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 programming courses in secondary schools. First, we use a design method that asks students to develop programs in a stepwise fashion such that each step produces a well-specified intermediate product. Second, we use an entire series of programming languages, not just one. Each element of the series introduce students to specific linguistic mechanisms and thus represents a cognitive development stage in the learning process. The third reform element is the use of a program development environment that was specifically developed for beginners. This paper presents the project's premises, the details of its innovations, and a preliminary experience report.

1 Programming for Everyone

A good course on programming belongs in the core of every secondary school's liberal arts curriculum. Good courses on programming introduce students to the systematic design of programs based on word problems. A student who learns to design programs systematically also learns (and practice) critical reading, analytic thinking, concise writing, problem solving, and fact checking, all of which are generally useful skills.

Programming also has two major motivational advantages over similar school subjects. First, programming is creative. With programs, students create artifacts that do something, unlike the solutions to mathematical exercises or English essays. Second, programming is an activity with an extremely fast feedback loop, which is a key factor in active learning, a style of learning that appeals to a majority of learners [6, 20].

Unfortunately, secondary schools fail to exploit programming on a large scale. For an indication of how few students US high schools engage in programming, consider the following table from the Educational Testing Service's Advanced Placement (AP) Yearbook [19]:

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 as take the CS test; even statistics is more than twice as popular as computer science. Considering that every one of the students in mathematics should also succeed in a programming course, these statistics say that conventional secondary school courses on programming are failures.

After studying existing curricula and observing existing AP programming courses, we have come to believe that the key problem is a prevailing but outdated view of programming as a vocational activity. Secondary school educators and administrator simply don't understand the power of programming and its potential role in the core of a liberal arts curriculum. This attitude strongly affects the content of the courses.[5] First, secondary schools often let the grammar of currently fashionable, vocational programming languages dictate their curriculum rather than sound principles of design and problem solving. Second, schools employ program development technology that is intended—and work best—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 inexpensive versions of the professional products. Third, investing energy into the study of complex grammars and programming environments distracts teachers and students from the true nature of programming. Instead of systematic design, students are often exposed to a tinker-until-it-works philosophy, which does not teach any of the desirable skills mentioned above.

Our TeachScheme! project, an initiative that we started in 1996, aims to move programming courses into the core of secondary school curricula. To achieve this goal, we have worked on three specific innovations:

1. First, we have designed and implemented a series of programming languages with a matching introductory curriculum. Each element of the series represents a cognitive stage in the learning process.
2. Second, we have also developed a matching IDE. The environment assists students with the design of programs and helps them understand language concepts.
3. Third, we have created a program design method for beginning students and their teachers. Students who follow the design method produce several well-specified intermediate products in addition to the final program. A teacher can use the intermediate products to help a student overcome problems during the design or to discover logical mistakes in the final product. Furthermore, a teacher can use the design method to justify grading standards that evaluate structural or aesthetic aspects of the final product in an absolute manner, something that is nearly impossible with any other design method.

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

In this paper, we report on the TeachScheme! project. In the next four sections, we describe our premises and the primary improvements over the existing computing and programming curricula. In the sixth section, we present a preliminary evaluation report. In the seventh section, we discuss how the curriculum fits into an existing context. The final section summarizes our experiences and presents some plans for the future.

[5] Indeed, in many places "introduction to computer science" is a course on application software and has no programming at all.

2 The Central Role of Mistakes

To understand how beginners work with programming languages and programming environments, we started our project with comprehensive observation sessions in our own lab (at Rice University) and in local high schools (Houston). These observations quickly showed that *all* beginners have problems with the notational conventions of *all* programming languages (C++, Pascal, and Scheme) and programming environments.

Here is a scenario that we repeatedly observed during the first two weeks of AP courses. One of the first problems that a teacher poses to students is to write a program that determines the total cost of some order from the number of ordered pieces and their price. A student's C++ program then contains 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 at this point in the course. Eventually some helpful classmate suggests to swap the two sides of the "equation", and the student then happily compiles her program—without understanding why.

In general, we found that above all,

beginners make 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 methods and tools that help students recognize and overcome errors.

Based on this insight, we identified distinct phases in the learning process and specified what students should know and *not* know in these phases. Then we used this decision to co-design a programming language for each phase, a supportive programming environment, and a program design method. In each case, we paid special attention to provide help in error cases for both students and teachers. The following three sections present these three elements in a sequential order.

3 The Programming Language

The explicit staging of the learning process has deep implications for the programming language that is used. After all, a programming language represents in the most concrete manner what a student knows and doesn't know about programming. To accommodate the staging, we chose to design and implement a *series of programming languages* based on Scheme [15] for our reformed course.

Figure 1 summarizes the three programming subsets of Scheme in our curriculum.[6] Let us consider each phase in turn:

[6] 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

beginners The goal for beginning students is to learn how to use built-in functions and how to design functions on their own. Hence, the first language is a purely functional language over numbers, strings, booleans, and (programmer-defined) structures. The entire language consists of four expression forms (function application, `cond`itional expressions, variables, and literal values) and two definition forms (function definitions and structure definitions).

intermediate The second language level introduces a construct for structuring programs (`local` definitions) and another for abstracting over common patterns (first-class functions). Students recognize the need for both because the first language forces them to write many similar programs.

advanced The last language is for advanced students in the course. The goal for those students is to master the notoriously difficult notion of state in programs [13]. 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.

We start with the purely functional subset of Scheme so that the programming course is easily integrated with an algebra course. A functional language is conceptually just a generalization of algebra. Specifically, the chosen language is based on prefix notation, and its functions work on many different kinds of values, not just numbers.

```
(define (convert-f-c f)
  (* 5/9 (- f 32)))
```

What (and why) is the value of:

```
  (convert-f-c 32)
= (* 5/9 (- f 32))
= (* 5/9 0)
= 0
```

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

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

What (and why) is the value of:

```
  (greet (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

Consider the example in figure 2. Both columns depict programs that our students tend to write after a few lessons. The program on the left converts Fahrenheit 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, while 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.

Since students have usually taken an algebra course before the programming course, they already know how to evaluate numeric expressions. From there, they can easily acquire the additional machinery that explains what happens when a program, i.e., a function, is applied to input values and especially when a program goes wrong. That is, students already know how "the hardware" works. As a result, teachers can spend more time on teaching design and problem-solving principles and less on the physical principles of computation. Conversely, the programming course reinforces the algebra skills that they already have.[7]

Concerning the programming mechanics, teaching a small *enforced* subset of any language, not just Scheme, has two advantages. First, the language dictates what students are and are *not* allowed to use. Hence, teachers can focus on the development of problem solving and program design skills. In particular, they no longer have to get into (usually

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

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, an implementation of full Scheme would print an error message concerning higher-order functions that a beginner can't possibly know.

4 The Program Development Environment

A programming language needs a program development environment (PDE). Roughly speaking, a PDE helps programmers with common tasks. 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 large tool suites just confuse beginners. To write 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 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 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 [9]. Here we focus on the use of DrScheme with our curriculum.

Figure 3 displays a screenshot of DrScheme. The basic PDE consists of just three panes: a toolbar with five visible buttons; an editor; and an interactive evaluator for the chosen teaching language at the bottom. The evaluator is an extremely powerful calculator. In general, it determines the value of any valid Scheme expression. 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).

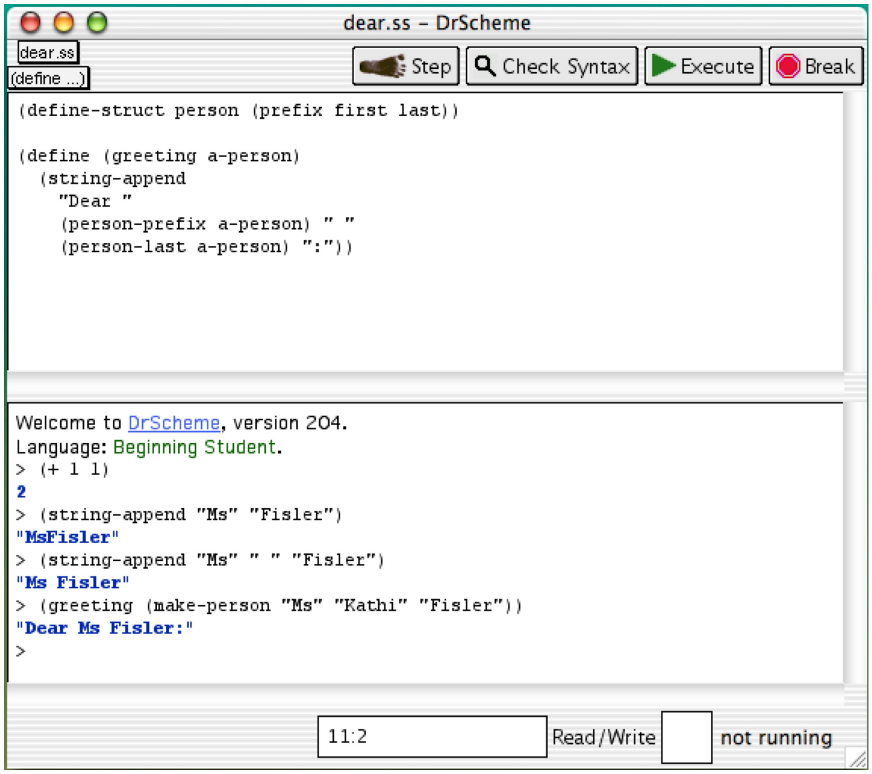

Fig. 3. DrScheme

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 complete expressions in grey as the student types. This minimizes confusion with parentheses and, in our experience, eliminates any discussions about syntax in less than a week.

DrScheme toolbar displays only five, carefully chosen buttons:

Execute evaluates the definitions 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;
Check Syntax analyzes the syntax and the scope of the program in the editor (see below);
Step allows students to step through the algebraic evaluation of an expression.

Besides the editor and the evaluator, the algebraic stepper is the most important tool that beginners use. The algebraic stepper displays the sequence of evaluation steps shown in figure 2 for expressions such as

```
(greet (make-person "Ms" "Kathi" "Fisler"))
```

Using the stepper, a teacher can easily explain the computations that take place when a program is applied to input values.

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 to understand the syntax of the language and the idea of substituting equals for equals. The stepper 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.

Like other PDEs, DrScheme also comes with a syntax coloring tool. The tool 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 variables 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 such as `cost`, the syntax checker renames that one `x` without affecting any others automatically—in contrast to conventional search-and-replace tools.

The final component of DrScheme is the test suite manager. Working with examples and testing are key elements of our program design method. The testing tool (currently in a separate window) provides an explicit space for writing down sample function calls and their expected values. When a student clicks on **Execute**, DrScheme compares the actual values with the expected values; mismatches indicate logical errors in the student's reasoning.

We have also developed a coverage analysis tool. When both pieces of the test suite manager are fully integrated, 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.

5 The Program Design Method

Like the programming language and the programming environment, the program design method in an introductory course must accommodate beginners and must grow with them. This suggests three concrete goals. 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 through intermediate results, 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.

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 a student requests help from a teacher, 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 subtract for some incorrect

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 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)	fill the holes in the template, using the purpose statement, the data structure of the data definition, and the examples
testing	test suite & coverage	look for mistakes using the examples from step 2; mistakes can show up in the examples, the program, and/or both

Fig. 4. The generic program design recipe

statement in a program. Instead a teacher can inspect the process that produced the program and grade a 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.

Figure 4 describes the generic design recipe. Each specific design recipe covers a particular kind of data definition, i.e., the rigorous description of a class of values. The course starts with simple data definitions and quickly proceeds to recursive and mutually related definitions. The shape of the data definition induces a specific shape for a function that processes this kind of data, as specified in steps 4 and 5 of the generic design recipe. Let us illustrate the use of the design recipe with two examples.

Example 1 Recall the function `greet` from figure 3. Here is a problem statement that might produce this function:

> **Problem** 1: Design the function `greet`, 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 components. 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 structure definition and a method for using the structure.

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

```
;; greet : PERSON -> STRING
;; to produce a letter greeting from a-pers
(define (greet a-pers) ...)
```

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 have 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 examples for `greet` from these examples:

```
(greet (make-person "Ms" "Kathi" "Fisler"))
;; should evaluate to
"Dear Ms Fisler:"

(greet (make-person "Mr" "John" "Clements"))
;; should evaluate to
"Dear Mr Clements:"
```

Each example consists of a complete function call and the 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 across functions with the same domain. Roughly speaking, the template is a translation of the data definition for the inputs into a program fragment. Since the functions (at this stage) can only produce information from the information that they consume, basing the structure

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. 5. Construction templates and programs from data definitions

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.

Figure 5 displays some basic hints on how to proceed at this point. 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 this:

```
;; greet : PERSON -> STRING
;; to produce a letter greeting from a-pers
(define (greet a-pers)
  ... (person-prefix a-pers) ...
  ... (person-first a-pers) ...
  ... (person-last a-pers) ...)
```

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

Now, and only now, are students 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 3 shows the final result (without comments and tests).

Example 2 At first glance, using a design recipe to design such simple functions like `greet` 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 programming *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, even if they have *no prior programming experience*. 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-OF-INVITEES` 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)` extracts 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 "Jon" empty)))
```

we can determine from the data definition that this is a `LIST-OF-INVITEES`.

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

```
;; is-mary-invited? : LIST-OF-INVITEES -> BOOLEAN
;; to determine whether "Mary" is on the list of 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 behavior of `is-mary-invited?`:

```
(is-mary-invited? empty)
;; should evaluate to
false

(is-mary-invited? (cons "Mary" empty))
;; should evaluate to
true
```

```
(is-mary-invited? (cons "Bob" empty))
;; should evaluate to
false

(is-mary-invited? (cons "Bob" (cons "Mary" (cons "Jon" empty))))
;; should evaluate to
true
```

After all, a visual inspection reveals for each example whether `"Mary"` is on the list.

Question: *How many clauses are in the data definition? How many* `cond` *clauses do we need in the function?* (See row 3 in figure 5)

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

Question: *Is* `empty` *a constant or a structure? Is* `(cons String L)` *a constant or a structure? How many selector expressions do we need in each clause?* (See row 2 in figure 5)

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

Fig. 6. Developing the template for *is-mary-invited*?

As we move from the preliminaries to the construction of the template, the design recipe helps a lot. 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 5 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:

```
;; 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 6. The figure shows how a combination of the hints in figure 5 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 person
... (rest invitees) ... ;; extracts the rest of the invitees
```

The trick according to figure 5 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 in the rest of the list
```

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") ;; is first equal to "Mary"
(is -mary-invited? (rest invitees))
;; determines whether "Mary" is in the rest of the list
```

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 definitions. 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.

In addition to the structural design recipes, the curriculum for an introductory college course would also cover design recipes for abstraction over common patterns; for generative (as opposed to structural) recursion; for context-sensitive recursive functions (accumulator style); and for functions that can change the state of the world. As students learn more and different ways to design programs, the curriculum can also discuss alternative designs, trade-offs among designs, and global design strategies such as iterative refinement. We refer the interested reader to our book [7].

6 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 during the summer with workshops of 10 to 40 teachers at several sites in the US. 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 and have taught with the new curriculum in high schools.

The project has been continuously evaluated by two independent consultants: Leslie Miller (Houston), for the first three 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 improve students' general problem solving skills. 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, student questionnaires 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 a majority of students preferred our approach to programming, the four-to-one preference for TeachScheme! 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 [5] 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 [18]. Students now learn to proceed in vocational manufacturing technology courses along the lines of the investigators' design recipes.

7 Related Work

Our work has three components: the program design method, a series of languages, and support software. 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 secondary 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 tailor-made tools. For a more detailed comparison, we refer the reader to a companion paper [8].

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* [4] have advocated a datatype-driven programming style. Nobody has come as close as Glaser et al. [10] to our design recipe idea. They developed the idea of *programming by numbers* concurrently to our work. Their idea vaguely corresponds to figure 5. They did not discover or use any of the other elements of our program design method.

Outside of computing, Poyla's [16] 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 4).

With respect to software tools, DrScheme was the first PDE developed specifically for beginners. From the technical side, DrScheme inherits several ideas from the Emacs editor [11, 17], but DrScheme is an attempt to tame Emacs for beginners. The idea of introducing a hierarchy of sublanguages—one of most critical elements of taming—is a rediscovery, originally due to Holt et al. [14]. They faced the challenge of teaching PL/1 to beginners; in response, they defined a series of sublanguages, called SP/k, that introduce the concepts of imperative programming in a gradual manner. Unlike DrScheme, SP/k didn't come with a complete program development environment, nor did the language implementation distinguish the various language levels [14, page 307]; in particular, the SP/k compiler did thus not tune error messages to the knowledge level of a learner.

In the meantime, the Java community has recognized the value of special environments for novices. The BlueJ PDE [3] and DrJava[8] [2] environments for Java beginners attempt to eliminate as many syntactic obstacles from a Java introductory course as possible. Using BlueJ or DrJava, students can invoked methods directly, without writing a `main` function; in BlueJ, they can also design programs using a diagrammatic approach. Still, neither BlueJ nor DrJava come even close to DrScheme. They miss 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 secondary school course. We believe that if PDEs wish to deal with

[8] Paul Graunke, a PhD student of Felleisen and Krishnamurthi, designed and implemented the first release of DrJava, which is why it was named using the TeachScheme! project conventions.

beginners in an appropriate way such restrictions are necessary to help students overcome the horrendous error messages that currently drive people away from programming. To inject this idea, our team is in the process of developing ProfessorJ, a Java environment in the true image of DrScheme [12].

8 Context

Programming courses at secondary schools are in serious 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 programming can help them solve problems; and in the US, the existing courses and curricula clearly deter many from experimenting with the subject 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 these courses.

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 carefully tailored languages to represent what beginners know at each stage in the curriculum. Third, the PDE uses these languages to provide feedback that is appropriate to a 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 currently fashionable languages, such as 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 many teacher training workshops and for her enthusiastic support of the project at all stages; John Clements for producing the stepper; and the rest of PLT for assisting with the creation of DrScheme and all the additional work.

Note For more information on the TeachScheme! project, please visit

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

The original submission for this article is available at

`http://www.ccs.neu.edu/scheme/pubs/`

Due to space limitations, this published paper is a highly abridged version of the submission.

References

1. Abelson, H., G. J. Sussman and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.
2. Allen, E., R. Cartwright and B. Stoller. DrJava: A lightweight pedagogic environment for Java. *SIGCSE Bulletin and Proceedings*, 34(1):137–141, 2002.
3. Barnes, D. J. and M. Koelling. *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall, 2003.
4. Bird, R. and P. Wadler. *Introduction to Functional Programming*. Prentice Hall International, New York, 1988.
5. CORD. Academy of Information Technology. https://www.cord.org/lev2.cfm/93.
6. Felder, R. M. and R. Brent. Cooperative learning in technical courses: Procedures, pitfalls, and payoffs. Technical Report ED 377038, ERIC Document Reproduction Service, 1994.
7. Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.
8. Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. The structure and interpretation of the computer science curriculum. In Hanus, M., S. Krishnamurthi and S. Thompson, editors, *Functional and Declarative Programming in Education*, 2002.
9. 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.
10. 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.
11. Gosling, James. *The Emacs Screen Editor*. Unipress Software Inc., 1984.
12. Gray, K. E. and M. Flatt. ProfessorJ: A gradual intro to Java through language levels. In *OOPSLA Educators' Symposium*, 2003.
13. Hoare, C. Hints on programming language design. In Bunyan, C., editor, *Computer Systems Reliability*, pages 505–534. Pergamon Press, 1974.
14. Holt, R., D. Wortman, D. Barnard and J. Cordy. Sp/k: A system for teaching computer programming. *Communications of the ACM*, 20(5):301–309, May 1977.
15. Kelsey, R., W. Clinger and J. Rees (Editors). Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
16. Polya, G. *How to Solve It*. Princeton University Press, 1945.
17. 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.
18. Tennessee Education Department. Tennessee manufacturing cluster curriculum standards. http://www.state.tn.us/education/vetimanufacclusteredcourses.htm.
19. The College Board. *AP Yearbook*. The College Board, 2002.
20. Tobias, S. *Revitalizing Undergraduate Science: Why some things work and most don't*. Research Corporation, 1992.