The TeachScheme! Project: Computing and Programming for Every Student

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

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

Abstract. The TeachScheme! project aims to reform three aspects of introductory high school courses on programming. The first is a systematic program design method. The key property of the method is that it asks students to design programs in a stepwise fashion. Each step produces a well-specified intermediate product. It thus gets students started and helps them to overcome obstacles. Furthermore, it empowers teachers to evaluate the reasoning process and not just the final product. The second novelty is the use of a series of increasingly powerful programming languages instead of a single (subset of a) language. Each element of the series introduce students to specific linguistic mechanisms and thus represents a cognitive development stage in a concrete manner. Consequently, the language implementations can provide knowledge-appropriate feedback when errors occur. The third new component is a program development environment that was specifically developed for beginners. It supports the teaching languages in a uniform manner and provides tools that assist with each stage in the curriculum. This paper reports on these three efforts. It includes a preliminary evaluation report on the effects of these changes on teachers and students.

1 Computing and Programming Matters to Everyone

Everyone programs. Accountants program spreadsheets and secretaries sometimes program word processors; photographers program photo editors; musicians program synthesizers; car mechanics program diagnosis tools; and professional programmers instruct plain computers. In short, it is increasingly clear that programming is becoming as indispensable a skill as arithmetic and natural language.

Yet programming is also more than just a vocational skill. Indeed, *good programming* can play the same role as any other creative activity, like playing a musical instrument, painting, writing, photography, and so on. Put differently, a programmer is an artist, and "hacking a program" can provide the same satisfaction that painting can generate [9].

Last, but not least, if programming is taught as a systematic design activity, it also teaches a variety of important skills. As many before us have observed, good programming requires critical reading, analytical thinking, creative synthesis, and attention to detail. And all of these skills are useful in many professions.

Unfortunately, today's education systems teach outdated views of programming in secondary schools. First, schools consider programming a part of vocational studies rather than the liberal arts core. This strongly affects the content of the courses; indeed, in many places "introduction to computer science" is a course on application software and has nothing to do with programming and computing. Second, high schools often let the grammar of currently fashionable, vocational programming languages dictate their curriculum rather than sound principles of design and problem solving. Third, schools employ programming technology that is intended for professional programmers. Unfortunately, industry does not build products with novices in mind. Instead it sells so-called educational editions of Integrated Development Environments (IDEs), which are just cheapened versions of the professional products.

Given the situation, it should not come as a surprise that US high schools don't enroll a large fraction of their students in computer science courses. For an indication, consider the following table from the Educational Testing Service's Advanced Placement (AP) Yearbook:

Discipline	Male		Female		Total	
	2001	2002	2001	2002	2001	2002
Calc AB, BC	100,766	107,767	84,138	91,542	184,904	199,309
Stat	20,842	24,961	20,767	24,863	41,609	49,824
Comp Sci A, AB	19,891	20,094	3,531	3,365	23,422	23,459

Eight times as many students take the mathematics test compared to computer science; even statistics is more than twice as popular as computer science. Considering how much computational thinking helps people directly, these numbers are simply unacceptable. Another indication of the vast problems with the existing approach is that among female test takers, mathematics is 30 times as popular, statistics eight times.

This paper presents the TeachScheme! project, an initiative that we started in 1996. The objective of the project is to develop an alternative to the existing high school curricula with the long-term goal of overcoming all the diagnosed problems. The major differences concern three aspects:

1. First, we have designed a series of programming languages for a matching introductory curriculum. Each element of the series represents a cognitive stage of the learning process in a concrete manner.
2. Second, we have also developed a matching IDE. The environment implements the cognitive stages and also helps students understand other language concepts, such as evaluation and lexical scope.
3. Third, we have created a program design method that helps beginning students and their teachers. If students follow the design method, they produce several intermediate products and a program with a well-defined, checkable aesthetic. A teacher can use the intermediate products to help a student in such a way that the student learns to help himself or to grade flawed programs, because the intermediate products expose the problems with the student's thinking. Similarly, a teacher can use the guideline to justify grading standards that evaluate structural or aesthetic aspects of the final product.

The project is now in its sixth supported year. We have trained over 200 teachers and college colleagues in the use of the program design method and the supportive tools. An independent evaluator is in the process of evaluating the program formally, but our first evaluations are already highly positive.

In the next three sections, we present the primary improvements over the existing computing and programming curricula. The fifth section is a preliminary experiences and evaluation report. In the sixth section, we discuss how the curriculum fits into an existing context. The final section summarizes our experiences and presents some plans for the future.

2 The Programming Language and the Curriculum

Beginners make all kinds of mistakes. Their programs contain syntax (compile time) errors, safety (run time) errors, and logical errors (mismatches with specifications). When the feedback for errors is obscure, beginners get easily frustrated. Hence, it is critical that an introductory course on programming must use ideas and tools that help students recognize and overcome errors.

To understand the nature of beginner errors, we started our project with comprehensive observation sessions in our own lab (at Rice University) and in local high schools (Houston). We quickly found out that beginners had severe problems with *all* programming languages (C++, Pascal, and Scheme) and programming environments.

Here is a scenario we observed during the first two weeks of AP courses (this scenario is not atypical). A student's C++ program contained the following fragment:

```
price_per_piece * number_of_pieces = total_cost;
```

Naturally, the compiler flags the left-hand side of this assignment statement and explains that an assignment statement expects an *lhs value*. Unfortunately, this explanation is completely obscure for the student, because she simply doesn't know about "lhs values" or pointer arithmetic. During most of our observations, some helpful classmate or perhaps the teacher would eventually suggest to swap the two sides of the "equation", and the student would then happily compile her program. The situation was no better in Pascal or Scheme labs.

Based on those observations, we decided to co-design a set of curricular goals, a programming language, a programming environment, and a program design method that helps students discover and overcome mistakes. More concretely, we specified three major phases in our curriculum and specified what students should (and should not) understand during those phases. Using these goals, we then designed a *series of programming languages* based on Scheme [11].

Figure 1 summarizes the three phases.[5] Let us consider each phase in turn:

beginners A beginner according to our curricular goals must learn to cope with classes of values and must learn to develop functions (or "methods") on such classes. Values can be either atomic or composite. Concerning functions, the curriculum first covers functions like those that students encounter in algebra, but our curriculum naturally extends this concept to a variety of data, not just numbers. Later, students encounter conditional expressions so that they can define functions that consume values from unions of classes (e.g., a class of geometric shapes that consists of the classes of squares, circles, and triangles) and recursion so that they can define functions that consume values from classes with recursive descriptions (e.g., the class of lists of invitees).

[5] Due to additional observations, we currently work with *five* teaching languages.

phase	goals	language characteristics
beginners	atomic values functions on atomic values conditionals structural values functions on structural values functions on unions of classes functions on recursive unions of classes writing automated tests	numbers, characters, symbols, strings, booleans basic functions (+, *symbol=?*, ...) function definition (**define**) and application predicates (*number?*, *symbol?*, ...) **cond** structure definition creation and destructuring **computational model**: algebra
intermed.	abstracting over recurring patterns functions as first-class objects generative recursion: why, how recursion with accumulators: why, how	local definitions (**local**) of variables, functions, and structures syntax for anonymous functions (**lambda**) **computational model**: algebra
advanced	changing the value of variables: why, how mutating structures: why, how	assignment statements (**set!**) mutator functions **computational model**: modified algebra

Fig. 1. The programming languages

To understand these ideas, it suffices to introduce a first-order functional language with only four syntactic constructions: function definition, function application, conditional expressions, and structure definitions. In Scheme, everything else is just the application of primitives such as +, which come with the language, or *make-person*, which is a function that a structure definition introduces.

```
(define (fahrenheit-as-celcius f)
  (* 5/9 (- f 32)))

What (and why) is the value of:

  (fahrenheit-as-celcius 32)
 = (* 5/9 (- f 32))
 = (* 5/9 0)
 = 0
```

```
(define-struct person (prefix first last))

(define (greeting a-person)
  (string-append
    "Dear "
    (person-prefix a-person) " "
    (person-last a-person) ":"))

What (and why) is the value of:

  (greeting (make-person "Ms" "Kathi" "Fisler"))
 = (string-append
    "Dear "
    (person-prefix (make-person ...)) " "
    (person-last (make-person ...)) ":")
 = (string-append
    "Dear "
    "Ms" " "
    (person-last (make-person ...)) ":")
 = ...
 = "Dear Ms Fisler:"
```

Fig. 2. The computational model of Scheme

intermediate As students study programming and computing with first-order functions, they quickly recognize that they repeatedly use similar patterns. It is therefore natural to introduce them to the systematic design of abstraction from concrete instances. The best way to introduce abstraction in Scheme is with functions as first-class values, in particular, functions that can consume functions as arguments.

Our intermediate teaching language therefore removes restrictions on the use of functions names and adds two constructs to the beginner language: **local** and **lambda**. More precisely, in the intermediate language the names of function are now not only allowed in the function position of an application, but also in all expression positions. The **local** construct introduces students to lexically nested definitions; the **lambda** construct allows them to define anonymous functions.

Other goals in this phase, such as generative recursion and accumulator-style data processing, do *not* need additional language constructs. Due to their complex nature, however, they are only introduced after students have mastered the basics of abstraction.

advanced The last language in our series is for advanced beginners. The goal for those students is to master the notoriously difficult notion of state in programs [10]. The use of

state is necessary when functions wish to record historic information or when programmers want physically distinct functions to communicate without composing them. In object-oriented languages, state is best implemented through assignments to field variables. In procedural languages such as Pascal or functional languages such as Scheme, this corresponds to the mutation of records or structures, i.e., assignments to fields. Both languages also tend to include vector mutations.
To teach assignment and structure mutation, our advanced programming language interprets the definition of structures differently from the beginning and intermediate languages. The latter generate functions for creating structures and for selecting field values from structure definitions; the advanced languages also generates functions that can mutate the field of a structure. In addition, the advanced language level also includes functions for mutating vectors, strings, and lists.

We decided to start with classes of values and functions on those classes, because it is conceptually just a generalization of algebra. In particular, students already know how to evaluate a numeric variable expressions and function applications. Since function applications are the primary vehicle of computation in the "beginner" language, students have a near-complete model of the computations that take place in the computer when a program is finally applied to values. (That is, when algebraic reduction is the model of computation, students enter the course already knowing how "the hardware" works. Therefore, valuable time that is otherwise spent teaching a physical computational model can be spent teaching more interesting and challenging material, postponing the discussion of the physical model until later.)

Consider the example in figure 2. Both examples columns depict programs that our students tend to write after a couple of lessons. The program on the left converts Fahrenheit temperatures into Celsius temperatures; the one on the right computes the opening line of a letter. In other words, the left one is a function that students know from ordinary pre-algebra courses, the one on the right is a similar function but works on personnel records and strings. The bottom half of each column shows how to evaluate a specific function application. The evaluation on the left again follows the familiar pattern of pre-algebra. The one on the right is a small generalization of the algebraic substitution rules to structures (records) and strings.

The first part of our introductory curriculum is thus neatly integrated with pre-algebra and algebra courses at approximately the same level. Students who enter the programming course can thus exploit their knowledge of algebra. Or, our course on programming can reinforce their algebra knowledge.[6]

The second part of the curriculum extends the algebraic model of calculation, but it doesn't fundamentally alter it. Take a look at figure 3. It defines the function *fold*, which is obviously similar to two first-order functions: *sum*, for adding up the numbers in a list of numbers, and *product*, for multiplying the numbers in a list of numbers.

The calculations below the definition illustrate how the computational model of the intermediate language extends that of the beginner language. In addition to ordinary values, *fold* also consumes a function. The calculation in the left column shows why *fold* is like *sum* and the one in the right shows why *fold* is like *product*. In both cases, the students deal with

[6] Indeed, to our surprise, some teachers have used the first part of our programming course to enliven, and make more accessible, their pre-algebra courses.

functions like $+$ and $*$ as ordinary values. They replace the parameter f of *fold* with one of the two functions and otherwise proceed as usual. Although the language is far more powerful than the beginner language, the computational model basically remains the same.

```
(define (fold f e alon)
  (cond
    [(empty? alon) e]
    [else (f (first alon) (fold f e (rest alon)))]))
```

What (and why) is the value of:

```
  (fold + 0 alon)
= (cond
    [(empty? alon) 0]
    [else (+ (first alon)
             (fold + 0 (rest alon)))])
= (cond
    [(empty? alon) 0]
    [else (+ (first alon)
             (sum (rest alon)))])
= (sum alon)
```

What (and why) is the value of:

```
  (fold * 1 alon)
= (cond
    [(empty? alon) 1]
    [else (* (first alon)
             (fold * 1 (rest alon)))])
= (cond
    [(empty? alon) 1]
    [else (* (first alon)
             (product (rest alon)))])
= (product alon)
```

Fig. 3. Evaluation with first-class functions

Finally, for the last part of the curriculum, the language introduces assignments and thus computational actions that are not just algebraic. Fortunately, it is possible to retain the substitution model in the presence of assignment statements: we just need to shift our perspective from the individual expression to the entire program [5]. A discussion of the relevant rules is beyond the scope of this paper, but they are accessible to students with experience in the first two languages.

Whether the chosen base language is Scheme or something else, providing a coherent series of sub-languages instead of a single language has two major advantages. First, starting with a small, well-defined language specifies implicitly what the appropriate linguistic mechanisms are. Hence, teachers can focus on the development of problem solving and program design skills. In particular, they no longer have to get into (usually niggling) discussions about whether particular lingusitic constructs are permissible at a certain level (such as using `for` as well as `while`). Later, as constructs are added, a discussion of the trade-offs and the reasons for using advanced constructs is natural.

Second, the representation of a student's understanding of programming via a specific sub-language enables the implementors to report errors in an knowledge-appropriate manner. Consider the specific example of a student who misplaces a parenthesis in a function application and writes

... *empty?*(*a-list*) ...

instead of

... (*empty? a-list*) ...

The implementation of the beginner language can explain that *empty?* is a function and must occur to the right of a left parenthesis. In contrast, a plain Scheme implementation would print an error message concerning higher-order functions or other notions that a beginner doesn't know.

While some curricular proposals have noticed the need for restricted subsets of full languages for teaching, most do not provide a means for enforcing this subset. This lack of enforcement is most unfortunate. Student errors invariably produce programs that fall outside this subset, and the lack of clear enforcement of the subset (as in the pointer example presented above) means the students sees a response from the computer that neither corresponds to concepts they know, nor makes clear that they have stepped outside the bounds of the subset. (Indeed, from their perspective, there *is* nothing outside the subset, since that is all the language they know at that point.) As a result, the creation of unenforced language subsets is almost as harmful as not defining one at all. Fortunately, the programming environment we discuss in the next session strictly and meticulously enforces the language subsets that we have discussed above.

3 The Program Development Environment and the Curriculum

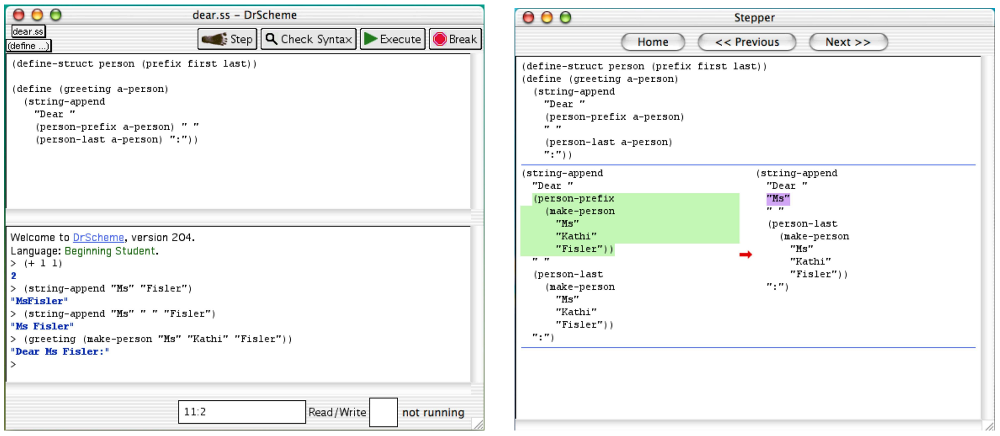
A programming language needs a program development environment (PDE). Roughly speaking, a PDE helps programmers with common task. Modern commercial PDEs, also known as integrated development environments (IDE), edit, compile, link, and run programs. Many come with a plethora of other tools for tasks ranging from project management to test coverage analysis.

Our above-mentioned classroom observations revealed, however, that these IDEs are often obstacles rather than helpful tools. Their complex control panels and their plethora of tools just confuse beginners. To write down even the simplest program, a novice sometimes has to create or copy a project, supply the proper paths to libraries, and arrange a package and class hierarchy. After a few interactions the monitor is often cluttered with a heap of windows and control panels. The learning editions of these PDEs don't improve this situation either, because they are often just cheapened cousins of their commercial editions. In short, like programming languages, commercial IDEs are intended for professional programmers and not tailored to the introductory curriculum and plain beginners.

DrScheme[7] is our response to these observations. Its design takes into account our classroom observations and the structure of our curriculum. We have reported on the implementation of DrScheme elsewhere [6]. Here we focus on the use of DrScheme with our curriculum.

The left column of figure 4 displays a screenshot of DrScheme. The plain PDE consists of just two panes: an editor at the top and an interactive evaluator for the chosen teaching language at the bottom. The evaluator is an extremely powerful calculator. In general, students can evaluate any expression in the evaluator. In particular, students can evaluate ordinary arithmetic expressions (line 1); they can explore how primitives work (lines 2 and 3); and they can apply their own functions to values (line 4).

[7] DrScheme, which runs on several platforms, is available for free from `www.drscheme.org`.



```
(define-struct person (prefix first last))

(define (greeting a-person)
  (string-append
    "Dear "
    (person-prefix a-person) " "
    (person-last a-person) ":"))
```

```
Welcome to DrScheme, version 204.
Language: Beginning Student.
> (+ 1 1)
2
> (string-append "Ms" "Fisler")
"MsFisler"
> (string-append "Ms" " " "Fisler")
"Ms Fisler"
> (greeting (make-person "Ms" "Kathi" "Fisler"))
"Dear Ms Fisler:"
>
```

```
(define-struct person (prefix first last))
(define (greeting a-person)
  (string-append
    "Dear "
    (person-prefix a-person)
    " "
    (person-last a-person)
    ":"))
```

```
(string-append                       (string-append
  "Dear "                              "Dear "
  (person-prefix                       "Ms"
    (make-person                       " "
      "Ms"                             (person-last
      "Kathi"                            (make-person
      "Fisler"))                   →       "Ms"
  " "                                      "Kathi"
  (person-last                             "Fisler"))
    (make-person                       ":")
      "Ms"
      "Kathi"
      "Fisler"))
  ":")
```

Fig. 4. DrScheme and the Stepper

Students define their own functions and structures in the editor. DrScheme's editor supports Scheme's syntax in a number of ways. Most importantly, it highlights maximal expressions in grey as the student types, which minimizes confusion with parentheses.

DrScheme's interface displays only five, carefully chosen buttons:

Execute evaluates the definitions and expressions in the editor and makes them available in the evaluator;
Break empowers programmers to stop any run-away program;
Save shows up when the program in the editor is modified and reminds students to save their work;
Check Syntax analyzes the syntax and the scope of the program in the editor (see below);
Step allows students to step through the evaluation of an expression.

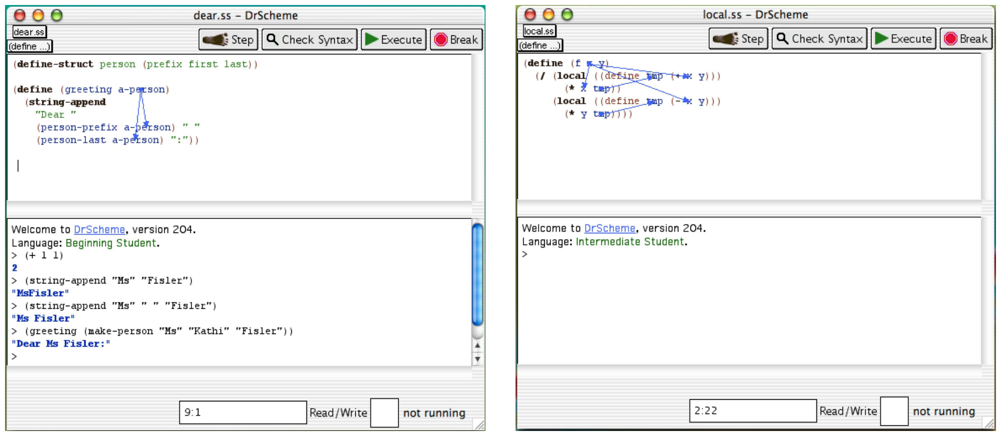
Besides the editor and the evaluator, the stepper is the most important tool that beginners use. Using the stepper, a teacher can easily explain a complete model of computation to students without ever mentioning a hardware concept. Indeed, when used with numeric programs, a teacher can use the stepper to solve the primitive homework exercises from the first few sections in any high school pre-algebra textbook.

Equally important, students can study the evaluation of an expression both without much help from another person, and without resorting to arcane terminology such as stacks, registers and so forth. They only need understand the syntax of the language and the idea of substituting equals for equals. This is particularly important when new constructs are introduced, e.g., structures, or when a construct's cognitive scope is extended, e.g., functions are defined in a recursive manner.

The right column in figure 4 depicts the stepper as it explains how the Scheme evaluator arrives at a value for the expression

(*greeting* (*make-person* "Ms" "Kathi" "Fisler"))

Specifically, the highlighted area to the left of the red area shows that the evaluator is about to extract the prefix field from a person structure. The highlighted area on the right shows the result. The context that surrounds these highlighted areas remains the same, illustrating the principle of substitution of equals for equals from algebra.



```
(define-struct person (prefix first last))

(define (greeting a-person)
  (string-append
    "Dear "
    (person-prefix a-person) " "
    (person-last a-person) ":"))
```

```
Welcome to DrScheme, version 204.
Language: Beginning Student.
> (+ 1 1)
2
> (string-append "Ms" "Fisler")
"MsFisler"
> (string-append "Ms" " " "Fisler")
"Ms Fisler"
> (greeting (make-person "Ms" "Kathi" "Fisler"))
"Dear Ms Fisler:"
>
```

```
(define (f x y)
  (/ (local ((define tmp (+ x y)))
       (* x tmp))
     (local ((define tmp (- x y)))
       (* y tmp))))
```

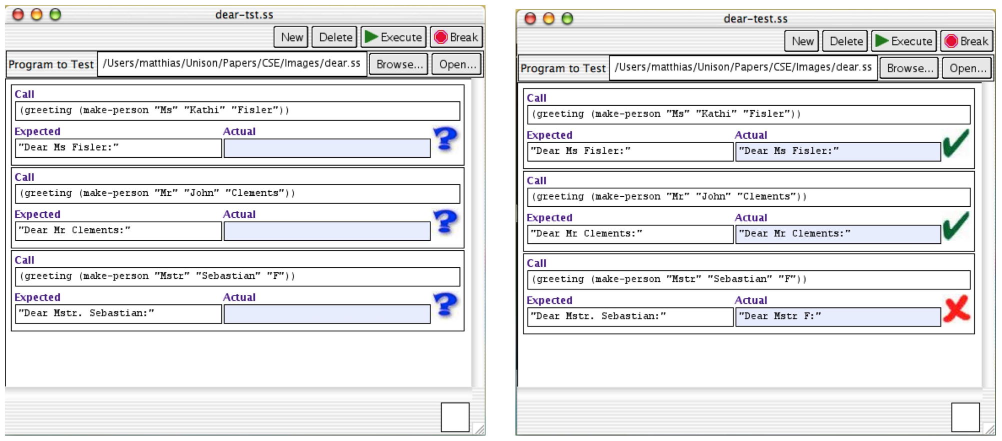
```
Welcome to DrScheme, version 204.
Language: Intermediate Student.
>
```

Fig. 5. DrScheme's syntax coloring and test suites

Like other PDEs, DrScheme also comes with a syntax coloring tool. The screenshot in figure 5 illustrates how the syntax coloring paints keywords, library functions, identifiers, and constants in different colors. In contrast to an ordinary syntax coloring tool, DrScheme's syntax analysis also understands the lexical scope of the program. When a programmer mouses over a function parameter, for example, DrScheme overlays arrows from the parameter to all its bound occurrences in the function's body. Furthermore, a programmer can also use the syntax checker to rename variable consistently. That is, if a programmer used *x* as a parameter for many functions and then wishes to rename the *x* in one particular function to something more meaningful, the syntax checker will rename that one *x* without affecting any others (technically, it *alpha*-renames); furthermore, it does this with just a few mouse clicks.

The syntax checker is an important tool for the second language level. There students encounter constructs for lexically nested definitions and their teacher must explain the scoping rules and the binding power of variable definitions. The right side of figure 5 shows how the syntax checker deals with a toy program that uses local definitions. The arrows indicate, for example, that the scope of *x* is the entire function body, while the scope for the locally defined *tmp* variables is just the two **local** expressions, respectively.

We are currently also developing an integrated test suite manager. Working with examples and testing are key elements of our program design method. The testing tool (which currently uses a separate window) provides an explicit space for writing down examples and tests during the development of the program. We are also developing a coverage analysis tool. When both tools are fully integrated into DrScheme, the environment will evaluate all test suites every time a student clicks "Execute" and will then highlight those portions of the program in red that the current test suite doesn't cover.



Program to Test /Users/matthias/Unison/Papers/CSE/Images/dear.ss

Call
```
(greeting (make-person "Ms" "Kathi" "Fisler"))
```
Expected `"Dear Ms Fisler:"` Actual ?

Call
```
(greeting (make-person "Mr" "John" "Clements"))
```
Expected `"Dear Mr Clements:"` Actual ?

Call
```
(greeting (make-person "Mstr" "Sebastian" "F"))
```
Expected `"Dear Mstr. Sebastian:"` Actual ?

Program to Test /Users/matthias/Unison/Papers/CSE/Images/dear.ss

Call
```
(greeting (make-person "Ms" "Kathi" "Fisler"))
```
Expected `"Dear Ms Fisler:"` Actual `"Dear Ms Fisler:"` ✓

Call
```
(greeting (make-person "Mr" "John" "Clements"))
```
Expected `"Dear Mr Clements:"` Actual `"Dear Mr Clements:"` ✓

Call
```
(greeting (make-person "Mstr" "Sebastian" "F"))
```
Expected `"Dear Mstr. Sebastian:"` Actual `"Dear Mstr F:"` ✗

Fig. 6. DrScheme's test suites

Figure 6 shows a before-and-after screen shot of a test suite for the *greeting* function. The test suite contains three tests, one for a woman, one for a man, and one for a child. Before running the test suite, the evaluator indicates with "?" that nothing is known about the validity of the example. After the execution, a check mark is next to all those examples for which the expected and the actual result are the same; for all others, DrScheme displays a red cross mark.

Based on preliminary observations, we believe that the test suite will strongly reinforce the our culture of examples and testing from the very beginning of the course. Since this aspect of our program design method had thus far been neglected in the PDE, we believe that the development of the test suite tool will finally bring our PDE into close synchronization with our program design method.

4 The Program Design Method

Like the programming language and the programming environment, the program design method in an introductory course must focus on beginners. Starting from this observation, we suggest three concrete goals for any program design method. First, since beginners make mistakes, the method must help students recognize design flaws and avoid them. Second, the design method must help teachers evaluate their students' reasoning process, not just the final product. This is necessary for grading students' work with objective criteria and for helping them throughout the process. Last, but not least, the method must instill good habits that scale to large projects. After all, the first program design method sets the tone for many students' further learning and, what they don't practice properly at this stage, they may never practice properly later.[8]

[8] Indeed, we maintain that these program design methods instill habits and trains skills that are useful above and beyond a programming course or curriculum. Since this claim is beyond the scope of this paper, we won't pursue it here.

phase	product	steps
problem and data analysis	data definitions	name the program read the problem statement determine the classes of data that the program consumes and produces
goal formulation	purpose statement contracts function header(s)	formulate a concise statement of *what* the program is to compute specify which classes of data the program consumes and which one it produces this may involve the parameters from the function header
examples	examples of data examples of the function's i/o behavior	use the data definitions to create typical examples of data use the purpose statement to create examples of what the function produces, given some concrete inputs
organization	function template(s)	use the data definitions to organize the function without regard to its purpose
programming	complete function(s)	to fill the holes in the template, using the purpose statement, the data structure of the data definition, and the examples
testing	test suite & coverage	to look for mistakes using the examples from step 2; mistakes can show up in the examples, the program, and/or both

Fig. 7. The generic program design recipe

Given these goals, it is natural to teach programming as a prescriptive process. We have formulated the process as a collection of *program design recipes*. Each design recipe consists of five to seven steps. Each step produces a well-defined intermediate product. Hence, when students come to a teacher because they are stuck, the teacher can ask how many steps are completed and can ask to see the products of these steps. After a while the students recognize that they can follow the appropriate design recipe on their own. Similarly, grading a program is no longer a process of guessing how many points to give for some correct statement in a program. Instead a teacher can inspect the process that produced the program and grade the student's reasoning. In short, shifting the focus from the final product (the program) to the process (of designing a program systematically) has many advantages for both teachers and students.

4.1 Design Recipe: The Basics

Figure 7 describes the generic design recipe for the first third of the course. This first part of the course actually covers six different design recipes. Each design recipe covers a particular kind of data definition. As the course proceeds the complexity of the data definitions grows. Using naive set theory as a rough guide, our course introduces these six forms of data definitions:

1. a data definition that describes a class of atomic data
2. a data definition that introduces intervals
3. a data definition that introduces a class of structures
4. a data definition that introduces a union of classes
5. a self-referential data definition (for a union of classes)
6. several mutually referential data definitions

The shape of the data definition is particularly important for steps 4 and 5 of the generic design recipe (see figure 7). Let us illustrate the use of the design recipe with some examples.

Example 1 Recall the function *greeting* from figure 4. Here is a problem statement that might produce this function:

> **Problem** 1: Design the function *greeting*, which formulates the opening line for a letter. The function consumes personnel records, which contain preferred prefixes, first and last names. It produces the greeting as a string.

The problem is for students who have just encountered structured data and need to practice working with structures and recognizing when structures are needed.

As the problem says, we need to represent information about people. The information consists of a fixed number of pieces of information. This implies that a structured form of data is most appropriate, yielding this data definition:

```
(define-struct person (prefix first last))
;; A Person is a structure:
;; — (make-person String String String)
```

The definition consists of one line of Scheme and two lines of comments. It introduces a constructive structure definition and a method for using the structure.[9]

The problem is easy to condense into a purpose statement with a contract:

```
;; greeting : Person → String
;; to produce a letter greeting from a-person, that is, a Person structure
(define (greeting a-person) ...)
```

The first line is the contract. It specifies the name of the function and the classes of data that the function consumes and produces. The second line is the purpose statement, which refers to the parameter of the function header in the third line.

Our data definitions are formulated in such a way that it is easy to make examples. For *Person*, a sample structure is constructed by applying the constructor *make-person* to three strings. From the structure definition, we know that they are a prefix, a first name, and a last name. So here are some examples:

```
(make-person "Ms" "Kathi" "Fisler")
(make-person "Mr" "John" "Clements")
```

The recipe now calls for the construction of i/o examples for *greeting* from these examples. In the past we have suggested that students add these to the bottom of the editor pane, producing the code in figure 8. Each example consists of a complete function call and the

```
;; greeting : Person → String
;; to produce a letter greeting from a-person, that is, a Person structure
(define (greeting a-person)
  ...)

(greeting (make-person "Ms" "Kathi" "Fisler"))
;; should produce
"Dear Ms Fisler:"
(greeting (make-person "Mr" "John" "Clements"))
;; should produce
"Dear Mr Clements:"
```

Fig. 8. Adding examples

expected value. For this example, we make up expected values because the problem statement doesn't specify the precise nature of the greeting. At this point, a teacher can also explain how examples can help to make vague specifications more precise *before* we waste too much energy on the program proper.

After these preliminary steps, it is time to organize the known facts. We have dubbed the outcome of this step a *function template*, because these organizations can often be reused for functions with the same domain. Roughly speaking, the template is a translation of the

[9] In a language such as C++ or Pascal, this is a single typed definition of a structure or a primitive class.

structure of data definition	structure of template	advice for program
atomic values	domain knowledge	
N intervals N enumerations	**cond** with N branches	deal with each branch separately; use domain knowledge
structures with M fields	M selector expressions	"combine" (+, *cons*, . . .) the field values with others
union of L classes	**cond** with L clauses	deal with each branch separately
self-referential union of K classes	recursive function definition, using a **cond** with K branches	deal with non-recursive cases first; rewrite the purpose statement for the recursive calls
⋮	⋮	⋮

Fig. 9. Construction templates and programs from data definitions

data definition for the inputs into program fragments. Since the functions (at this stage) can only produce information from the information that they consume, basing the structure of the program on the structure of the input data definition is natural. Students can then just re-combine the pieces, possibly adding in some constants, to get the final function. Also, if the data definition ever changes, it is almost obvious how the program changes.

Figure 9 displays some basic hints on how to proceed. Given that the data definition of our running example involves structures and given the hints in the table, the template for a function that consumes a *Person* structure is thus as follows:

```
;; greeting : Person → String
;; to produce a letter greeting from a-person, that is, a Person structure
(define (greeting a-person)
  ... (person-prefix a-person) ...
  ... (person-first a-person) ...
  ... (person-last a-person) ...)
```

Since a *person* structure has three fields, we added three expressions. Each expression extracts one field value from *a-person*, the parameter that represents the *Person* structure to which the function will be applied.

Now, and only now, students are allowed to program in the narrow sense of the word. The advice of the design recipe is that they must consider what each expression in the function template represents. In our running example, the selector expressions combined with some constants is almost everything the programmer needs; as a matter of fact, one of the selector expressions (that for the first name) is superfluous. To finish the definition, the students just need to append all these strings to obtain the full function definition.

Figures 4 and 6 show the final result. Students enter the definitions and the comments (not shown) into the editor and the test cases into the test suite manager. When they click "Execute" in the test suite manager, DrScheme tests all the examples and then makes its evaluator listen for additional experiments that students may wish to conduct.

Example 2 As the test suite window shows, tests can go wrong. The first version of *greeting* does not produce the expected answer for inputs that represent young boys. Those should be addressed as ‘”Mstr” with their first names. Let us look at a matching extension of the problem 1:

> **Problem** 1′: The function should distinguish between adult prefixes (`"Mr"`, `"Ms"`) and prefixes for youth (`"Mstr"`, `"Miss"`). For the former, use the last name, for the latter the first name, plus the prefixes.

This problem extension introduces a new form of data and restricts the class of *Person* structures:

```
(define-struct person (prefix first last))
;; A Person is a structure:
;; — (make-person Prefix String String)
;; A Prefix is one of:
;; — "Mr"
;; — "Ms"
;; — "Mstr"
;; — "Miss"
```

```
;; greeting : Person → String
;; to produce a letter greeting from a-person, that is, a Person structure (version 2)
(define (greeting a-person)
  (cond
    [(string=? (person-prefix a-person) "Ms")
     ... (person-first a-person) ... (person-last a-person) ...]
    [(string=? (person-prefix a-person) "Mr")
     ... (person-first a-person) ... (person-last a-person) ...]
    [(string=? (person-prefix a-person) "Miss")
     ... (person-first a-person) ... (person-last a-person) ...]
    [(string=? (person-prefix a-person) "Mstr")
     ... (person-first a-person) ... (person-last a-person) ...]))

(greeting (make-person "Ms" "Kathi" "Fisler"))
;; should produce
"Dear Ms Fisler:"
(greeting (make-person "Mr" "John" "Clements"))
;; should produce
"Dear Mr Clements:"
(greeting (make-person "Mstr" "Kaspar" "Weimer"))
;; should produce
"Dear Mstr Kaspar:"
```

Fig. 10. A template for *greeting* (version 2)

The revised data definition does not affect the purpose statement, the contract, the function header, or examples per se, assuming the third example from figure 6 is already included. It does, however, affect the construction of the template. Since the data definition involves a *union of four sub-classes*, the function template should consist of a *conditional with four clauses* according to figure 9. The template is displayed in figure 10. Each clause in the conditional compares the prefix with one of its feasible values. Each clause also contains the two other expressions that extract field values from the given structure.

Defining the second version of greeting is now straightforward. In each clause, we append the appropriate strings. Here is the full function:

```
(define (greeting a-person)
  (cond
    [(string=? (person-prefix a-person) "Ms")
     (string-append "Dear " (person-prefix a-person) " " (person-last a-person) ":")]
    [(string=? (person-prefix a-person) "Mr")
     (string-append "Dear " (person-prefix a-person) " " (person-last a-person) ":")]
    [(string=? (person-prefix a-person) "Miss")
     (string-append "Dear " (person-prefix a-person) " " (person-first a-person) ":")]
    [(string=? (person-prefix a-person) "Mstr")
     (string-append "Dear " (person-prefix a-person) " " (person-first a-person) ":")]))
```

A little bit of boolean algebra now also allows students to reduce this definition to a conditional function with two clauses:

```
(define (greeting a-person)
  (cond
    [(or (string=? (person-prefix a-person) "Ms") (string=? (person-prefix a-person) "Mr"))
     (string-append "Dear " (person-prefix a-person) " " (person-last a-person) ":")]
    [else
     (string-append "Dear " (person-prefix a-person) " " (person-first a-person) ":")]))
```

Example 3 At first glance, using a design recipe to design such simple functions like *greeting* appears to be overkill. And indeed, using design recipes for such functions is more about instilling good habits and preparing students for the design of complex programs than programmer *per se*. The use of the recipes pays off by the time students encounter self-referential data definitions. This typically happens after four to eight weeks assuming they have *no prior programming experience*, but much sooner than in ordinary courses. At that point students recognize the value of the design recipes and how they enhance their abilities.

Consider the following problem:

> **Problem** 2: Design the function *is-mary-invited?*, which determines whether "Mary" is on a list of invitees. The function consumes a list of invitees and produces true or false.

Let us assume that the data definition for lists of invitees is given:

```
;; A List-of-invitees is one of:
;; — empty
;; — (cons String List-of-invitees)
```

The definition says that a *List* is either empty or *cons*tructed from a *String* and another *List*.

In DrScheme's languages, empty is a constant like 0 or "hello world" or true. The function *cons* is built-in. It is a structure constructor like *make-person* above. The two field selector functions for a *cons* structure are *first* and *rest*. If *loi* is a *cons*tructed *List-of-invitees*, then (*first loi*) extracts the string and (*rest loi*) the list that went into the construction.

Still, the definition is *self-referential* and this is unusual. It is therefore best that we first consider some examples to ensure that the definition makes sense. Fortunately, there is at least one such list, because empty is a list according to the first clause in the data definition. From this it follows that (*cons* "Mary" empty) and (*cons* "Bob" empty) are lists. Both are *cons*tructed from the strings "Mary" and "Bob", respectively, and empty, which we know is a *List-of-invitees*. Conversely, if we look at a value such as

```
(cons "Bob" (cons "Mary" (cons "John" empty)))
```

we can determine from the data definition that this is a *List-of-invitees*.

Following the design recipe, we next write a concise abstract purpose statement for the program:

```
;; is-mary-invited? : List-of-invitees → Boolean
;; to determine whether "Mary" is on the list invitees
(define (is-mary-invited? invitees) ...)
```

Again, this step is just a reformulation of the problem statement, but it ensures that students understand *what* the function is supposed to compute.

The four examples of *List-of-invitees* also make up a natural set of examples for the i/o behavior of *is-mary-invited?*:

```
(is-mary-invited? empty)
;; should produce
false
(is-mary-invited? (cons "Mary" empty))
;; should produce
true
(is-mary-invited? (cons "Bob" empty))
;; should produce
false
(is-mary-invited? (cons "Bob" (cons "Mary" (cons "John" empty))))
;; should produce
true
```

After all, a visual inspection reveals for each example of *List-of-invitees* whether or not "Mary" is on the list.

As we move from the preliminaries to the construction of the template, the design recipe helps even more. The data definition involves three distinct elements: a union of two classes, a structure in one of the clauses, and a self-reference in the second clause. The hints in figure 9 suggest that the function template therefore consists of a conditional expression with two clauses; two selection expressions in the second clause; and a self-reference in the second clause for the *rest* expression:

```
(define (is-mary-invited? invitees)
  (cond
    [(empty? invitees) ...]
    [(cons? invitees) ...]))
```

```
(define (is-mary-invited? invitees)
  (cond
    [(empty? invitees) ...]
    [(cons? invitees)
     ... (first invitees) ...
     ... (rest invitees) ...]))
```

Fig. 11. Developing the template for *is-mary-invited*?

```
;; is-mary-invited? : List-of-invitees → Boolean
;; to determine whether "Mary" is on the list invitees
(define (is-mary-invited? invitees)
  (cond
    [(empty? invitees) ...]
    [(cons? invitees)
     ... (first invitees) ...
     ... (is-mary-invited? (rest invitees)) ...]))
```

The first two steps of this template design are displayed in figure 11. The figure shows how a combination of the hints in figure 9 helps students construct the program organization step by step from the data definition.

Filling the gaps in the template to obtain the full function follows a similar series of steps. First we deal with the clauses that don't involve recursive function calls. The examples show that the function should produce false for this case. Second we deal with the recursive clauses. To do that, remember that a student should write down the meaning of each expression. For the *first* and *rest* expressions, this is easy:

```
... (first invitees) ... ;; extracts the first field from invitees
... (rest invitees) ... ;; extracts the rest field from invitees
```

The trick according to figure 9 is to use the purpose statement for the recursive call and to reformulate it as a sentence:

```
(is -mary-invited? (rest invitees)) ;; determines whether "Mary" is on the rest of invitees
```

Now a teacher can convince a student that this expression should compute the correct answer for all-but-one element in *invitees*. The missing element is (*first invitees*), and for this, the function can simply compare it to "Mary", which refines the template as follows:

```
(string=? (first invitees) "Mary") ;; determine whether first is "Mary"
(is -mary-invited? (rest invitees)) ;; determine whether "Mary" is on the rest of invitees
```

If one **or** the other of these two expressions is true, "Mary" is on *invitees* and the result of (*is-mary-invited? invitees*) should be true.

The full definition of *is-mary-invited?* is just a reorganization of these thoughts:

```
;; is-mary-invited? : List-of-invitees → Boolean
;; to determine whether "Mary" is on the list invitees
(define (is-mary-invited? invitees)
  (cond
    [(empty? invitees) false]
    [(cons? invitees)
     (or (string=? (first invitees) "Mary") (is-mary-invited? (rest invitees)))]))
```

With a little practice, students can now routinely design functions for values of arbitrary size, i.e., for data definitions that involve self-references.

As a matter of fact, the design recipe scales naturally to the design of complex systems of functions for systems of mutually referential data. This in turn empowers students to design programs for deep and interesting problems after just a minimum of introduction to the language, the environment, and the design recipes.

4.2 Other Elements of the Program Design Method

Additional Design Recipes The remaining design recipes cover elements of programming based on the structural design concept of the first part:

abstraction Over the course of eight to ten weeks, the students see and design many similar data definitions and functions. After a while, they tend to demand abbreviations, at which point we introduce a design recipe for abstracting from concrete data definitions and functions. The course thus naturally introduces parameterized data definitions and higher-order functions.[10]

generative recursion While the first part of the course heavily relies on recursion, it is *not* the form of recursion that is often taught with trepidation in introductory courses. Instead, the first part uses *structural* recursion. More specifically, the structure of the program reflects the structure of the data definition, and if the data definition refers to itself, then so does the function definition.

In contrast, functions such as *quick-sort* or functions that draw fractals do *not* mirror the recursive nature of a data definition. Their recursive calls consume values that are newly generated; these values are not *structural* components of the input. For instance, contrast *insertion-sort* with *quick-sort*. The former recursively traverses the list, using the *rest* of the list for each *sort* and for each *insert* step. The latter instead partitions the given list into two separate pieces at each stage and then sorts the two lists. The algorithm chooses different partitions for different inputs; there is no fixed structural connection between the given list and the two partitions.

Because of this lack of a fixed relationship, developing a generative recursive function obviously requires some insight peculiar to the problem at hand: a "eureka step". While finding such insights is not teachable within the scope of the course, it is still possible to provide students with some guidance for the design of a generative recursion. An additional help is to teach structural recursion first; it greatly reduces fear in students of this maligned topic.

[10] In conventional object-oriented programming languages, the two concepts are implemented as generic classes and callback protocols.

accumulator-style processing Formal languages provide another way of thinking about the data definitions and program design. From this perspective the first part of the course covers data definitions that correspond to regular expressions and context-free languages. The part on generative recursion covers languages that are accepted by Turing machines. Between those two, we have context-sensitive languages. This corresponds to functions that know something about the context in which they are called.

Consider a function that consumes a list of numbers and processes it in a structural manner. When this function is applied, it doesn't know whether the given list is the entire list or just some suffix of some larger list. We can distinguish those two situations with the addition of an argument. The extra argument, dubbed an accumulator, represents some knowledge about the rest of the first (other) parameter(s). Thus, if the list-processing function is also given a number that represents the product of the entire list, it can compute the product for the entire list.

Using accumulators is a powerful programming technique. It solves a large number of problems, and in many cases it also produces a more efficient program than the standard design technique. The design recipes provide students with criteria that show when the technique is useful and how to develop programs with accumulators.

state change The last third of the course covers the design of functions that mutate the given structures. While this is a widely used technique in ordinary courses, we believe that it is also used in an *ad hoc* manner. We have observed that once programs grow to a certain size, students are left with strong doubts regarding why and how their programs work.

Our design recipes show how state mutation is an aspect of structurally or generatively recursive functions. More specifically, the design recipes explain when mutation is needed and how to add it to a program in a systematic manner. A concrete discussion is beyond the scope of this paper.

Iterative Refinement In addition to the systematic design of functions, the course also introduces guidelines for the design of complete programs. We equate a complete program with a system of cooperating functions. Our course teaches the design of such a system as an iterative refinement process.

For each step in the iterative refinement process, students state the central purpose and then identify the important forms of data. The goal of the initial step is to represent the important pieces of information as data and to choose some pieces that are intentionally ignored. For example, students may model a file system with names for files and with lists for folders. The goal of subsequent steps is to refine this data representation, say to change the representation of files to include a size and of folders to include a name.

Based on the data representation at each stage, students also determine which functions they can implement. Of course, each refinement stage will allow them to write a more complex system of functions than the previous one. Furthermore, they will also find that the design of a single function requires the use of functions that don't exist yet. They formulate such functions as "wishes" and place them on a "wishlist" before they continue. If, for example, they discover that combining the available information from a template cannot be done with a built-in function, they write down a purpose statement and assume they have the function. They process the remaining wishlist in a bottom up fashion, i.e., starting with functions that don't (seem to) need other help functions. Students thus construct complicated programs early and in a systematic manner.

Questions and Answers Finally, it is our belief that the goal of the course is to empower students to solve problems and to design programs systematically on their own. To achieve this goal, we train teachers to bring the design recipes across as a question-and-answer game. It is natural to translate each stage of the design recipe into a short series of scripted questions. If teachers repeatedly use those questions, students recognize that they can pose those questions themselves and thus learn to tackle problems with questions. This is just a well-established teaching technique, but it is particularly easy to implement with the design recipes.

5 Preliminary Evaluation Results

Over the past few years, we have trained over 200 teachers in the use of the program design method and the software. The training sessions take place in the summer with workshops of 10 to 40 teachers at several sites in the US. We also consult with teachers over email and on a project-focused mailing list. We also train a select group of teachers as master teachers, who enrich the workshops by providing the viewpoint of teachers who have previously attended the workshops themselves and have since taught this material in courses similar to the ones the workshop attendees will be teaching.

The project has been continuously evaluated by two independent consultants: Leslie Miller (Houston), for the first two years, and Roger Blumberg (Providence), for the past two years. Their evaluation efforts focus on two aspects of the project: the training of the teachers and the effect of the curriculum on the students of those teachers who are allowed to implement the curriculum.

The results from the workshop evaluations are outstanding. Approximately 98% of the participants complete the workshops successfully. Of those, 90% believe that the workshops fundamentally alters their view of the introductory course on computing and programming. Many state that the workshop has also renewed their enthusiasm for teaching because the design recipes clarify how such a course can help students with problem solving outside of computing. Furthermore, by providing a rigorous curriculum, our course helps the teachers justify that computing is not just a peripheral vocational subject but deserves to be a core component in the school curriculum.

The preliminary results of the student evaluations are equally encouraging. Students seem to cope well with Scheme's parenthetical syntax, which is a common objection from outsiders. Much more important, however, first questionnaires with students suggested that female students prefer our course to a traditional course by a factor of four to one (4:1). In a controlled experiment, a trained teacher 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 stunning. Our second evaluator is now investigating this aspect of the project in more depth.

The curriculum has also been noticed by third parties. CORD [4] adopted it for the introductory course of their "Academy of Information Technology" project. By 2005, some

300 schools will offer an "academy" program as a school-within-a-school. The state of Tennessee, USA, has adapted our material for its manufacturing technology curriculum [14]. Students now learn to proceed in vocational manufacturing technology courses along the lines of the investigators' design recipes.

6 Related Work

Our work has two components: the program design method and support software for the introductory course. Here we compare our efforts to some other obvious counterparts in the literature.

At first glance, the course is a close cousin to Abelson and Sussman's *Structure and Interpretation of Computer Programs* (SICP) [1]. Both approaches use Scheme; both courses teach more than the syntax of the currently fashionable programming language (Pascal, C/C++, Java, or some flavor of Basic). These superficial similarities are, however, deceiving. TeachScheme! is not just a version of SICP for high schools. While the latter uses Scheme as a sketchpad to introduce students to a broad spectrum of topics from computer science, the emphasis of the TeachScheme! project is the systematic design of programs. The TeachScheme! project also differs from SICP in that it comes with a full-fledged suite of tools tailored to its curriculum.

Still, Scheme and functional programming in general are a deep inspiration for the TeachScheme! project's program design method. Books such as SICP and Bird and Wadler's *Introduction to Functional Programming* [3] have advocated a datatype driven programming style. Nobody has come as close as Glaser et alii [7] in their work on teaching functional programming at the introductory college level. Simultaneously to the TeachScheme! project, they developed the idea of *programming by numbers*. Their idea vaguely corresponds to figure 9; they did not discover or use any of the other elements of our program design method.

Outside of computing, Poyla's [12] work on mathematical problem solving was an additional inspiration. While Polya does not spell out the ideas of data-driven programming, his step-wise approach to problems is similar to that of our general design recipe (figure 7).

Concerning the software tools, DrScheme was one of the first PDEs developed specifically for beginners. From the technical side, DrScheme inherits several ideas from the Emacs editor [8, 13], but pedagogically, DrScheme is an attempt to tame Emacs for beginners. In the meantime, the Java community has recognized the value of such environments. The BlueJ PDE [2] for Java beginners attempts to eliminate as many syntactic obstacles from a Java introductory course as possible. Using BlueJ, students can invoked methods directly, without writing a `main` function, and they can design programs using a diagrammatic approach. Still, BlueJ doesn't even come close to DrScheme. It misses several of DrScheme's major features including the stepper, the test suite manager, and the restriction of Java to a (or several) subset(s) teachable in a high school course. We believe that if PDEs wish to deal with beginners in an appropriate way such restrictions are necessary to help students overcome the horrendous error messages that currently drive people away.

7 Context

High school computing is in need of reform. The approaches of the past have produced unacceptable results. Too few students take courses on programming and computing; too few of those enrolled take away a sense of how computing and programming can help in other areas; and in the US, the existing courses and curricula clearly deter many from experimenting with the subject subsequently in college. Writing another old-fashioned curriculum for the next fashionable programming language simply won't improve this situation. We need to take a look at all aspects of the course.

The TeachScheme! project is such a reform effort. In this paper, we have shown how it tackles three sources of persistent problems with high school level courses. First, it uses a well-specified process to teach program design and problem solving in a systematic manner. Second, it uses a series of well-tailored languages to represent what beginners know at certain stages in the curriculum. Third, the PDE uses these languages to provide feedback that is appropriate to the student's knowledge level. We believe that all three innovations are major improvements over existing practices, and deserve serious consideration from future course developers.

In the future, we will focus our efforts on the creation of a bridge between this first course and the traditional follow-up curriculum. If students get seriously interested in programming after such a first course, they need to study concepts that help them with first industry experiences in internships and co-op jobs. Furthermore, while our course introduces and uses the notion of classes, it does not teach class-based programming in the spirit of C# or Java. To bridge the gap between our course and conventional CS 2 courses, and to show students that our design method applies in an object-oriented world, we intend to adapt the design method for such languages and to create a PDE that introduces students to these topics in a graceful manner.

Acknowledgments : The authors acknowledge Kathi Fisler for teaching numerous teacher training workshops; John Clements for producing the stepper; and the rest of PLT for assisting with the creation of DrScheme and the summer training courses.

References

1. Abelson, H., G. J. Sussman and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.
2. Barnes, D. J. and M. Koelling. *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall, 2003.
3. Bird, R. and P. Wadler. *Introduction to Functional Programming*. Prentice Hall International, New York, 1988.
4. CORD. Academy of Information Technology.
https://www.cord.org/lev2.cfm/93.
5. Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. In *Theoretical Computer Science*, pages 235–271, 1992.

6. 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.
7. Glaser, H., P. H. Hartel and P. W. Garratt. Programming by numbers – a programming method for complete novices. *Computer Journal*, 43(4):252–265, 2000.
8. Gosling, James. *The Emacs Screen Editor*. Unipress Software Inc., 1984.
9. Graham, P. Hackers and painters. http://www.paulgraham.com/hp.html.
10. Hoare, C. Hints on programming language design. In Bunyan, C., editor, *Computer Systems Reliability*, pages 505–534. Pergamon Press, 1974.
11. Kelsey, R., W. Clinger and J. Rees (Editors). Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
12. Polya, G. *How to Solve It*. Princeton University Press, 1945.
13. Stallman, R. EMACS: the extensible, customizable, self-documenting display editor. In *Proc. ACM SIGPLAN/SIGOA Symposium on Text Manipulation (SIGPLAN Notices)*, pages 147–156, 1981.
14. Tennessee Education Department. Tennessee manufacturing cluster curriculum standards. http://www.state.tn.us/education/vetimanufacclusteredcourses.htm.