EDUCATIONAL PEARL

The Structure and Interpretation of the Computer Science Curriculum

Matthias Felleisen, Northeastern University, Boston

Robert Bruce Findler, University of Chicago, Chicago

Matthew Flatt, University of Utah, Salt Lake City

Shriram Krishnamurthi, Brown University, Providence

Email: {matthias,robby,mflatt,shriram}@plt-scheme.org

Abstract

Twenty years ago Abelson and Sussman's *Structure and Interpretation of Computer Programs* radically changed the intellectual landscape of introductory computing courses. Instead of teaching some currently fashionable programming language, it employed Scheme and functional programming to teach important ideas. Introductory courses based on the book showed up around the world and made Scheme and functional programming popular. Unfortunately, these courses quickly disappeared again due to shortcomings of the book and the whimsies of Scheme. Worse, the experiment left people with a bad impression of Scheme and functional programming in general.

In this pearl, we propose an alternative role for functional programming in the first-year curriculum. Specifically, we present a framework for discussing the first-year curriculum and, based on it, the design rationale for our book and course, dubbed *How to Design Programs*. The approach emphasizes the systematic design of programs. Experience shows that it works extremely well as a preparation for a course on object-oriented programming.

1 History and critique

The publication of Abelson and Sussman's *Structure and Interpretation of Computer Programs* (SICP) (Abelson *et al.*, 1985) revolutionized the landscape of the introductory computing curriculum in the 1980s. Most importantly, the book liberated the introductory course from the tyranny of syntax. Instead of arranging a course around the syntax of a currently fashionable programming language, SICP focused the first course on the study of important ideas in computing: functional abstraction, data abstraction, streams, data-directed programming, implementation of message-passing objects, interpreters, compilers, and register machines.

Over a short period, many universities in the US and around the world switched their first course to SICP and Scheme. The book became a major bestseller for MIT Press.[1] Along with SICP, the Scheme programming language (Sussman & Steele Jr.,

[1] According to Bob Prior (editor at MIT Press), SICP sold 45,000 copies in its first five years [personal communication, 9 June 2003].

1975; Steele Jr. & Sussman, 1978; Clinger, 1985; Clinger & Rees, 1991; Kelsey *et al.*, 1998) became widely used. It was no longer the subject of a few individual courses at Indiana University, MIT, and Yale, but the language of choice in introductory courses all over the world.

Unfortunately, the use of Scheme and SICP quickly dwindled again in the early 1990s. After working with SICP and Scheme for a while, instructors started to complain. Some said that SICP's content was too difficult for students outside of MIT. Others blamed Scheme directly, claiming that functional programming in Scheme was too different from programming in other languages. Even the functional programming community criticized the SICP approach; around this time, Wadler wrote his *Critique* of SICP and Scheme (Wadler, 1987).

Nowadays the critics even include professors at MIT, where the book and the course have become legends. Jackson and Chapin, who both have significant experience teaching SICP 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 (Jackson & Chapin, 2000).

In short, SICP, Scheme, and functional programming don't prepare students properly for other programming courses and thus fail to meet a basic need.

Advocates of Scheme and functional programming alike must be concerned about these reactions. To address them and to overcome the problems of the SICP approach, we present this pearl. It consists of three pieces: a structural framework for analyzing the first-year computing curriculum; an interpretation of SICP with respect to this framework;[2] and our alternative to the SICP approach that overcomes SICP's problems while retaining the essence of Scheme and functional programming.

2 Structure

2.1 Solving constraints

The primary goal of a computing curriculum is to produce programmers and software engineers. After all, most of its graduates accept industry positions and produce software. Many will stay involved with software production for a long times, even if only as managers, and therefore also need to learn to adapt to the ever-evolving nature of the field.

Translating the primary goal into a set of goals for the introductory curriculum is a difficult task because various groups impose a range of unrelated constraints. 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

[2] We chose SICP as our yard stick because it is the most widely used and known text that uses functional programming and because we believe that all other texts—of almost equal age (Bird & Wadler, 1988) or of recent vintage (Hudak, 2000)—on functional programming suffer from similar flaws.

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 demand that the first course teach languages that are used in upstream courses. Sometimes they believe that the instructor of the second course should not have to start from scratch and that the simplest solution is to use a single programming language. Sometimes they wish to expose students to languages that are used in popular upstream courses such as operating systems.

First-year students also come with strong, preconceived notions about programming and computing. Some students (or their parents) 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. The latter group is used to sophisticated development environments (IDEs) that include mechanical support for syntactic conventions, GUI development, etc.

The state of the first-year students' education adds another set of constraints to the mix. Some understand calculus; for others, even rudimentary algebra is a minefield.

Finally, students also have a wide range of expectations. Some students wish to learn what computer science is about; others have three years of programming experience. Some wish to know why things work; others want to learn how to construct games. Almost everyone expects that the college training will help them find internships and professional positions.

Satisfying the primary goal of producing software professionals subject to these constraints poses a complex problem. On one hand, learning to program well requires a lot of practice and in particular a lot of hands-on practice. Hence, early courses must introduce programming and must choose a specific programming language. On the other hand, choosing one language over another must disappoint some constituents, and we must therefore convey to them our choices with good reasons. After all, education is as much about satisfying human needs as it is about technical correctness.

We propose to solve this constraint problem with a second look at the primary goal and the timing constraints. Clearly, a computer science curriculum must not, and doesn't have to, become a vocational training ground for the latest industrial programming language and programming tools. Superficial aspects of industrial practice change as fast as fashion trends. No academic department can switch its course content fast enough and maintain a curriculum that passes on tested wisdom. Still, when students cross over from academia into industry, they must be prepared to program and ideally to program well. From this perspective, two points in the curriculum take on special meaning: the first summer, when students work in internship positions, and the last year, when students interview for their first full-time positions.

Following this reasoning, we believe it is natural to concentrate on principles for most of the time and to accommodate industrial needs during the second semester of the first year and the last year of a college program. Considering that college is the only time in a programmer's life when he is exposed to principled ideas on a regular and rigorous basis, the idea of emphasizing principles in college is obvious. Once a programmer has a full-time position, there are too many constraints and distraction for principled additional education. At the same time, however, a curriculum must also teach how these principles apply to the real world. Nobody can expect students to take this step on their own. In short, teach good habits early; otherwise bad habits become ingrained and require costly fixes—just like bugs in programs.

Applied to the first-year courses, these suggestions say that the year should start with a heavy emphasis on principles and should add some industrially relevant components during the second semester. 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. Of course, the "principled" semester may integrate fashionable parts where they aren't an obstacle, and, more importantly, the "fashionable" part of the first year must continue to practice good design habits.

2.2 Principles of programming

The first challenge is thus to identify technical principles for the first-year programming courses. Clearly, we should teach good program design habits (not just syntax and programming style). Based on our experience, we have identified the following set of program design ideas that a first course should translate into habits:

1. Students must learn to read problem statements carefully, extract information, and rewrite it into useful pieces:
   (a) a concise purpose statement for the program and each of its major pieces;
   (b) a description of the classes of data that play a role;
   (c) a collection of examples that illustrate both the classes as well as the purpose statements.

   Ideally the latter should (eventually) make up a rigorous test suite for the program and its functions.
2. Students must learn to organize programs so that they match the class descriptions of item 1b. For example, a functional programmer must define datatypes and functions on these types whose structure matches the type; an object-oriented programmer must define class hierarchies and appropriately distributed methods.

   If students learn to organize programs in this manner, they quickly learn that small changes to the problem statement translate into small changes in the program's code. Considering the rapid changes in the requirements for real-world software, we consider this principle central to our effort.
3. Students must learn to use the examples developed in item 1c above. They must learn to calculate through examples before they code. They must learn to translate the examples into automatic test suites, so that they can test programs as they create them and as the programs evolve later.

More concisely, students must learn that programming requires far more than writing down code and running it on some haphazardly chosen examples afterwards.

The last point in particular suggests that functional languages with their natural model-view separation are superior choices for this first year. When students write automatic test suites, they must to split a program into a part that deals with computation proper (the "model") and another part that interacts with the user (the "view"). They then use the model in two distinct contexts: with a test suite and with the view. In order to re-use the model in a test suite context, they don't want to print results but hand them over directly to a comparison function. Put differently, teaching good software architecture principles to beginners requires function composition and discourages a programming style that is primarily about reading and printing values.

2.3 Principles of teaching

The second challenge for a first-year instructor is to understand the teaching priorities concerning the first language and the first course. Currently, most instructors teach programming with examples. In a typical week, they introduce a new (control) construct, explain with a few examples how to use it, and then assign some exercises from a text book. Students copy the examples and modify them to fit the homework exercises. Since these exercises tend to change the context for the new construct, students also begin to appreciate its general powers and pitfalls. Put differently, the teaching of (control) constructs is *explicit* while the teaching of design principles remains *implicit*; instructors leave it to the students to discover how to go from a blank screen to a full-fledged program.[3]

We believe that the conventional approach to teaching programming reverses the natural roles of data and control. Recall Brooks slogan page 102

> Show me your [code] and conceal your [data structures], and I shall continue to be mystified. Show me your [data structures], and I won't usually need your [code]; it'll be obvious.

as paraphrased by Raymond (Raymond, 1998). When we reason about a program, we want to know the format of the data that it uses, and we can almost imagine how it works. In analogy, when we teach how to program, we should let data drive the syllabus. First we show how to design a program that works on simple data and what kind of (control) constructs this requires. Then we increase the complexity of the data and show how to design programs for these classes of data. Such a step may, or may not, require new constructs, but in the end it forces students to understand how to go from data to design *explicitly*, and they will pick up language constructs implicitly.

Since most students are active learners, it is important to retain the example-driven strategy that is currently used. The examples must, however, focus on the

[3] Challenging instructors throw in ideas from data structures and algorithms or, worse, pose problems that require significant domain knowledge, that is, knowledge about non-computing topics. The problem is then that students tend to confuse algorithms and application domain knowledge with program construction, and neither helps students come up with good program organizations on their own when they are left to their own devices.

use of program design principles in new situations instead of the use of language constructs.

In summary, the first course should introduce the principles of program design, state them explicitly as habits, and have students practice them with numerous examples. To avoid any confusion, the course should not pose problems from complex application domains and it should not use a complex language that distracts from the design principles.

3 Interpretation: functional versus object-oriented programming

Now that we have discussed the structure of the first-year curriculum and its teaching methods, we can turn to the choice of programming language. If we accept the premise that first-year students should learn to use two programming languages, we now face the question which (kind of) languages we should choose. If we also accept the premise that the first language should facilitate the teaching of design principles, choosing a simple functional language for the first course is natural. The second course can then use a (subset of a) complex, industrially fashionable language, such as C# or Java, and show how the design principles apply there.

We justify this suggestion in more detail in the first subsection and explain our concrete choice in the second one.

3.1 Functional and object-oriented programming

Functional and object-oriented programming share the desired curricular focus on data as the starting point for program design. A functional programmer begins with the definition of types and then defines functions on these types. An object-oriented programmer defines classes and adds methods to these classes. Once the vocabulary of data and operations are defined, programs are usually just a function or a method call.

Functional programming and object-oriented programming differ with respect to the syntax and semantics of the underlying languages. The core of a functional language is small. All a beginning programmer needs are function definition, function application, variables, constants, a conditional form, and possibly a construct for defining algebraic types. In contrast, using an object-oriented language for the same purposes requires classes, fields, methods, inheritance in addition to everything that a functional language needs. Furthermore, the computational model of a functional language is a minor extension of that of secondary school algebra. The model of object-oriented computation requires far more sophistication, especially its focus on method dispatch (instead of conditional reasoning) and early state modification.

Using a functional language followed by object-oriented language is thus the natural choice. The functional language allows students to gain confidence with program design principles. They learn to think about values and operations on values. They can easily comprehend how the functions and operations work with values. Better still, they can use the same rules to figure out why a program produces the wrong values, which it often will. Teaching an object-oriented language in the second course is then a small shift of focus. It requires instructors to spend more time on the syntactic complexities of the language, yet they can still rely on, and reinforce, the design principles of the first course. In particular, the switch is of a mostly syntactic nature, because the focus on designing classes of data and operations on these classes remains the same.

3.2 The role of Scheme

Given this context, we picked Scheme as the most suitable starting point for the first language. The arguments in its favor have been told time and again. We have already argued elsewhere that *plain* Scheme is a weak language for the first course and that it requires more support (Findler *et al.*, 2002). We briefly summarize these arguments here:

Scheme's syntax is simple. Indeed, it is too simple because almost every parenthesized expression is a syntactically valid program. When a student misplaces a parenthesis, the program may produce an indecipherable error message or a meaningless value. Our fix is to define a series of teaching subsets of Scheme and to implement each of them in our DrScheme programming environment. Implementing each subset enables us to produce error messages on the appropriate knowledge level for beginners.[4]

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. Semantically speaking, the language is a generalization of high school algebra. If a Scheme implementation comes with an algebraic stepper that illustrates this concept (Clements *et al.*, 2001), students can easily explore a program's evaluation without thinking about registers, stacks, memory cells, and other low-level concepts.[5]

Scheme is safe. More precisely, Scheme's standard (Kelsey *et al.*, 1998) allows a Scheme implementation to be safe. DrScheme, for example, implements a safe language with fully predictable behavior. When a computational operation violates its stated invariants, the implementation 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 language.[6]

Scheme is dynamically typed. The lack of a type system means that we don't have to spend energy on finding and explaining type errors with the same care with which we explain syntax errors. Better yet, when we use Scheme to teach design principles we can informally superimpose a type system and use the types

[4] In the 1970s, instructors who taught PL/1 faced a similar challenge and came up with a similar solution, though without the full compiler support for error messages that we provide (Holt *et al.*, 1977).

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

[6] This partly explains why C++ is such a failure. Its lack of safety does not even guarantee that when a program prints a number, it is actually interpreting bits that represent a number. Similarly, core dumps and bus errors are much worse than exceptions, because they typically happen long after the first violation occurred.

for program design. In particular, it is easy to define and use sets and subsets of Scheme values. This comes close to students' intuitions about classes and subclasses in object-oriented programs and thus provides a good transition for the second course.

3.3 Programming environments

The choice of language for a first-year course isn't just about the linguistics; it must also take into account the programming environment. After all, developing and running a program means more than just writing correct code. It requires support for editing; compiling and running programs; understanding how a program is evaluated; and so on.

Like the language, we believe that the programming environment for the first course should be a lightweight, easy-to-use tool. That is, it should provide just enough to edit and execute functions and programs, plus some tools for understanding fundamental concepts, e.g., lexical scope and program reduction. Everything else should be hidden from the student.

We believe that the lack of such a programming environment hurt the *SICP* approach of teaching and the functional community in general. For that reason, we have produced a programming environment that supports teaching program design principles with Scheme (Findler *et al.*, 2002). Others have had similar insights and have produced alternative environments independently (Schemer's Inc., 1991).

4 Interpretation: teaching design principles

4.1 Structure and Interpretation of Computer Programs

SICP covers many important program design ideas. The course starts with an overview of Scheme and recursive programming. In parallel, the course explains how to evaluate variable expressions and function applications; that is, it introduces a symbolic model of computation so that students understand the actions that a program performs. The book then covers topics such as higher-order procedural abstraction; data abstraction; mutable data objects; a message passing model of objects; streams; modularity; meta-linguistic abstraction; and compilation.

Although this collection of topics is impressive at first glance, a second look shows that SICP suffers from a serious flaw. While the course briefly explains programming as the definition of some recursive 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.

More generally, SICP doesn't state how to program and how to manage the design of a program. It leaves these things *implicit* and implies that students can discover a discipline of design and programming on their own. The course presents the various uses and roles of programming ideas with a series of examples. Some exercises then ask students to modify this code basis, requiring students to read and study code; others ask them to solve similar problems, which means they have to study the construction and to change it to the best of their abilities. In short, SICP students learn by copying and modifying code, which is barely an improvement over typical programming text books.

SICP's second major problem concerns its selection of examples and exercises. All of these use complex domain knowledge. Consider the left column in figure 1. It presents the choice of major examples that are used in the first few chapters of SICP. Some early sections and the last two chapters cover topics from computer science: see lower half of the left column in figure 1.

While these topics are interesting to students who use computing in electrical engineering and to those who already have significant experience of programming and computing, they assume too much understanding from students who haven't understood programming yet and they assume too much domain knowledge from any beginning student who needs to acquire program design skills. On the average, beginners are not interested in mathematics and electrical engineering, and they do not have ready access to the domain knowledge necessary for solving the domain problems. As a result, SICP 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] Similarly, many students lack an understanding of the role of compilers, logical models of program execution, and so on. While first-semester students should definitely find out about these ideas, they should do so in a context that reaffirms the program design lessons.

In summary, while SICP does an excellent job shifting the focus of the first course to challenging computer science topics, it fails to recognize the role of the first course in the overall curriculum. In particular, SICP's implicit approach to program design ideas and its emphasis on complex domains obscures the goal of the first course as seen from the perspective of a typical four-year curriculum.

4.2 How to Design Programs

Over the past few years, we have developed an alternative approach to teaching the first course. We have translated the approach into a new text book, and we believe that it addresses SICP's failings along four dimensions. First, the book discusses

[7] Some faculty members argue that a course on introductory programming is a good place for teaching students mathematical problem solving. 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.

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
normal/applicative order	checking (on) queens
strictness/laziness	accumulators on trees
non-determinism	missionaries and cannibals
logic programming	board solitaire
register machines	exploring places
compilers	moving pictures, a last time

Fig. 1. SICP and HTDP exercises

explicitly how programs should be constructed. Second, to tame the complexity of programming, it defines a series of teaching languages based on Scheme that represent five distinct knowledge levels through which students pass during their first course. The levels correspond to the complexity of data definitions that the program design guidelines use. Third, the book uses exercises to reinforce the explicit guidelines on program design; few, if any, exercises are designed for the sake of domain knowledge. Finally, the book uses more accessible forms of domain knowledge than SICP. Because of this shift in emphasis, we gave our book the title *How to Design Programs* (HTDP).

A cursory look at HTDP's table of contents reveals the new emphasis. Every chapter comes with at least one section on the design of a particular class of functions. At the same time, no section title concerns domain knowledge, except for those labeled "extended exercise."

HTDP's explicit design knowledge is encapsulated in design recipes. Every design recipe enforces basic design habits:[8]

1. analyze the problem and describe the classes of problem data;
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 steps 1 through 3;
5. write code; and

[8] Glaser et al.'s notion (Glaser *et al.*, 2000) of "programming by numbers" is a simple version of our notion of design recipe. It only uses the core (step 4) of our design recipes for functions on algebraic datatypes.

6. turn the examples into (automatic) test cases.

The book contains a series of approximately 10 design recipes. The first half of the series shows how the description of the classes of data suggest a natural organization of the functions that process them. These recipes addresses the design of functions for classes of atomic data (numbers, booleans, characters), intervals and unions, composites, self-referential definitions, groups of mutually referential definitions, and so on. The second half of the series 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 especially address the topic of *when* to use a technique or mode of an existing recipe; no technique is introduced as just another trick for the toolbox.

The recipes also introduce a new distinction into program design: *structural* versus *generative* recursion. The structural 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 *insort* and *kwik*, two standard sort functions:

```
;; (listof X) → (listof X)
(define (insort l)
  (cond
    [(empty? l) empty]
    [else
      (place
        (first l)
        (insort (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, *insort*, recurs on a structural portion of the given datum, namely, (*rest l*). The second function, *kwik*, recurs on data that are 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 that students add a discussion about termination to the definition of generative recursive functions.

Distinguishing the two forms of recursion and focusing on the structural case makes our approach scalable to the object-oriented (OO) world. In an OO world, the structural recipes naturally suggest class hierarchies and recursive 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.

Contrast this with SICP's treatment of recursion. The two notions are not distinguished and, worse, the book's first recursive procedure (*sqrt-iter* page 23) uses generative recursion. The structural aspect of recursion is almost ignored and certainly never presented as the bridge to object-oriented programming. More generally, because SICP misses structural recursion and structural reasoning, it confuses implementing objects with object-oriented programming. The book never actually discusses reasoning about, and programming with, classes of data, which is the essence of modern OO programming.

HTDP introduces the idea of iterative refinement for both programs and data separately. As students learn to cope with increasingly complex forms of data, the book shows how a programmer can design programs with a series of correspondingly more precise data representations. As the representations become more precise, the program implements more of the desired functionalities. Combining the design recipes with the idea of refinement then helps students produce complex programs systematically.

HTDP and SICP also vastly differ with regard to the treatment of language syntax. HTDP uses an analog of Quine's approach to studying set theory and its logic (van Orman Quine, 1963). Each language level is tuned to a particular stage in the exploration of design. HTDP shows what kinds of programs are natural to write and explains during the next stage why a construct should be added. Thus, for example, HTDP students work with classes of data and hierarchies of classes long before they encounter an assignment statement and before DrScheme interprets **set!** for them. This represents our insight that it is critically important for students to organize programs according to measurable criteria and for teachers to be able to tell students when working programs are justifiably bad.

Finally, HTDP uses domain knowledge differently from SICP. Figure 1 juxtaposes the section titles in SICP and HTDP that are concerned with exercises. Even a short glance shows that HTDP uses domain knowledge that is within reach of most students. It does offer 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.

5 Experience and outlook

The HTDP approach has been implemented at about a dozen colleges and, to some extent, at several dozen high schools. At the college level, the change has always shown strong results. For example, at Rice University and at Northeastern University, students can/could enter the second course (using Java) from either an HTDP course (taught in computer science) or a C++ course (taught in computer engineering). At both universities, independent instructors confirmed that the HTDP students are better prepared to program in an OO world than the C++ students and that they have much better programming habits. The Northeastern HTDP students received five times as many A's (best grade) as the C++ students; at the other grade levels, the numbers are approximately the same.[9]

High school teachers who implement HTDP report similar success stories as colleges but in a less measurable manner. Still, the HTDP curriculum has had an interesting measurable effect concerning female students. Several instructors reported that female students like the *HtDP* curriculum exceptionally well. In a controlled experiment, an HTDP-trained instructor taught a conventional AP curriculum and the Scheme curriculum to the *same* three classes of students. Together the three classes consisted of over 70 students. While all students preferred our approach to programming, the preference among females was a stunning factor of four. An independent evaluator is now investigating this aspect of the project in more depth.

In general, we believe that the HTDP project has validated the usefulness of functional programming and functional programming languages in the first programming course. We have found that teaching Scheme for Scheme's sake (or Haskell for Haskell's sake) won't work. Combining SICP with a GUI-based development environment for Scheme won't work better than plain SICP. The two keys to our success were to tame Scheme into teaching languages that beginners can handle and to distill well-known functional principles of programming into generally applicable design recipes. Then we could show our colleagues that a combination of functional programming as a preparation for a course on object-oriented programming is an effective and indeed superior alternative to a year on just C++, Java, or a combination.

We are hoping that other functional communities can replicate our success in different contexts. We suggest, however, that using plain Erlang, Haskell, or ML and that teaching programming in these languages implicitly will not do. We all need to understand the role of functional programming in our curricula and the needs of our students. Fortunately, Chakravarty and Keller's recent educational pearl (Chakravarty & Keller, 2004) shows that we are not the only ones who have recognized the deficiencies of conventional approaches.

Note: DrScheme and *How to Design Programs* are freely available on the Web at `http://www.teach-scheme.org/`.

References

Abelson, Harold, Sussman, Gerald Jay, & Sussman, Julie. (1985). *Structure and interpretation of computer programs*. MIT Press.

Bird, R., & Wadler, P. (1988). *Introduction to functional programming*. Prentice Hall International, New York.

Chakravarty, Manuel M. T., & Keller, Gabriele. (2004). The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, **to appear**, ??–??

[9] The numbers are normalized. The sample is approximately 150 students with approximately 40 students from C++ and the rest from an *HtDP* course.

Clements, John, Flatt, Matthew, & Felleisen, Matthias. (2001). Modeling an algebraic stepper. *European symposium on programming*.

Clinger, William. (1985). *The revised revised report on the algorithmic language Scheme*. Joint technical report. Indiana University and MIT.

Clinger, William, & Rees, Jonathan. (1991). The revised^4 report on the algorithmic language Scheme. *ACM Lisp pointers*, **4**(3).

Findler, Robert Bruce, Clements, John, Flanagan, Cormac, Flatt, Matthew, Krishnamurthi, Shriram, Steckler, Paul, & Felleisen, Matthias. (2002). DrScheme: A programming environment for Scheme. *Journal of functional programming*, **12**(2), 159–182. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pages 369–388.

Glaser, H., Hartel, P. H., & Garratt, P. W. (2000). Programming by numbers – a programming method for complete novices. *Computer journal*, **43**(4), 252–265.

Holt, R.C., Wortman, D.B., Barnard, D.T., & Cordy, J.R. (1977). Sp/k: A system for teaching computer programming. *Communications of the acm*, **20**(5), 301–309.

Hudak, Paul. (2000). *The Haskell school of expression*. Cambridge University Press.

Jackson, Daniel, & Chapin, John. (2000). Redesigning air traffic control: An exercise in software design. *Ieee software*, **17**(3).

Kelsey, Richard, Clinger, William, & Rees (Editors), Jonathan. (1998). Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, **33**(9), 26–76.

Raymond, Eric S. (1998). The cathedral and the bazaar. *First monday*, **3**(3).

Schemer's Inc. (1991). *EdScheme: A modern Lisp*.

Steele Jr., Guy Lewis, & Sussman, Gerald L. (1978). *The revised report on scheme, a dialect of lisp*. Tech. rept. 452. MIT Artificial Intelligence Laboratory.

Sussman, Gerald L., & Steele Jr., Guy Lewis. (1975). *Scheme: An interpreter for extended lambda calculus*. Tech. rept. 349. MIT Artificial Intelligence Laboratory.

van Orman Quine, Willard. (1963). *Set theorey and its logic*. Harvard Press.

Wadler, Philip. (1987). A critique of Abelson and Sussman, or, why calculating is better than scheming. *SIGPLAN Notices*, **22**(3).