The Structure and Interpretation of the Computer Science Curriculum

Matthias Felleisen[1], Robert B. Findler[2],
Matthew Flatt[3], and Shriram Krishnamurthi[4]

[1] Northeastern University, Boston, MA, USA
[2] University of Chicago, Chicago, IL, USA
[3] University of Utah, Salt Lake City, UT, USA
[4] Brown University, Providence, RI, USA

`http://www.teach-scheme.org/`

Abstract. Nearly twenty years ago, *Structure and Interpretation of Computer Programs* (SICP) changed the intellectual landscape of introductory computing courses. Unfortunately, three problems—its lack of an explicit program design methodology, its reliance on domain knowledge, and the whimsies of Scheme—have made it integrate poorly with the rest of the curriculum and fail to maintain its position in several departments.
In this paper we analyze the structural constraints of the typical computer science curriculum and interpret SICP and Scheme from this perspective. We then discuss how our new book, *How to Design Programs*, overcomes SICP's problems. We hope that this discussion helps instructors understand the structure and interpretation of introductory courses on computer science.

1 SICP Conquers the World

The publication of *The Structure and Interpretation of Computer Programs* (SICP) [1] changed the landscape of the introductory computing curriculum. Originally intended for MIT's introductory course for computer science and electrical engineering majors, use of the book quickly spread around the world. As universities from Japan to Germany adopted the book, they also switched to Scheme and a (mostly) functional approach to programming.

The success of SICP and Scheme is mostly due to its fresh philosophy. The SICP approach liberates the introductory course from the tyranny of syntax. Instead of arranging a course around the syntactic constructs of a language, SICP briefly explains the simple syntax of Scheme and then presents a series of deep concepts in computing and programming: functional programming, higher-order functions, data abstraction, streams, data-directed programming, a simulation of OOP via message passing, logic programming, interpreters, compilers, and register machines.

To this day, SICP has a deserved reputation as one of the great books of computer science. A large number of colleges and universities still put the book on reading lists. Amazon.com's pages contain 85 overwhelmingly positive reviews. Computer scientists invariably recommend it to gifted students. Yet, over the past decade, the use of SICP and Scheme rapidly declined in introductory courses. In the US, many of those who implemented SICP have switched to the currently fashionable object-oriented programming language (C++, Java, C#).

SICP declined in part because the material is difficult and partly out of systemic problems with the approach. Some instructors complained about the electrical engineering bias of the book.[1] Others blamed functional programming, or specifically Scheme: Wadler's critique [9] exposed genuine problems with the language. Nowadays, the critics even include MIT professors with experience in teaching SICP. Jackson and Chapin, both at MIT, recently wrote that

> [f]rom an educational point of view, our experience suggests that undergraduate computer science courses should emphasize basic notions of modularity, specification, and data abstraction, and should not let these be displaced by more advanced topics, such as design patterns, object-oriented methods, concurrency, functional languages, and so on [5].

Although this quote doesn't mention SICP, the two are referring to their courses at MIT, where they encounter the "products" of the SICP course.

Advocates of Scheme and functional programming should be concerned by these reactions. Clearly, SICP, Scheme, and functional programming fail to meet some basic needs in the introductory curriculum. We shouldn't just brush off these criticisms but study them, determine their true nature, and react to them.

In this paper, we present our own personal reaction. More specifically, we present the design rationale behind our new book *How to Design Programs* (HTDP). In section 2, we analyze the structure of a conventional computing curriculum and the constraints it imposes on the first course. In section 3, we remind readers why Scheme is a good starting point for a first language, given a good programming environment. Section 4 presents our interpretation of SICP and explains how HTDP improves on SICP.

We compare HTDP to SICP rather than to the current situation for two reasons. First, we believe that SICP is currently the strongest approach to an introductory course on programming and computing, even if it is no longer in vogue. Second, SICP made functional programming widely visible. We believe that it is unfortunate how its flaws overshadowed this movement to functional programming.

[1] This criticism is, of course, unfair, considering that the book was intended for electrical engineering majors.

2 Structure

2.1 Contextual Constraints

An introductory computer science course must satisfy a set of complex contextual constraints. Faculty, students, and even parents have certain expectations. The instructor of the course can assume only so many things from their freshmen. The instructor of the second course rely on a certain set of skills. Since an effective curriculum designer must accommodate—or at least respond to—all these constituents, it is necessary to understand their expectations.

Faculty colleagues (inside and outside of computer science) often have an emotional preference for a specific language in the introductory course. To some, the first language is the one that they know and work(ed) with. To others, it is the currently fashionable industry language, e.g., C++ and Java over the past ten years.

Some computer science faculty also demand that the first course teach languages that are used in upstream courses. Sometimes they believe that the first course should teach the same language as the second course. Sometimes they wish to expose students to languages that are used in popular upstream courses such as operating systems or graphics.

Freshmen also come with strong, preconceived notions about programming and computing. Some have read about the latest industry trends in popular magazines, such as (in the US) *Time*, *Newsweek* and *US News and World Report*, and expect to see some of these things in a freshman course. Some base their understanding on prior experiences in high schools. They are used to sophisticated GUI-based program development environments (IDEs) that include mechanical support for syntactic conventions and project construction.

At the same time, freshmen take the first course for many different reasons and with vastly different levels of mathematical background. Some students wish to find out what computer science is all about. Others want to learn how to construct games. Some know and understand calculus; for others, algebra is a minefield.

In the end, the instructor of the second course should not have to start from scratch. The simplest interface between the first and second course is the syntax of the a shared programming language and a rudimentary understanding of the underlying model of computation. For that reason, many colleges teach the same programming language in the first two courses and often just allocate certain language topics to one or the other course. While this solution is easy to implement, it is ad hoc and lacks a rational goal structure. We argue in the next section that computer science departments can do better than that.

2.2 Goal Structure

A curriculum designer must accommodate all these constraints and yet not lose sight of the overall goal of a university education. We believe that

> *the university's goal is to produce effective software developers who can quickly adapt to current practice and who can survive in a software-related profession for decades.*

After all, most computer science students accept positions as software developers after they graduate (as opposed to proceeding to graduate study). Many will be involved in the software industry for years to come. Those who become managers, especially, should be able to understand and evaluate new insights about programming and computing that their curricula cannot envision.

Hence, a computer science curriculum must acknowledge the realities of the current software world without becoming a vocational training ground. This suggests an education that integrates university training with real-world professional experience in industry, as it exists in a de facto manner in many places. If we accept this position, two points in the curriculum take on special meaning: the academic term before students work as interns for the first time, usually their first summer, and their last academic year, when they prepare themselves for their first full-time positions.

Based on this reasoning, we suggest that

> *a university curriculum concentrate on principles for most of the time and accommodate industrial needs during the second semester of the first year and the last year of the program.*

After all, college is the only time in a programmer's life when he is exposed to principled ideas on a regular and rigorous basis. Once a programmer has a full-time position, there are too many constraints and distraction for additional courses on principles. Thus, in analogy to the software development process, the training process should debug people's programming habits as early as possible. The later they detect flaws, the more the "bugs" become ingrained and the more costly the fixes will be.[2]

Our suggestion has an obvious corollary for the first-year courses.

> *The first year should start by emphasizing principles and should add some industrially relevant concepts during the second semester.*

[2] Ideally, software professionals should continue to educate themselves like medical professionals. But neither industry nor universities encourage this form of continuing education in a serious manner.

Even more precisely, the first semester should emphasize programming principles and habits; the second part should illustrate the use of these principles in currently fashionable programming languages (since many students will want to seek summer positions). Of course, the "principled" semester may integrate fashionable parts as long as they don't obscure the principles, and, more importantly, the "fashionable" part of the curriculum must continue to practice good design habits.

Considering the above we propose the following division for the first year. The first semester should teach a high-level model of computation and robust, portable programming habits, i.e., habits that should apply in a spectrum of programming languages (functional, OO, scripting languages). To accommodate industrial needs, the second semester should place these principles of computing and programming into the context of a currently fashionable industrial programming language.

2.3 Programming Principles for the First Year

The real challenge is thus to identify good programming principles and habits. Based on our experience with first-year students and first-year courses, we propose the following (minimal) candidates:

1. Students must learn to read problem statements carefully and extract useful pieces of information:
   (a) a concise statement of purpose for the program and each function or method;
   (b) a description of the classes of problem data;
   (c) a set of examples that illustrate both the data and the purpose statements.
2. Students must learn to use this information to organize programs. In a conventional OOP context, program organization means class hierarchy. More precisely, the student should learn to translate the description of classes and the purpose statements into class hierarchies and methods. In an FP context, program organization means type and function definitions. That is, a program consists of type definitions and function definitions that reflect the structure of the type definition. In either case, the students must also learn to compose functions and methods.

   Matching the problem analysis and the program organization in this manner has several advantages. First, it provides guidance to students who might otherwise see programming as a black hole. Second, it helps instructors evaluate a student's progress. Third, if students learn to organize programs in this manner, they quickly see how to connect changes to the problem statement to changes in the program. That is, they learn that program organization matters for program maintenance, one of the most important parts of their future jobs.

3. Students must learn to test their programs. The key is that students learn to make up examples *before* they write down code. Above anything else, it forces them to think through the workings of the function.[3] Next, if they design tests after programming, their implementation reasoning will affect testing. For example, they may miss some basic requirements. Finally, if they don't thing about expected results first, they may just be happy if their program outputs something that looks close to the desired result.

   Furthermore, students must learn that tests should become a part of the program. In other words, a core function should *not* print results—chars, strings, lists, vectors, objects—but hand them over to a tester function for checking. This principle naturally forces students to separate between *model* and *view* functions, i.e., they implicitly learn to use a model-view-control architecture [4].

Clearly, these general principles don't inherently favor one language over the other or one model of computation over another. They apply to almost all languages, especially ones in which it is relatively convenient to describe a rich set of values with relatively little effort. Hence, we must ask in which context these principles are easiest to teach.

3 The Role of Scheme

We claim that Scheme, suitably supported, is near-ideal for teaching the principles of programming and computing. It is a lightweight language, provides an interactive mode for exploration, and has a simple semantic model. The arguments in support of Scheme have been told time and again. We summarize them here briefly and add comments where we believe the Scheme community has been overly simplistic and should work on further improvements:

Scheme's syntax is simple. Turning virtue into vice, Scheme's syntax is *too* simple. Too many of a beginner's S-expressions are legal Scheme expressions. Our prior work explains this problem and solves it by defining Scheme as the limit of a series of teaching subsets in DrScheme [3]. Each subset is strictly enforced and produces error message that use only concepts from the learner's corresponding knowledge level.

[3] This is also the closest we can approach the preferable practice of separating developers and testers entirely.

Scheme's semantics is easy to understand. SICP can quickly move from syntax to computer science concepts because it uses a language subset with a straightforward substitution semantics. Loosely speaking, the language is a generalization of high school algebra. A Scheme implementation must then provide a stepper that illustrates this concept, so students can easily understand a program evaluation without understanding physical notions like registers, environments, pointers, stacks, etc.[4] Figure 1 illustrates how DrScheme's stepper [2] presents a step in the reduction sequence of a beginner program.

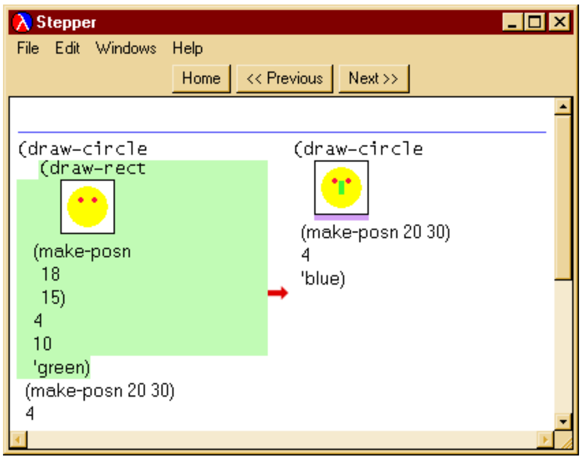

Fig. 1. Stepping through Scheme programs

Scheme is safe. Scheme's standard [6] permits implementations with fully predictable positive and negative behavior. DrScheme is such an implementation. All primitive operations totally specify the classes of values with which they can cope. When a primitive operation violates these stated invariants, DrScheme raises an exception and high-lights the offending expression. For beginners, detecting and pinpointing the source of run-time exceptions are critical elements of the learning process.[5]

[4] It may still be valuable to teach these concepts later in the course, when students have absorbed the basic ideas of program construction.

[5] This partly explains why C++ is such a pedagogic failure. Its type system notwithstanding, the lack of safety means it does not even guarantee that a printed number corresponds to a number in memory; the bits can indeed come from anywhere in the store. Similarly, core dumps and bus errors are much worse than exceptions, because they typically happen long after the first violation occurred.

Scheme is dynamically typed. Although a static typing discipline, especially akin to those of Haskell or ML, is highly useful for programming in general, it introduces two complication for introductory courses. First, beginners, who are often incapable of distinguishing between the computer, the programming environment, and the run-time context now must understand that there is yet another layer. Implementors must deal with types the way we have dealt with syntax, if they wish to ensure that type error messages reflect the knowledge level of the learner. Second, different type systems capture different classes of invariants. Hence, it is difficult (though not impossible) to mold a statically typed introductory language to the needs of the second course. In Scheme, instructors superimpose their own (unchecked) type discipline, thus anticipating the type discipline of the second semester.

Unfortunately, the choice of language for a freshman course must take into account some of the emotional judgments that students bring into the course. One of them is the idea that people program in a fancy GUI environment. Another one is that a language is useful, which typically means that it comes with support for GUIs and GUI construction, Web scripting, database connections, networking, regular expressions searches, and so on. Fortunately, the Scheme community has developed just such environments [3, 7, 8] over the past ten years so that there is no need to compromise along those lines.

4 Interpretation

Equipped with an analysis of the constraints and a goal structure for the first course, we can now interpret SICP in this context and explain its problems. This explanation motivates the HTDP approach to teaching the principles of programming and computing.

4.1 *Structure and Interpretation of Computer Programs*

SICP's coverage of topics in programming and computing sets a new standard for introductory computer science texts. Even a partial list of SICP topics makes this point: first-order and higher-order functional programming, abstracting with data abstraction, objects and assignment, streams, modularity, logic programming, meta-linguistic abstraction, compilation and register machines. Also, the chapters on functional programming and functional programming with assignment introduce interpreters for those portions of Scheme and thus introduce comprehensive models of computation. The portion on register machines refines these models to the lowest point of abstraction for computer scientists.

At first glance, SICP covers important topics with regard to our stated goal of teaching program design. After all, functional (procedural) programming, data abstraction, and programming with assignment are the fundamental building blocks of large programs. Instead of looping constructs, SICP uses tail recursive functions and explains how tail recursive functions behave like ordinary loops. These first appearances are deceiving, however. SICP does not *teach* program design; it merely illustrates programming with a long series of examples.

More specifically, SICP teaches program design *implicitly*. It presents the various uses and roles of program constructions with a series of examples. Then some exercises ask students to modify this code basis, requiring students to read and study code. Others ask them to solve problems that are similar to the ones just covered. This means students have to study the underlying construction and simulate it as much as possible. In other words, while SICP shows *that* programs benefit from organizing them as a collection of procedures, it does not discuss *how* programmers determine which procedures are needed or *how* to organize these procedures. While it explains *that* programs benefit from functions as first-class values, it does not show *how* programmers discover the need for this power. While SICP introduces the idea *that* programs should use abstraction layers, it never mentions *how* or *when* programmers should introduce such layers of abstraction. Finally, while the book discusses the pros and cons of stateful modularity versus stream-based modularity, it does so without explaining *how* to recognize situations in which one is more useful than the other.

We consider this lack of program design knowledge in SICP the major flaw of the book. If teachers don't state such principles *explicitly*,[6] the first programming course turns into a course on syntax, no matter how simple the underlying programming language. In our experience, *implicit* learning does not work well for the majority of students. Most students are not the type of learner who can extract abstract principles on program design from a series of examples. As a matter of fact, across the spectrum of universities that the authors represent, most students simply are not prepared to abstract *anything* from a series of exercises. Instead students focus on the surface level of knowledge in such a course, which is the syntax of the programming language. Worse, they think that this focus on syntax is natural and accept it as a key insight for future learning efforts concerning computing and programming.

SICP's second major problem concerns its selection of examples and exercises. All of these use complex domain knowledge. Consider the left column in

[6] The reader should not confuse the distinction between *explicit* and *implicit* with the one between *active* and *passive* learning. Using explicit statements about design knowledge does not mean students should not "learn by doing." It is still necessary for students to practice, fail, learn from mistakes, and study feedback from computers and teachers.

figure 3. It presents the choice of major examples that are used in the first few chapters of SICP. The last two cover topics from computer science: laziness, non-determinism, logic programming, register machines, and compilers. That is, they are meta-topics.

These topics are interesting to students who use computing in electrical engineering and the sciences and to students who already have a perspective of computing. In general, however, using such topics for examples and exercises assumes far too much domain knowledge of the ordinary beginning student. On the average, beginners are not interested in mathematics and electrical engineering, and they do not have the domain knowledge at their finger tips for solving the domain problems. As a result, students must spend a considerable effort on the domain knowledge and often end up confusing domain knowledge and program design knowledge. They may even come to the conclusion that programming is a shallow activity and that what truly matters is an understanding of domain knowledge.[7]

In summary, SICP did an excellent job shifting the focus of the first course. Unfortunately, it failed to teach explicitly *how* to design functions, *how* to structure programs, and *why* such organizations work well. It also overemphasized domain knowledge over program design knowledge. It was this insight that motivated our work on *How to Design Programs*.

4.2 *How to Design Programs*

Our book improves on SICP in four ways. First, the book discusses explicitly how programs are constructed—and not just in a functional context. Second, it uses more accessible forms of domain knowledge than SICP and deemphasizes domain knowledge in exercises that focus on program construction. Third, the book tames the syntactic obscurity of Scheme with a series of well-defined language subsets. Because of this shift in emphasis, we gave our book the title *How to Design Programs—An Introduction to the Principles of Programming and Computing* (HTDP).

Design Knowledge Even a cursory look at HTDP's table of contents reveals the emphasis on program design. Every chapter comes with at least one section on the design of a particular class of functions. The title of no section concerns domain knowledge, except for those labeled "extended exercise."

[7] Some faculty members argue that a course on introductory programming is a good place for teaching students mathematical problem solving, which most of them never understood in primary school. While we partly agree with the idea that programming can teach domain knowledge, we also believe that a course on *programming* should teach knowledge about program design. We therefore ignore this line of argument here.

Data Definition:

```
;; MockXml is
;; — either a String
;; — or a (cons MockXml (cons MockXml empty))
```

Contract, Purpose Statement:

```
;; MockXml → Number
;; to count the number of chars in an-xml
;; (define (size a-xml) ...)
```

MockXml Examples:

```
;; "Hello World"
;; (cons "This is my first paragraph." (cons "Help!" empty))
```

Examples for *size*:

```
;; "Hello World" should produce 11
;; (cons "This is my first paragraph." (cons "Help!" empty)) should produce 32
```

Template:

```
#|
(define (size a-xml)
  (cond
    [(string? a-xml) ... ]
    [else ... (size (first a-xml)) ... (size (second a-xml)) ... ]))
|#
```

Definition:

```
(define (size a-xml)
  (cond
    [(string? a-xml) (string-length a-xml)]
    [else (+ (size (first a-xml)) (size (second a-xml)))]))
```

Tests:

```
(= 11 (size "Hello World"))
(= 32 (size (cons "This is my first paragraph." (cons "Help!" empty))))
```

Fig. 2. A sample Scheme program

The sections on program design present design knowledge in the form of a *design recipe*. Every design recipe enforces basic habits that generalize to all kinds of problem solving situations. Most of them have the following structure:

1. analyze the class of problem data and describe the class;
2. formulate a concise purpose statement (and a type signature);
3. illustrate the data definitions and the purpose statement with examples;
4. create a function layout based on this information;
5. write code;
6. and then turn the examples into (automatic) test cases.

The design recipes mostly differ on how they relate the function organization (step 4) to the data description (step 1).

The first half of the design recipes show how the description of the class of problem data directly suggests an organization of a function that processes this class of data. This series of recipes addresses the design of functions for classes of atomic data (numbers, booleans), intervals and unions, composites, self-referential definitions, groups of mutually referential definitions, and so on. The second half of the design recipes cover other important topics: abstracting over similar functions and data definitions, generative recursion, accumulator-style programming, and programming with mutation. In these cases, the design recipes include a discussion on when to use a technique. That is, no piece of design knowledge is introduced as just another trick in the toolbox.

Figure 2 illustrates the design recipe for data descriptions that involve recursive unions. The data definition involves basic Scheme classes (strings, e.g., `"hello"`, with basic functions, e.g., string-length) and a "union," that is, an enumeration of (disjoint) alternatives. The goal of the exercise is to develop the function *size*, which counts the number of characters in the strings within an element of *MockXml*.

Given a data definition with two clauses, the function consists of a **cond** expression with two clauses: one for strings and one for `cons`es. Since strings are atomic for the purpose of our exercise, there is no other information in the first clause than *an-xml* (the argument). For the second clause though, we know from the data definition that *an-xml* contains two pieces, which the function can access via `first` and `second`. Because both pieces are elements of *MockXml*, the function template refers to itself for those two expressions. Putting all these insights together produces the function template.

The template deserves some explanation. Its purpose is to express a data definition as code. To construct it, students answer the following questions:

1. Is the data definition an enumeration of alternatives? Use a **cond**.
2. How many branches are in the enumeration? Add that many clauses.

3. Which predicate describes each branch? Write down the predicates.
4. Which of these predicates test for compound values? Write down the selectors in the answer part.
5. If any selection expression produces a value in some other defined set of values, add an appropriate function application.

The other sections are self-explanatory. The test step deserves some additional explanation. The eventual goal is to produce expressions that automatically check whether the function produces the desired result for some given input. It is important to get students used to the idea that this step is automated as much as possible. On the one hand, automation requires that functions (or programs) not just write some silly answer to the output stream. On the other hand, automation inspires a thorough discussion and understanding of equality, a topic that belongs to the very core of computing (and philosophy).

Scaling Design Knowledge The recipes also introduce a new distinction into program design: *structural versus generative recursion*. The structural, or data-based, design recipes in the first half of the book match the structure of a function to the structure of a data definition. When the data definition happens to be self-referential, the function is recursive; when there is a group of definitions with mutual cross-references, there is a group of function definitions with mutual references among the functions. In contrast, generative recursion concerns the generation of new problem data in the middle of the problem solving process and the re-use of the problem solving method.

Compare *insert* and *kwik*, two standard sort functions:

```
;; (listof X) → (listof X)
(define (insert l)
  (cond
    [(empty? l) empty]
    [else
      (place
        (first l)
        (insert (rest l)))]))
```

```
;; (listof X) → (listof X)
(define (kwik l)
  (cond
    [(empty? l) empty]
    [else
      (append (kwik (larger (first l) l))
        (first l)
        (kwik (smaller (first l) l)))]))
```

The first function, *insert*, recurs on a structural portion of the given data, namely, (rest *l*). The second function, *kwik*, recurs on data that is generated by some other functions. To design a structurally recursive function is usually a straightforward process. To design a generative recursive function, however, almost always requires some *ad hoc* insight into the process. Often this insight is derived from some mathematical idea. In addition, while structurally recursive functions naturally terminate for all inputs, a generative recursive function may diverge.

HTDP therefore suggests to add a discussion about the termination condition to the implementation of a generative recursive function.

Distinguishing the two forms of recursion is important for connecting a course that uses a functional language to a course that uses an OO language. In an OO world, the structural recipes naturally lead to class hierarchies and methods that call directly along containment ("has a") relationships. Indeed, an OO purist might argue that OO programming languages arise from implementing structural recipes as a linguistic construct, i.e., triggering all computations via method calls and nothing else.

Contrast this treatment of recursion with SICP's. The two notions are not even distinguished in SICP. Worse, the book's first recursive procedure (*sqrt-iter* on page 23) uses generative recursion. The structural aspect of recursion is ignored. SICP thus misses structural recursion and structural reasoning. As a result, the book never actually discusses reasoning about, and programming with, classes of data, which is the essence of modern OO programming.

In lieu of programming with classes, SICP shows how to implement a message passing protocol among functions. While such a protocol brings across a (poor) simulation of object-oriented computation, it does not teach students how to organize classes, how to assign methods to classes, and how to connect them. In a sense, it confuses implementing objects with object-oriented programming.

Domain Knowledge Concerning the choice of examples and domain knowledge HTDP distinguishes itself from SICP in many ways. Figure 3 juxtaposes the section titles in SICP and HTDP that are concerned with exercises. As a supplement to the book, we are also developing a series of exercises on managing both static and dynamic Web sites. Even a short glance shows that HTDP uses domain knowledge that is within reach of most students. It does offers some exercise sets that introduce mathematics that may be new to some students (such as Gaussian elimination and adaptive integration), but such exercises are never on the critical path through the book.

Teaching Languages SICP and HTDP also vastly differ in their treatment of Scheme syntax. SICP introduces the full functional Scheme subset in the first few sections. HTDP uses a well-defined series of language subsets. The first subset consists of structure and first-order function definitions; the second adds list abbreviations to make working with lists more manageable. The third and fourth subset enriches the functional world with first-class, higher-order functions. The fifth language subset adds assignment. In turn, DrScheme enforces these subsets.

SICP:	HTDP:
primality	moving circles
interval arithmetic	hangman
symbolic differentiation	moving shapes
representing sets	moving pictures
huffman encoding trees	rearranging words
symbolic algebra	binary search trees
digital circuits	evaluating scheme
	more on web pages
	evaluating scheme again
	moving pictures, again
	mathematical examples
	Gaussian elimination
	checking (on) queens
	accumulators on trees
	missionaries and cannibals
	board solitaire
	exploring places
	moving pictures, a last time

Fig. 3. SICP and HTDP exercises

Based on our experience with HTDP and DrScheme, we believe that the restricted languages help students in two ways. They force students to focus on design principles rather than the random use of features they may not understand; in turn, students understand syntactic and semantic error messages much better because the explanations are restricted to the student's ken. Since beginners (and even experienced programmers) make mistakes and need feedback about mistakes, this is an important improvement over traditional approaches.

The second advantage of HTDP's approach concerns the transition to an OO course. Using a functional Scheme subset in conjunction with structural design recipes means that students work with classes of data and hierarchies of classes long before they encounter their first assignment statement. They thus learn to appreciate how important it is to reason about classes and functions on classes long before they study the encapsulation of state inside of objects.

The design recipes offer an invaluable means for evaluating student understanding, leading to a grading rubric. We find that though students may approach instructors with questions about bugs in their programs, the flaw often resides much earlier in the recipe's steps: for instance, they may have at best a shaky understanding of their data. Eliciting their response to each step of the recipe helps us differentiate superficial mistakes from fundamental ones. Fixing the problem early in the recipe invariably leads to much better final solutions. Finally, by evaluating all their steps, not just their final program, we can assign credit much more fairly and accurately. In effect, the steps of the recipe help the student create a structured portfolio of their work on the problem.

5 Conclusion

SICP had a deep impact on the introductory curriculum. It especially put functional programming in Scheme on the map for people who had never seen an alternative to procedural programming. Unfortunately, teaching SICP is more difficult than teaching a conventional programming course in the currently fashionable syntax. Worse, teaching SICP does not build a bridge to the OO courses of the second semester. Hence, many first-semester instructors have given up on functional programming.

HTDP and DrScheme are attempts to rectify this situation by learning lessons from the SICP experience. We preserve the value of functional programming as a pedagogic foundation, but temper it with the constraints of the rest of the curriculum. The result is more than just SICP with a GUI-based environment or "Scheme with pretty pictures." It is instead a synthesis of these approaches, combined with a solid foundation of program design principles.

Acknowledgment: The authors thank the members of PLT at their respective institutions, and their inimitable users, for their contributions over the years.

References

1. Abelson, H., G. J. Sussman and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.
2. Clements, J., M. Flatt and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, 2001.
3. Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pages 369–388.
4. Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
5. Jackson, D. and J. Chapin. Redesigning air traffic control: An exercise in software design. *IEEE Software*, 17(3), 2000.
6. Kelsey, R., W. Clinger and J. R. (Editors). Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
7. Schemer's Inc. EdScheme: A Modern Lisp. ISBN: 0-9628745-8-2, 1991.
8. Serrano, M. Bee: an integrated development environment for the Scheme programming language. *Journal of Functional Programming*, 10(2):1–43, May 2000.
9. Wadler, P. A critique of Abelson and Sussman, or, why calculating is better than scheming. *SIGPLAN Notices*, 22(3), March 1987.