it. This paper adds to that work, showing some significant mistakes to avoid, and telling us more about students: what they learn to do, and where they do and don't struggle in learning to do it. Plenty of problems are still open, of which the most difficult is assessing whether students can do what we think they can do: we need reliable, cost-effective ways of knowing when and to what degree students are really programming by design.

Acknowledgments

For help with the manuscript, Stephen Bloch, Matthias Felleisen, Shriram Krishnamurthi, Ben Shapiro, Mike Sperber, Aaron Tietz, Mitch Wand, and Jayme Woogerd.

For analysis of senior surveys, Dawn Terkla and Lauren Conoscenti.

For help preparing the way, Sam Guyer, Ben Hescott, Kathleen Fisher, and Carla Brodley.

For preliminary planning and for learning portfolios, Ariel Hamlin.

For educational matters, Annie Soisson and Donna Qualters.

For help teaching, in addition to those named in the text, many members of the `plt-edu` mailing list, including Stephen Bloch, Matthew Flatt, Gregor

Kiczales, Shriram Krishnamurthi, Viera Proulx, and Mitch Wand. Especially Stephen and Viera.

For help above and beyond the call of duty, provided to me and to my staff, Matthias Felleisen.

For making it happen, the students and staff of COMP 50, Fall 2013.

References

Susan A. Ambrose, Michael W. Bridges, Michele DiPietro, Marsha C. Lovett, Marie K. Norman, and Richard E. Mayer. 2010. *How Learning Works: Seven Research-Based Principles for Smart Teaching*. Jossey-Bass higher and adult education series. Wiley.

Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler, Michael Sperber, Marcus Crestani, Herbert Klaeren, and Eric Knauel. 2008. HtDP and DMdA in the battlefield: A case study in first-year programming instruction. In *FDPE '08: Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*, pages 1–12, New York, NY, USA. ACM.

Richard Bird and Philip Wadler. 1988. *Introduction to Functional Programming*. Prentice Hall, New York.

Stephen Bloch. 2010. *Picturing Programs: An Introduction to Computer Programming*. College Publications (Kings College London).

Frederick P. Brooks, Jr. 1975. *The Mythical Man-Month*. Addison Wesley, Reading, MA.

Hugh Burkhardt and Alan H Schoenfeld. 2003. Improving educational research: Toward a more useful, more influential, and better-funded enterprise. *Educational Researcher*, 32(9):3–14.

Koen Claessen and John Hughes. 2000 (September). QuickCheck: a lightweight tool for random testing of Haskell programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00),* in *SIGPLAN Notices*, 35(9):268–279.

Marcus Crestani and Michael Sperber. 2010 (September). Experience Report: Growing programming languages for beginning students. *Proceedings of the Fifteenth ACM SIGPLAN International Conference on Functional Programming (ICFP'10),* in *SIGPLAN Notices*, 45(9):229–234.

Stephen H. Edwards. 2003 (September). Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing*, 3(3).

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, first edition.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004a. The structure and interpretation of the Computer Science curriculum. *Journal of Functional Programming*, 14(4):365–378.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004b. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14(1): 55–77.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2009 (August). A functional I/O system or, fun for freshman kids. *Proceedings of the Fourteenth ACM SIGPLAN International Conference on Functional Programming (ICFP'09),* in *SIGPLAN Notices*, 44 (9):47–58.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182.

David R. Hanson. 1996. *C Interfaces and Implementations*. Addison Wesley.

John Hughes. 1989 (April). Why functional programming matters. *The Computer Journal*, 32(2):98–107.

Michael A. Jackson. 1975. *Principles of Program Design*. Academic Press, London.

Samuel N. Kamin. 1990. *Programming Languages: An Interpreter-Based Approach*. Addison-Wesley, Reading, MA.

Bertrand Meyer. 1997. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, second edition.

Norman Ramsey. 2016. *Programming Languages: Build, Prove, and Compare*. Cambridge University Press. Forthcoming.

Emmanuel Schanzer, Kathi Fisler, and Shriram Krishnamurthi. 2013. Bootstrap: Going beyond programming in after-school computer science. In *SPLASH-E (Education track of OOPSLA/SPLASH)*.

Michael Sperber and Marcus Crestani. 2012. Form over function—teaching beginners how to construct programs. In *SFP12: Scheme and Functional Programming*.

Danelle D. Stevens and Antonia Levi. 2005. *Introduction to Rubrics: An Assessment Tool to Save Grading Time, Convey Effective Feedback, and Promote Student Learning*. Stylus.

Barbara E. Walvoord and Virginia Johnson Anderson. 2011. *Effective Grading: A Tool for Learning and Assessment in College*. Wiley.

Grant P. Wiggins and Jay McTighe. 2005. *Understanding by Design*. ACSD, Alexandria, VA, USA, second edition.

A Teaching experience

My teaching experience includes a track record of creating required programming courses that have long-term impact. One measure of this impact is our university-wide survey of graduating students, which asks them about highlights of their four years at Tufts. In 2012, 16 graduating students named me as an influential individual who contributed significantly to their development, and 11 named one of my courses as an exemplar of "what a truly excellent college course should be." In 2013 the numbers were 16 and 18, respectively. (Students named both the third and fourth courses in our programming sequence.) Among the hundreds of faculty named on the surveys, the responses to my teaching place me in the 98th, 98th, 95th, and 99th percentiles.

Note to reviewers: The following four pages contain four additional appendices, which are, as stated in the call for papers, optional. If the paper is accepted, I do not intend to include these appendices in the ICFP proceedings. Instead, I will make the additional material available on the Web.

B High-level learning outcomes

This section presents my high-level summary of learning outcomes, as mentioned in Section 5.1. All figure and page numbers refer to the first edition of *How to Design Programs*.

Level 0: Tyro (sums of products)

Tyros learn simple computation on data descriptions that can be written in a way that defines each data class completely before any other description uses that class.

- Simple design recipe; named functions; `define`; auxiliary functions arise from dependencies in the problem [Sections 1 to 3, design recipe Fig 4 page 21]
- Conditional expressions and functions; `cond` [Section 4, design recipe Fig 6 page 44]
- Symbols [Section 5] and *strings* [2nd edition, Section 1.2]
- Structures, `define-struct` [Section 6, design recipe Fig 12 page 71]
- Mixed data, type predicates such as `number?` and so on [Section 7, design recipe Fig 18 page 89]
- BNF Grammar for Beginning Student Language [Intermezzo 1]

Level 1: Beginning student (lists and trees)

Beginning students learn to work with data descriptions that refer to themselves and so cannot be completely defined before they are used. The most common such data are lists and trees.

- Defining and consuming lists using `empty`, `cons`, `first`, `rest`, and `empty?` [Section 9, design recipe Fig 26 page 132]
- Producing lists; lists of structures [Section 10]
- Peano numerals; `zero?`, `sub1` [Section 11]
- Problem-solving; auxiliary functions [Section 12]

- List abbreviations [Intermezzo 2]
- Trees, nested lists (using existing structure and list primitives) [Section 14]
- Groups of mutually referential data definitions [Section 15, design recipe Fig 43 page 218]
- Problem-solving: iterative refinement [Section 16]
- Forms of case analysis with multiple complex arguments [Section 17]

Level 2: Intermediate student (abstraction)

Intermediate students learn a key technique of computer science: abstraction.

- `local` definitions of variables and functions; lexical scope [Intermezzo 3]
- Don't Repeat Yourself: abstracting similar functions and similar data definitions [Section 19]
- Functions are values: `filter1`, `map`, `sort`, parametric polymorphism; *"loops"* [Section 20]
- How to design abstractions; single point of control; clone and modify considered harmful [Section 21]
- Designing abstractions with 1st-class functions [Section 22]
- Examples: sequences, series, graphing [Section 23]
- Anonymous functions with `lambda` [Intermezzo 4]

Level 3: Recursive reasoner

Recursive reasoners have the insight to find a recursive decomposition even when the decomposition is not there in the data. And they can write recursive algorithms that remember past actions.

- Generative recursion; quicksort [Section 25]
- Problem-solving: algorithm design; termination [Section 26]
- Extended examples: fractals, files, Newton's method, Gaussian elimination [Section 27]
- *Input via state machines* [No book coverage]
- Remembering the past with accumulating parameters [Sections 30, 31, 32]
- Graph algorithms and search [Section 28]

Level 4: Cost container (costs)

Cost containers can reason about costs.

- Cost modeling, vectors, big *O* notation [Section 29]
- Inexact (floating-point) numbers [Intermezzo 6]

Level 5: Memory mutator (mutation)

Memory mutators cut costs by using mutable state.

- Mutable variables and mutation (`set!`) [Sections 34 to 37]
- Syntax and semantics of Advanced Scheme [Intermezzo 7]
- Abstraction with mutable state [Section 39]
- Mutable structures and vectors [Section 40]
- Mutating elements (atomic or structured) [Section 41]
- Extensional and intensional equality [Section 42]
- Mutation practicum: quicksort, cyclic structures [Section 43]

C Curricular constraints at Tufts

Curricular decisions are local. To put my decisions into their proper context, I highlight our most salient local constraints.

- Our department has historically favored the teaching of programming; we are not pressured to teach "computational thinking" or a new kind of science or anything else. We believe that if students learn good methods of programming, they will also learn good methods of thinking.
- We admire and respect the work that Bob Sedgewick has done at Princeton, which I characterize as a combination of "computation as a science" and "computation in service to the sciences." But for our own students, we are more interested in an engineering approach that emphasizes solving problems and building artifacts.
- Our programming courses emphasize "projects." A project is a relatively large programming assignment, done primarily outside of class, where the assignment is focused on a problem in the world rather than a lesson in the class. Students either build the entire project themselves or use a well-designed, general-purpose library like that of Hanson (1996). We work hard to avoid the trap in which an instructor designs the project and students "fill in holes."

Examples of project problems we have used in the first four semesters of instruction include "Identify the natural language of this web page," "Search a database of song lyrics," "Compress and decompress images," and "Implement full integers using machine integers." These projects are connected to real-world technology (Google Translate, Google Search, JPEG, and bignums). We also like projects that are connected to what Tufts calls "active citizenship" (a form of public service), to "big data," and to scientific applications.

- Our department is very concerned to serve and retain members of underrepresented groups, including, e.g., women and first-generation university students. We work hard to avoid creating courses that offer advantages to students who have a special affinity for mathematics or to students who have tinkered with computers for many years. Many of our students have no prior experience, and we want the playing field to be level. For this reason, we minimize the amount of student work devoted to games and puzzles; we feel that students who are drawn to games and puzzles are already overrepresented in computer science. In particular, the video-game examples in *How to Design Programs* would be considered a poor choice for our students.
- Our central administration has articulated that, as a strategic goal, Tufts should provide *transformative* educational experiences. This goal helps support the choice of unusual technologies (such as functional programming) in the classroom. It also means that we can ask students to work hard.

Given these constraints, our department endorsed the creation of a course *in problem-solving by computer, where students will learn to solve problems "starting from a blank page," and that will be available to every student who is motivated to work hard, regardless of background.* We decided that our greatest opportunity for success lay in adapting *Program by Design* for Tufts:

- We liked the tactical advantages of a mature course designed to be adopted, with a substantial supporting infrastructure, including both software and teaching workshops.
- *Program by Design*'s design process was the closest thing we could find to a systematic method of software development suitable for beginning students.

- We liked the thought of the DrRacket programming environment and its language levels, which had been proven to work well with beginning students.

On revisiting our unpublished white paper, which can be found at `http://curriculum.cs.tufts.edu`, I am sobered to see that we understood almost nothing about the method.

Functional programming was never a goal; it was a means to an end. We arrived at functional programming through this reasoning:

- What we want most from our first course is for students to learn systematic methods of problem-solving and software development.
- Such methods are the essence of *Program by Design*.
- *Program by Design* is most easily taught using functional programming.
- Therefore, functional programming is a good choice for the introductory course.

D Trivia and Ephemera

This section information that is too trivial to warrant space in the body of the paper or that is likely to be invalidated by future changes in *How to Design Programs* or in DrRacket. (The course described in this paper was taught using both the first and second editions of *How to Design Programs* and using DrRacket version 5.3.6.)

D.1 Choosing an edition of the textbook

How to Design Programs is available in two editions. The first edition is complete, available in print from MIT Press, and also available online. The second edition is not only still unpublished; it is not yet complete. But it contains many changes and improvements:

- It describes a function's type as its "signature," not its "contract."
- It introduces `check-expect` and related forms.
- It is much more explicit about the difference between designing *functions* and designing *programs*.
- It contains new sections on designing programs in two model families of programs (interactive graphics programs and batch I/O programs).
- It explains the most up-to-date versions of the `image` and `universe` libraries described by Felleisen et al. (2009).

- It contains new material on higher-order functions, the use of abstractions, and effective use of wish lists.
- It contains myriad other small improvements (e.g., explicitly quantified type variables in polymorphic signatures, less promiscuous use of `else`).

While I borrowed a number of ideas and terms from the second edition, and I assigned a few readings from the second edition, I required the first edition as the official textbook for my course. While I find much to admire in the second edition, and I look forward eagerly to its completion, I felt that, especially for a new course, everyone concerned would be uncomfortable with the idea of an incomplete textbook available only on the web, whereas everyone would be very comfortable with a printed book that had received the imprimatur of the MIT Press.

D.2 Complaints about the teaching languages

I have only very minor complaints: I frequently wished for "unequal" functions on various base types, and I wished there were no `struct?` predicate. In a rational design process, I can identify no place for the `struct?` predicate, and my students routinely misused it.

I didn't realize `check-within` could be used on any value—I had thought it was limited to floating-point values. Also, not all my students understood how to use its "epsilon" parameter. These students sometimes wrote `check-within` forms that would pass tests even if a computation was grossly inaccurate.

D.3 Pain points with DrRacket 5.3.6

First, I found it difficult to create libraries ("teachpacks") for my students' use. There are two different, incompatible methods for installing libraries, the simpler of which turns out to be irreversible (there is no "uninstall"). Neither method was easy for me to learn. I believe that these problems can be addressed easily enough by better explaining Racket's "collection" and "package" models, as well as some other minor improvements in the documentation.

The second pain point is a component called the "handin server," which enables to submit programs from their own computers, be they OSX, Windows, or Linux. The details are not worth recounting here, but my systems staff and I spent a couple of full days each getting it (mostly)

working. What you need to know is that the handin server exists, and that getting it working may take much more time than you expect.

A third pain point seems almost silly, but it prevented one student from submitting a working homework assignment, and none of my course staff was able to diagnose the problem. The student had somehow cut and pasted code containing a Unicode en dash or em dash, and had tried to use the dash as a minus sign. To say that the error message was baffling would be rank understatement.

D.4 Importing libraries

I must mention a minor pitfall connected to the `universe` library. Both the second edition `universe` and the first edition `world` build on an `image` library. But while there is only one `universe` library and only one `world` library, there are *two* `image` libraries, and they are incompatible. All libraries are chosen using DrRacket's menus, and unfortunately a number of my student chose the wrong `image` library from the menus. This error took our course staff some time to diagnose, and until we diagnosed it, it caused significant unnecessary suffering and confusion. In the future, as noted in the body of the paper, I will mandate that all libraries be imported *not* using the graphical user interface but using an explicit statement in the source code such as `(require 2htdp/image)`. This procedure will also have the advantage of putting my libraries on the same footing as the book's libraries.

D.5 Two of my minor oversights

70% of my students were freshmen in their first semester at university. Many of them were expecting an "introduction to computer science," and I did not manage their expectations about what "introduction" means to a computer scientist. Many students were shocked to learn that an introductory course would require weekly lab exercises and weekly programming assignments. Most of these students dropped the course.

Students were very curious to know about full Racket and how it is used in real life. I was not prepared to answer.

E Design guidelines for world programs

This section reproduces a handout I created to help my students design world programs.

The second edition says:

> Your task is to develop a data representation for all possible states of the world.

This is the hardest part of designing a world program. Some ideas:

- Draw scenes that the program might display. (Imagine you film the program in operation. What are the key still images you would put on a storyboard to show the program at work?)
- What data do you need to draw your scenes and images?
- What data remains the same in every scene? Things that don't change shouldn't be part of the world state. Such things should instead be made named constants using `define`.
- What data changes? That data becomes part of the data definition for the world state. Here are some general ideas:
  - A very simple program might have just one item of atomic data as its world state.
  - A scene with multiple elements might have a structure as its world state (definition by parts).
  - A program with multiple kinds of scenes, each with its own set of elements, might have a set of variants as its world state (definition by choices *on top of* definition by parts).
  - *Protip*: It is also possible to define the world state using definition by parts on top of definition by choice, or even definition by parts on top of definition by choice on top of definition by parts. This is one of the deepest tricks in the business, and we'll talk about it in class.
- What datum is needed to draw the very first scene? This datum is a world state; in fact, it is the *initial state of the world*. It is sometimes called "world 0."
- A world is full of events such as mouse movement, button presses, key presses, and the passage of time. What events cause changes in scenes? One way to answer this question is with a *state-transition diagram* that

shows a box for each choice of state and and arrows connecting related states.

If you can answer this question, and you can say *how* the events affect the state, then you're ready to finish your data description and move on to your order list.

E.1 Developing an order list for world programs

Questions to ask about any world program:

1. Does it need to draw anything?

   Yes. You will need to define a function to pass to `to-draw`. That function will need a name; in the example below, it is called `render`.
2. Does it need to respond to the mouse?

   If so, you will need to define a function to pass to `on-mouse`. That function will need a name; in the example below, it is called `mouse-event-handler`.
3. Does it need to respond to the keyboard?

   If so, you will need to define a function to pass to `on-key`. That function will need a name; in the example below, it is called `key-stroke-handler`.
4. Do things need to happen as time passes, even if the mouse and keyboard are untouched?

   If so, you will need to define a function to pass to `on-tick`. That function will need a name; in the example below, it is called `clock-tick-handler`.
5. Does the program run indefinitely (or until killed)?

   If not, you will need to define a function to pass to `stop-when`. That function will need a name (which should end in a question mark); in the example below, it is called `end?`.

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

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

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

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

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

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

Figure 1: Signatures of world functions (reproduced, with minor formatting changes, from Figure 10 of *How to Design Programs*, second edition)

The signatures for all these functions are explained in Section 2.6 (Designing World Programs) of the second edition textbook. Figure 1 reproduces the key figure from that section.

Here is an example call to `big-bang`:

```
(big-bang first-world-state
          (on-tick clock-tick-handler)
          (on-key key-stroke-handler)
          (on-mouse mouse-event-handler)
          (to-draw render)
          (stop-when end?)
          ...)
```

Here is another example that uses shorter names for some of your functions:

```
(big-bang w0
          (on-tick tock)
          (on-key ke-h)
          (on-mouse me-h)
          (to-draw render)
          (stop-when end?)
          ...)
```

E.2 Resources

The Racket Documentation is not the best way to learn `big-bang`. The best way is to consult Part One of the second edition:

- Section 2.4.2 (Interactive Programs) presents some simple `big-bang` examples, but it does not explain systematically what `big-bang` does.
- Section 3.6 (Designing World Programs), especially Figure 10 (which is reproduced above), *systematically* outlines the structure of every program that uses `big-bang`, and it explains what `big-bang` does with each piece of this structure.
- Section 3.7 (A Note on Mice and Characters) explains mouse-event handlers and keystroke handlers.