Measuring the Effectiveness of Error Messages Designed for Novice Programmers

Guillaume Marceau
WPI
100 Institute Road
Worcester, MA, USA
+1 (508) 831-5357
gmarceau@wpi.edu

Kathi Fisler
WPI
100 Institute Road
Worcester, MA, USA
+1 (508) 831-5357
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
115 Waterman St
Providence, RI, USA
+1 (401) 863-7600
sk@cs.brown.edu

ABSTRACT

Good error messages are critical for novice programmers. Many projects attempt to rewrite expert-level error messages in terms suitable for novices. DrScheme's language levels provide a powerful alternative through which error messages are customized to pedagogically-inspired language subsets. Despite this, many novices still struggle to work effectively with DrScheme's error messages. To better understand why, we have begun using human-factors research methods to explore the effectiveness of DrScheme's error messages. Unlike existing work in this area, we study messages at a fine-grained level by analyzing the edits students make in response to various classes of errors. Our results point to several shortcomings in DrScheme's current treatment of errors; many of these should apply to other languages. This paper describes our methodology, presents initial findings, and recommends new approaches to presenting errors to novices.

Keywords

Error message design, Novice programmers, User-studies

1. INTRODUCTION

In a compiler or programming environment, error messages are arguably the most important point of contact between the system and the programmer. This is all the more critical in tools for novice programmers, who lack the experience to decipher a poorly-constructed error message. Indeed, many research efforts have sought to make professional compilers more suitable for teaching by rewriting their error messages [16] or by supplementing them with hints and explanations [6]. Such efforts complement more general research on improving error messages through techniques such as error recovery during parsing.

DrScheme[1] [10] reflects a philosophy that programming languages designed for experts cannot be shoehorned into a teaching role. Programming courses teach only a few constructs of a full language; at any time, students have seen only a fragment of the full language. This creates a mismatch between the programming language that the students believe they are using—the subset that they are aware of—and the language the compiler processes. Students experience this mismatch in two ways: (1) when they use an advanced construct by mistake and their program does not fail, but instead behaves in a weird way; and (2) when their mistakes are explained by the error message in terms of concepts they do not yet know.

To address this issue, DrScheme offers several *language levels* [15]. Each level is a subset of the next level up. As the course progresses, students move through five language levels, from Beginner Student Language (BSL) to Advanced (ASL). Each level's error messages describe problems by referring only to concepts the student has learned so far. The levels also rule out programs that would be legal in more advanced levels; as a corollary, errors are not preserved as students move up the chain. Figure 1 illustrates the impact of switching levels on the messages. Running program (a) in BSL results in the error message "*define: expected at least one argument name after the function name, but found none*". The same program runs without errors in ASL, since once students reach ASL they have learned about side effects, at which point it makes sense to define a function without arguments; this illustrates point (1). Similarly, running program (b) in ASL does not raise an error, since placing a variable in function position is not a mistake for students who have been taught first-class functions; this illustrates point (2).

The DrScheme error messages were developed through well over a decade of extensive observation in lab, class, and office settings. Despite this care, we still see novice Scheme programmers struggle to work effectively with these messages. We therefore set out to quantify the problem through finer-grained studies of the error messages as a feedback mechanism, following HCI and social science methods [33]. Specifically, we set out to understand how students respond to individual error messages and to determine whether some messages cause students more problems than others. Over the longer term, we hope to develop metrics for good error messages and recommendations for developers of pedagogical IDEs that generalize beyond Scheme.

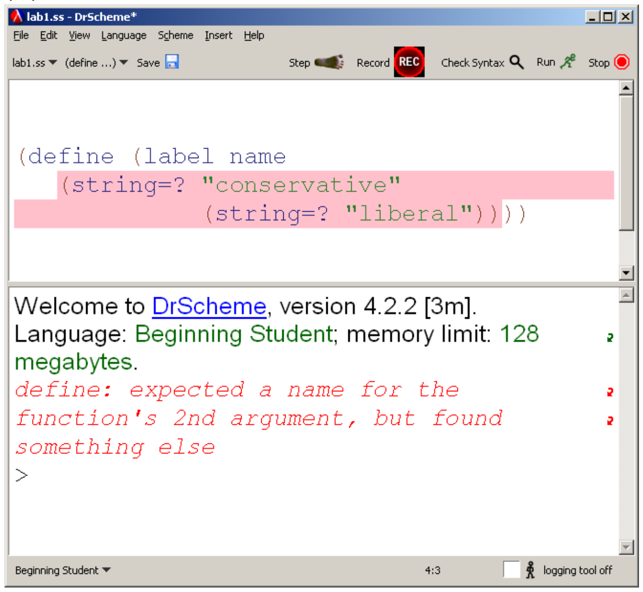
(a)
```
(define (add-numbers)
     (5 + 3))
```
```
define: expected at least one argument name after
the function name, but found none
```

(b)
```
(define (add-numbers x y)
       (x + y))
```
```
function call: expected a defined name or a
primitive operation name after an open parenthesis,
but found a function argument name
```

Figure 1. Not an error in ASL (a) Function without arguments (b) Variable in callee position

[1] Now known as DrRacket.

(a)


lab1.ss - DrScheme*
File Edit View Language Scheme Insert Help
lab1.ss (define ...) Save
Step Record REC Check Syntax Run Stop
Beginning Student
4:3
logging tool off

```
(define (label name
    (string=? "conservative"
              (string=? "liberal"))))
```

```
Welcome to DrScheme, version 4.2.2 [3m].
Language: Beginning Student; memory limit: 128
megabytes.
define: expected a name for the
function's 2nd argument, but found
something else
>
```

(b)

```
(define (label name
    (string=? name "conservative"
              (string=? name "liberal"))))
```

Figure 2. (a) A student's program and its error message, (b) The student's response to the error message

This paper presents results from a multi-faceted study of student interactions with DrScheme's error messages. We have looked at student edits in response to errors, interviewed students about their interpretations of error messages, and quizzed students on the vocabulary that underlies the error messages in a typical introductory college course using DrScheme. Our work is novel in using fine-grained data about edits to assess the effectiveness of individual classes of error messages. Most other work, in contrast, changes the IDE and measures student performance over an entire course. The evaluation rubric we designed, which measures the performance of error messages through edits, is a key contribution of this work. We also identify several problems in DrScheme's current error messages and recommend changes that are consistent with our observations.

To motivate the project, Section 2 gives examples of errors that students made in actual class sessions. Section 3 presents our methodology in detail. Section 4 describes the rubric and the evolution of its design. Sections 5 through 7 describe the results of our analysis thus far, while Section 8 presents initial recommendations for error message design arising from our observations. Related work appears in Section 9.

2. RESPONSES TO ERROR MESSAGES

We begin by showing a few examples of student responses to error messages during Lab #1. When Lab #1 begins, most students have not had any contact with programming beyond four hours of course lectures given in the days before and two short homeworks due the day before and evening after the lab.

```
(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(and (string=? check-for-match stringOne))]
        [(and (string=? check-for-match stringTwo))])
```

define: expected only one expression for the function body, but found at least one extra part

```
(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(string=? check-for-match stringOne)]
        [(and (string=? check-for-match stringTwo))]
        [(and (string=? check-for-match stringThree))])
```

```
(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [and ((string=? check-for-match stringOne))]
        [(and (string=? check-for-match stringTwo))]
        [(and (string=? check-for-match stringThree))])
```

```
(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(string=? check-for-match stringOne)]
        [(string=? check-for-match stringTwo)]
        [(string=? check-for-match stringThree)])
```

```
(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(string=? check-for-match)]
        [(string=? check-for-match stringTwo)]
        [(string=? check-for-match stringThree)])
```

Figure 3. A sequence of responses to an error message

Figure 2 (a) shows one function (excerpted from a larger program) submitted for execution 40 minutes after the start of the lab. The student is defining a function `label`, with one argument `name`. Most likely the student is missing a closing parenthesis after `name`, and another one after `"conservative"`. The nesting suggests that the student is struggling to remember how to combine two different Boolean tests into one using the `or` operator.

Figure 2 (b) shows the student's edit in response to that particular error message. The student inserted `name` as an argument to the function call to `string=?` . There is a logic to this response: the message says a name is expected, so the student provided a name. Beginning programmers often make this mistake (confusing a literal reference with an indirect reference). Learning to reflect with accuracy about the difference between referent, referee, and literal references is one of the skills students learn in programming courses. There is however an ambiguity in the error message that might have prompted the mistake in the response: the word "function" in the fragment "for the function's second argument" can refer to either the function being defined (`label`) or the function being called (`string=?`). DrScheme means the former, but it seemed that the student understood the latter (perhaps influenced by the highlighting). We found this kind of ambiguity common. Specifically, whenever the error messages of DrScheme use referencing phrases to point at pieces of code, it is often too vague to be understood well, and it uses technical vocabulary that impedes clarity rather than helps it. We return to this subject in Section 6.

Figure 3 shows another example. The program at the top of the figure was the first of a sequence of programs that each triggered the same error message. What follows are the student's first four attempts to correct the problem. The student never identifies the actual problem, which is a missing open parenthesis before the `cond`. The entire sequence lasts 10 minutes, until the end of the lab session. A few weeks later, the student participated in this study's interviews and mentioned how frustrating the experience had been.

Even with our years of experience teaching with DrScheme, the state of the programs we collected was often surprising, if not humbling. Students manage to create quite mangled functions, which the error messages must attempt to help them sort out.

3. METHODOLOGY

To explore how students respond to error messages, we sought a combination of data from a large number of students and in-depth data from a handful of students. In the spring of 2010, we set up a study around WPI's introductory programming course, which enrolled 140 students. Our data gathering had four components:

1. We assembled records of students' programming sessions. We configured DrScheme to save a copy of each program each student tried to run, as well as the error message received (if any) plus any keystrokes that the student pressed in response to the error message, up to their next attempt at running the program. Amongst the 140 students registered for the course, 64 agreed to participate in this data collection.

   We collected data during the course's normal lab sessions, which ran for an hour per week for six weeks (normal course length at WPI is seven weeks, so the data covers the entire course). During labs, students worked on exercises covering the last week's lecture material. We also have data from editing sessions that occurred outside the lab from 8 students who installed our monitoring software on their laptops.

2. We interviewed four students about their experience with DrScheme's error messages. These interviews helped us interpret the content of the session recordings. These students ranged from medium to good (we were not able to attract any of the weaker students). Each interview started with a short introduction in which we discussed the student's experience in the class, and his general impression of the error messages. Then we gave the student erroneous programs taken from the session recordings from Lab #1 (some his own and some from other students) and asked them to fix the proximate error mentioned in the error message. This naturally led to a discussion on the strategy the student used to respond to error messages and how the error messages could be improved.

3. During the interviews, it became apparent that students often struggle with the technical vocabulary that DrScheme uses to describe code (see Section 7). We designed a vocabulary quiz to quantify this effect. We identified 15 technical words that appear in the 90th-percentile error messages most frequently presented to students throughout the semester. Each student received a quiz with five words amongst those, and was asked to circle one instance of that vocabulary word in a short piece of code. We administered the quiz to 90 students (self-selected) at WPI. For calibration, we also administered it to Brown University undergraduates who had taken a DrScheme-based course the previous semester and to freshmen and transfer students in a programming course at Northeastern University, Boston.

4. We asked the three professors of students who participated in the vocabulary quiz to describe which vocabulary words they used in class. We received thoughtful answers from all three, indicating that they had put much effort in maintaining a consistent usage of vocabulary throughout their semester. They could say with confidence which of the 15 vocabulary word they used often, regularly, seldom, or never, in class.

To date, we have carefully analyzed only the data from the first lab week. Students' initial experiences with programming influence their attitudes towards the course and programming in general. For many students, the first week determines whether they will drop the course. Making a good first impression is critical for the success of a programming course.

4. THE DESIGN OF A CODING RUBRIC

There are many ways one might study the effectiveness of error messages. A common approach in the literature (as reviewed in Section 9) is to change the messages or their presentation and compare the impact on student grades at the end of the course. We are interested in a more fine-grained analysis that determines which error messages are effective and in what ways. There is also no single metric for "effectiveness" of an error message. Possible metrics include whether students demonstrate learning after working with messages or whether the messages help novice programmers emulate experts. We have chosen a narrower metric: does the student make a reasonable edit, as judged by an experienced instructor, in response to the error message?

We used two social science techniques to gain confidence that both our metric and its application to our data were valid. First, we developed a formal rubric for assessing each student edit. Then, we subjected the rubric to a test of *inter-coder reliability* [5] (where "coder" is the standard term for one who applies a rubric to data).[2] Inter-coder reliability tests whether a rubric can be applied objectively: multiple coders independently apply the rubric to data, then check for acceptable levels of consistency in their results. When tackling subjective topics, good inter-coder reliability can be quite difficult to achieve. After describing the evolution of our rubric, we present a standard measurement of inter-coder reliability and our high scores on this metric.

Our rubric attempts to distinguish ways in which error messages succeed or fail. Our design starts from a conceptual model of how error messages intend to help students: if an error message is effective, it is because a student reads it, can understand its meaning, and can then use the information to formulate a useful course of action. This is a three step sequence:

Read ▸ Understand ▸ Formulate

Students can get stuck at any of these steps. One interesting question is whether students get stuck earlier in the sequence with particular kinds of errors. To explore this, we would ideally like a rubric that identifies how far a student successfully went in the sequence when responding to a given error message. This would suggest a rubric with at least four categories: failure-on-read, failure-on-understand, failure-on-formulate, and fixed-the-error. Our initial attempts to distinguish failure-on-read from failure-on-understand were not successful (in that we could not achieve inter-coder reliability). Our recordings of student editing sessions lack attention-based data (such as eye-tracking) that indicate where a student looked or reacted when an error occurred; such data might have helped distinguish between read- and understand-failures. We concluded that a more realistic rubric would combine failure-on-read and failure-on-understand into a single category separate from failure-on-formulate.

[2] This paper uses "coder" exclusively as a social science term; in particular, it does not refer to programmers.

[DEL]	Deletes the problematic code wholesale.
[UNR]	Unrelated to the error message, and does not help.
[DIFF]	Unrelated to the error message, but it correctly addresses a different error or makes progress in some other way.
[PART]	Evidence that the student has understood the error message (though perhaps not wholly) and is trying to take an appropriate action (though perhaps not well).
[FIX]	Fixes the proximate error (though other cringing errors might remain).

Figure 4. Rubric for responses to error messages

Figure 4 presents our final rubric for assessing students' edits. The [UNR] and [PART] codes capture failure-on-read/understand and failure-on-formulate, respectively. All the responses in the sequence shown in Figure 3 were coded [UNR], for example, since none of the edits tried to change the number of parts in the function body position of the `define`, and nothing else suggested that the student had read or understood the message.

Earlier versions of our rubric attempted to discern two nuances of failure-on-understand: failure to understand the text as separate from failure to understand what the message *really* means in terms of the code. An error message can use simple words and simple grammar but still be hard to understand because the underlying problem is difficult or because the message inadequately describes the problem. Responding to these error messages requires students to read beyond the words and understand that "when DrScheme says X, it really means Y". Figure 5 shows an example. On its face, the message contradicts the text of the code: there definitely is a parenthesis before the `and`. To understand the message, one has to realize that the parenthesis before the `and` has been attributed to the `cond;` in the parser's view, the `and` stands on its own without a parenthesis. Predictably, the student failed to formulate a useful response to that message (they deleted the parenthesis before the `and`). Early versions of the rubric tried to capture how often students failed to formulate a response according to the deep meaning of the message (what an expert would understand from the message) because they were being misled by its literal meaning. However, coders were not sufficiently reliable when making these distinctions, and so the final rubric has only one code corresponding to a failure to formulate, namely [PART].

For the remaining codes in Figure 4, [DEL] captures cases when students simply deleted error-inducing code rather than attempting to fix it, [DIFF] captures edits that were useful but unrelated to the reported error (such as fixing a different error or adding more code or test cases), and [FIX] captures successful completion of the read/understand/formulate sequence. These codes and their precise wordings reflect several design decisions that arose while developing the rubric:

- **The rubric should assess the performance of the error messages, not the students.** Consider a situation in which a student's edit corrects a problem that had nothing to do with the original error message. While this is a positive outcome, it does not address our primary concern of how effective error messages are at guiding students through the read/understand/ formulate sequence. Similarly, students may experience difficulties with problem solving or program design that should not be attributed to shortcomings of the error messages. To keep our coding focused on the error messages, we include the [DIFF] code for reasonable edits unrelated to the proximate error. Unreasonable edits unrelated to the proximate error are coded [UNR]. Our first rubric design had unified [DIFF] and [UNR]; we split them after considering when the error message could be held accountable. Sometimes, students simply avoid the proximate error by deleting their code (for example, deleting a test case that yields an error). To avoid judging the error message (as [UNR] might), we introduced the separate [DEL] code for such cases. When deletion is the appropriate action (such as when removing an extra function argument) and it is performed on a reasonable code fragment, we code it as [PART] or [FIX] as appropriate. Together, [DIFF] and [DEL] attempt to characterize situations in which the student's action provides no information about the quality of the error message.

- **Coding decisions have to be made narrowly,** strictly in relation to the proximate error described in the message. DrScheme's error messages always describe one particular problem, regardless of other problems that might be present. Fixing the problem mentioned in the message sometimes makes the overall code worse (for example, a student might delete an extra expression rather than add an operator to combine it with the rest of the code). Frequently a student's edit fixes the error mentioned, while leaving other glaring errors in surrounding code untouched. We nevertheless code such edits as [FIX]. The code [FIX] does not imply mastery on the part of the student, nor does it imply oracle-like accuracy on the part of the message. Rather, [FIX] means that the student formulated a reasonable response to the problem mentioned in the message. If the student is as myopic as the error message, but no more, they may still receive the code [FIX]. The text "though other cringing errors might remain" in the [FIX] case remind the coders to take this narrow interpretation. In practice, we found that each coder needed that reminder explicit in the rubric in order to be self-consistent in their use of [FIX].

- **Coding needs a holistic view of multi-faceted error messages.** DrScheme's error messages have two components: text and a highlight. In assessing whether a student had "read" or "understood" an error message, we had to decide whether it sufficed for students to edit within the highlight component, even if their action showed no evidence of considering the text component. As we discuss in Section 6, some students come to glance first at the highlight for a quick overview of the error; this should be a credit to the error message, even though we have a bias towards the text when assessing "understanding". At the same time, students often made random edits in the highlighted code that were arguably unrelated to the proximate error. We ultimately decided that location was not sufficient justification for ascribing [PART] or [FIX].

As computer scientists, not social scientists, we sometimes found the subjective nature of coding uncomfortable, but ultimately more successful than decomposing all observations into purely objective observations. For example, we accepted liberally any evidence that the student read and understood something from the message. In some cases, making this determination required

```
(define (label-near? name bias word1 word2 word3)
  (cond
    (and (cond [(string=? name word1) "Name Located"]
               [(string=? bias word1) "Bias Located"])
         (cond [(string=? name word2) "Name Located"]
               [(string=? bias word2) "Bias Located"])
  "Mark")
))

and: found a use of `and' that does not follow an
open parenthesis
```

Figure 5. A counterfactual error message

human judgment or teaching experience, as was the case with the "expect a name" example in Figure 2. Because we decided that the student probably got the idea of inserting "name" from having read the words "expected a name" in the message, we coded that response [PART] rather than [UNR]. We found such subjective decisions surprisingly consistent across the coders.

During the design process, we also ruled out ideas that failed to survive inter-coder reliability tests or our own evaluation:

- *Distinguishing [FIX] codes based on elapsed time*: we considered factoring in students' response time by having separate codes for "fixed with hesitation" and "fixed without hesitation" (we have timestamp data on all edits, and can replay editing sessions at their original pace). In theory, errors to which students respond more slowly might be harder for students to process. We ultimately ruled this out for two main reasons. First, response time could be affected by corrupting interferences (such as a student taking a bathroom break or differences in working styles across students). Second, we lacked a good metric for the expected difficulty of each error message; without that, we would not be able to identify messages that were performing worse than expected.
- *Considering whether the edit yielded a new error message as a criterion for [FIX]:* this is a corollary to our observation about coding narrowly. In practice, we found cases in which the student really did fix the error, but had code of such a form that the same error applied after the edit. We chose to ignore this criterion in final coding.

The rubric as shown in Figure 4 meets standards of inter-coder reliability on the data from Lab #1. We used the standard metric of inter-coder reliability [5], κ, which is defined as

$$\kappa = \frac{\text{Agreement} - \text{Expected Agreement}}{1 - \text{Expected Agreement}}$$

κ compares the agreement of the human coders to the agreement that would be expected by chance according to the marginal probabilities. Because of this, it is a more demanding metric than the simple proportions of the number of times the coders agreed. Values of κ usually lie within 1.0 (meaning perfect agreement) and 0.0 (meaning agreement exactly as good as would be expected by chance), but values of κ can be negative if the human coders perform worse than chance. We executed a test of inter-coder reliability on each version of the rubric. The final version of the rubric (the one shown in Figure 4) was the first version which met the $\kappa > 0.8$ standard, with $\kappa = 0.84$ on 18 different responses.

Table 1. Coding results for Lab #1

Category	Number presented	Number coded	Fixed	DEL	UNR	DIFF	PART	FIX
paren. matching	129	26	76%	0	3	1	3	19
unbound id.	73	33	84%	1	3	2	2	25
syntax / define	73	32	50%	2	11	4	4	11
syntax / func. call	63	29	36%	1	10	2	7	9
syntax / cond	61	31	49%	2	12	0	4	13
arg. count	24	21	52%	1	5	0	8	7

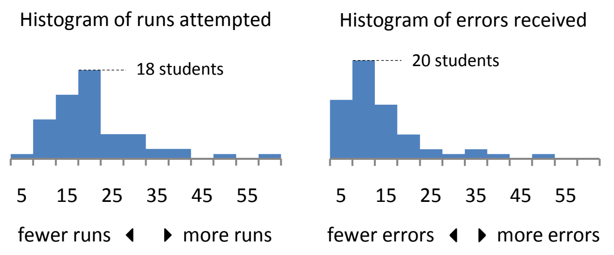


Figure 6. Histograms, Lab #1 (50 minutes)

5. APPLYING THE RUBRIC

Our rubric is designed to identify specific error messages that are problematic for students. Given that many error messages are variations on the same underlying problem, however, we found it more effective to consider messages in thematically-related categories, such as "parenthesis matching", "syntax of define", and "syntax of cond". The six categories shown in the leftmost column of Table 1 cover 423 of the 466 error messages presented to students during Lab #1.[3] Appendix B lists the specific messages that comprise each category. The second column shows the number of times students saw an error message of that category. The third column shows the number of those responses that we coded; the samples were chosen randomly from responses that contained at least one keystroke (as opposed to cases in which the student simply ran the program again with no edit to their program). The five columns to the right of the vertical line show how many samples fell under each rubric code. When running the data samples to ascribe codes, we used Köksal's edit-replay software [20]. The *Fixed* column to the left of the vertical line attempts to measure the effectiveness of errors in each category. This number is not simply the ratio of the "FIX" column to the "Number coded" column. That computation would be misleading in two ways: first, [DEL] and [DIFF] codes should not count against the effectiveness of a message; second, it does not account for differences in how often students attempt to run their programs. Figure 6 shows the histogram of run attempts in the

[3] All errors in Table 1 are syntax errors in BSL. The remaining errors consisted of 24 run-time errors, 7 syntax errors caused by illegal characters (periods, commas, hash marks and such), 7 caused by the ordering of definitions, 4 regarding the syntax of `if` (which is not taught in the course), and 1 duplicate definition.

dataset; note its long right-tail. The mode is *15 to 20 attempts*, with 18 students in this histogram bucket. This corresponds to about one attempt every 3 minutes. We avoid undue influence of frequent runs by first computing the ratio of [FIX] against the denominator $[UNR] + [PART] + [FIX]$ per individual student. Specifically, for student s and category c, we compute:

$$p_{s,c} = \frac{[FIX]}{[UNR] + [PART] + [FIX]}$$

Then we take the unweighted average across the n students who are represented in the selected samples:

$$p_c = \left(\sum p_{s,c}\right)/\mathrm{n}$$

The column *Fixed* shows the p_c's.

The data in the *Fixed* column show some clear trends. Error messages pertaining to unbound identifiers were easy to fix (84%), which is no surprise since most of them arise from simple typos. Parenthesis-matching errors were also relatively easy (76%), especially when compared to the errors pertaining to the syntax of define, function calls, and conditionals. Removing (or adding) the right number of parentheses is not as hard as choosing which ones to remove. Even though Scheme is often chosen as the programming language for introductory courses because of its simple syntax, students still struggle with that syntax. We saw many editing sequences in which students struggled to manipulate the parentheses so that their expressions ended up in the right syntactic locations.

These results support our claim that even in a project that has spent significant design effort in getting error messages right, formal human-factors studies are a critical component. Implicitly, the results emphasize the challenge in textually describing syntax errors to students with a shaky command of the grammar at hand. Figuring out how to do this effectively is a promising open research question.

While the data illustrate where students are having difficulties with the error messages, they do not suggest concrete changes to DrScheme's error message design. For that, we turn to observations from our one-on-one interviews with students.

6. SEMANTICS OF THE HIGHLIGHT

Whenever DrScheme presents an error message, it highlights at least one fragment of code that is pertinent to the error message. In contrast to the more common combination of line number and column number provided by many compilers, highlights are presumed clearer for beginners and less likely to be ignored.

Our interviews with students hinted that their interaction with the highlight is less straightforward than we thought. The following exchanges were eye-opening. We asked the students about the meaning that they attribute to the highlight, and received similar answers from three of them.

Interviewer: *When you get these highlights, what do they mean to you?*

Student #1: *The problem is between here and here, fix the problem between these two bars.*

———

Interviewer: *You were saying that you pattern match on the highlight and don't read the messages at all.*

Student #2: *I think that in the beginning it was more true, because the highlight were more or less "this is what's wrong," so when I was a beginning programmer that's what I saw and that's what I would try to fix.*

———

Interviewer: *When DrScheme highlights something, what does it highlight?*

Student #3: *It highlights where the error occurred.*

Interviewer: *Do you usually look for fixes inside the highlight?*

Student #3: *mmm… I think I did at the beginning.*

In retrospect, it makes sense. DrScheme never explicates the meaning of its highlight; students are on their own to deduce what DrScheme might mean. In fact, the semantics of the highlight varies across error messages. By manual inspection, we have found five different meanings for DrScheme's highlights, depending on the error message:

1. This expression contains the error
2. The parser did not expect to find this
3. The parser expected to see something after this, but nothing is there
4. This parenthesis is unmatched
5. This expression is inconsistent with another part of the code

The students' interpretation of "edit here" applies in at most two of these cases: the first and the fifth (though the correct edit for the fifth is often in the other half of the inconsistency). In the second case, the student must edit around the highlighted code, perhaps to combine it with another expression. In the third case, the student may need to add code to the right of the highlight or adjust parentheses to change the number of expressions within the surrounding constructs.

Interestingly, highlights do provide visually distinctive patterns that distinguish certain classes of errors. Mismatched-parenthesis errors highlight a single parenthesis. Unbound-identifier errors highlight a single identifier. Students quickly learn the highlighting semantics of these patterns. Lacking distinctive patterns for the other cases, however, students default to the (entirely reasonable) "edit here" interpretation. This is consistent with students treating DrScheme as an authoritative oracle about the errors in their programs.

Table 2. Vocabulary words

Primitive name	Predicate	Function header
Procedure	Defined name	Argument
Primitive operator	Type name	Clause
Field name	Identifier	Expression
Procedure application	Function body	Selector

During the interviews we observed multiple patterns of behavior that can be attributed to the students' confusion about the meaning of the highlight.

- In the case of inconsistency between a definition and its use, DrScheme only highlights one of the two halves of the problem, typically the use location. Students had greater difficulty fixing these errors if a correction was needed in the non-highlighted half of the inconsistency. The highlight had an over-focusing effect, blinding the students to the possibility that the problem lay in the other half.
- Students often look for a recommended course of action in the wording of the error message. For instance, once the error message mentions a missing part, students feel prompted to provide the missing part, though this might not be the correct fix. This was the case in Figure 2, where the student took the expression "expected a name" to mean "insert 'name' here", while the actual fix was to add a parenthesis. Students who follow the advice of the error risk adding further erroneous code to their already broken program. Highlighting the location of the missing part seems to strengthen this prompting effect, since students guess that these highlights mean "add something here".
- Once students recognize the visually-distinctive patterns described earlier, they seem to develop the habit of looking at the highlighting first to see if they recognize the error before consulting the text. This puts additional responsibility on the highlighting mechanism.

Most students grow out of these patterns of behavior as they progress into the course and gain more familiarity with the error messages. But even as they do, their original model still influences their approach. The best student we interviewed had learned to avoid the over-focusing effect, and would look around the highlight for possible causes of the problem. This led to the following exchange:

Interviewer:	*Which one was more useful, the highlight or the message?*
Student #2:	*mmm… I would say the message. Because then highlight was redirecting me to here, but it didn't see anything blatantly wrong here. So I read the error message, which said that it expected five arguments instead of four, so then I looked over here.*
Interviewer:	*Would you say the highlight was misleading?*
Student #2:	*Yeah. Because it didn't bring me directly to the source.*

What the student wrote:

```
(define (label-near2? label name word-1
         word-2 word-3))
```

What DrScheme Says:

define: expected an <u>expression</u> for the <u>function body</u>, but nothing's there.

What the Student Sees:

define: expected only one <u>rigmarole</u> for the <u>blah's foo</u>, but nothing's there.

Figure 7. Message vs perception

A fix was found outside the highlight, but the student described the highlight as wrong, suggesting that the student maintained a perception that the intended semantic of the highlight was "the bug is here". The student had simply developed some skepticism about the accuracy of the oracle.

Attempting to explain the different highlighting semantics to students in their first week of programming is challenging. Each interpretation has a semantics in terms of the processes that detect errors (parsing and run-time checking). However, CS1 students do not have knowledge necessary to make sense of this interpretation, and they surely cannot be expected to deduce it from their observation of DrScheme's behavior. Without a systematic way of understanding the messages given to them, students learn that programming is a discipline of haphazard guessing—the very reverse of our teaching objective.

Programming texts frequently present formal grammars (through syntax diagrams [35] or textual BNF) to help explain language syntax; some include exercises on deciphering text through grammar rules [2]. Unfortunately, the highlighting is undermining this effort by describing syntax rejection in terms of a different process (parsing) that the students have not been taught, and which they cannot be expected to understand at this early stage of their computing education.

7. VOCABULARY

DrScheme's error messages use precise technical vocabulary to describe the problem and to refer to the parts of the code that are involved in the error. Table 2 shows the 15 technical vocabulary words in the 90th-percentile of the most frequently-presented error messages over our entire data set (not just Lab #1).

When we reviewed the text of the error messages, we found that DrScheme is mostly accurate and consistent in its usage of its technical vocabulary. Yet, throughout all four interviews, we noticed that the students had only a weak command of that vocabulary. When describing code, the students misused words, or used long and inaccurate phrases instead of using the corresponding precise technical word. This was perplexing, since the interviews occurred after the students had spent 4 to 6 weeks reading these technical words in the error messages. Plus, some exchanges during the interview suggested that the students' poor command of the vocabulary undermined their ability to respond to the messages.

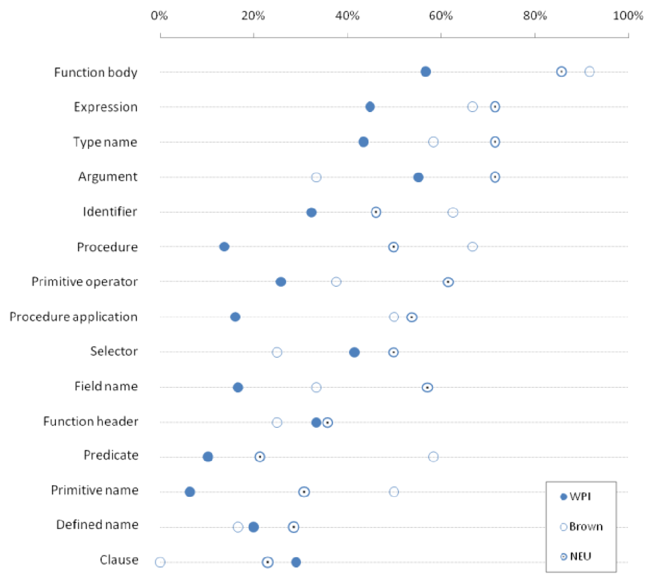


Figure 8. Average percent correct per word on the vocabulary quiz

	Brown	NEU	WPI
Function body		✓	✓
Expression	✓	✓	✓
Type name			✓
Argument	✓	✓	✓
Identifier	✓		✓
Procedure	✓		
Primitive operator	✓		
Procedure application	✓	✓	
Selector	✓	✓	✓
Field name		✓	
Function header		✓	✓
Predicate	✓	✓	
Primitive name	✓		
Defined name			✓
Clause	✓	✓	✓

✓ = Used in Class

Table 3. In-class word use

The following exchange happened after the student had spent two and a half minutes trying to formulate a response to the error message shown in Figure 7. After observing that the student was not making progress, the interviewer decided to provide a hint.

Interviewer: *The error message says "the function body." Do you know what "function body" means?*

Student: *Nah… The input? Everything that serves as a piece of input?*

Interviewer: *Actually, it's this. When DrScheme says "function body" it means this part.*

Student: *Oh man! I didn't…*

The student then proceeded to fix the error successfully. To help the student, it was sufficient to provide a non-definitional meaning for the expression "function body", by pointing at the function body of a different function.

To measure students' command of the vocabulary, we developed a short quiz that asked them to circle instances of five vocabulary words from Table 2 in a simple piece of code. Appendix A contains one version of this quiz. We administered the quiz at three different universities: WPI, Brown, and Northeastern. We received 90, 32, and 41 responses respectively. At each university, students had used DrScheme for at least a couple of months before taking the quiz.

The results are roughly similar across all three universities (see Figure 8). Some words are harder than others. Northeastern's data are slightly stronger, while WPI's are slightly weaker. More importantly, only four words were correctly identified by more than 50% of the students. These results do not necessarily imply that vocabulary underlies students' difficulties responding to errors; students could have conceptual understanding of the messages without the declarative understanding of the vocabulary.

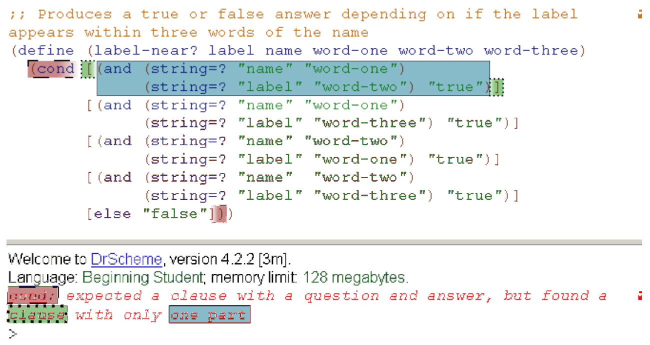
Nonetheless, these results question whether students are able to make sense of the error messages. As the quizzes were anonymous, we were not able to correlate quiz performance to our coding data on the recorded editing sessions.

We asked the professors which of the terms from Table 2 they had used in class to describe code. Table 3 presents their answers. Whenever a word used by DrScheme was not used in class, the professors either elected to use a different word or simply found it was not necessary to introduce the concept in class. For instance, the two professors who did not use the term "procedure" used the term "function" instead.

Studies frequently use control groups to quantify the effect of an intervention. While we did not create control groups around the usage of terms in class, by happenstance 11 of the 15 words were used at some universities but not others. These words formed controlled trials (a technical term), in which it was possible to quantify the effect of a word being used in class on the students' understanding of that word. To help factor out the effect of uninteresting variability, namely the variability in university strengths and in word difficulty, we fitted a linear model to the data. The model had 17 variables total. The first 14 variables were configured to each capture the intrinsic difficulty of one word, relative to a fixed 15th word, the next two variables were configured to capture relative university strength. The last variable was set to capture the influence of a word's use in class. The fit on this last variable indicated that using a word in class raises its quiz score by 13.8% (95% confidence interval, 2.93% to 24.7%), a result which is statistically significant at the 0.05 level ($p=0.0147$).

These results raise many interesting research questions:

- We know that students struggle to respond to error messages. Can we quantify the extent by which this is caused by their poor command of the vocabulary?



```
;; Produces a true or false answer depending on if the label
appears within three words of the name
(define (label-near? label name word-one word-two word-three)
  (cond [(and (string=? "name" "word-one")
              (string=? "label" "word-two") "true")]
        [(and (string=? "name" "word-one")
              (string=? "label" "word-three") "true")]
        [(and (string=? "name" "word-two")
              (string=? "label" "word-one") "true")]
        [(and (string=? "name"  "word-two")
              (string=? "label" "word-three") "true")]
        [else "false"]))
```

```
Welcome to DrScheme, version 4.2.2 [3m].
Language: Beginning Student; memory limit: 128 megabytes.
cond: expected a clause with a question and answer, but found a
clause with only one part
>
```

Figure 9. Colored-coded error message

- Using a word in class raises the students' understanding of the word relatively little. How are they learning the vocabulary, then? If they are learning it by reading error messages that they do not understand well, what are they learning?
- Some error messages make statements where everyday words are used in a technical sense, such as "indentation" or "parenthesis" (which DrScheme sometime uses to refer to a square bracket, since the parser considers them equivalent). Are these words a problem as well?

The results also raise pedagogic questions about good approaches to teach the technical vocabulary of programming. Should courses use specialized vocabulary training tutors (such as FaCT [28])? Lecture time is limited, as are homework contact hours; could the error messages help teach the vocabulary?

All three professors agreed that the mismatch between their vocabulary usage and DrScheme's was contrary to their efforts to use consistent language in class. Moreover, once the issue was pointed out to them, they all agreed that adjustments were needed. In general, we suspect professors tend to forget about the content of errors and other IDE feedback when designing lectures; the connection between curricula and IDEs needs to be tighter.

8. RECOMMENDATIONS

The results presented in Sections 5 through 7 point to three broad issues: students' difficulties working with syntax in the first week of class, inconsistent semantics of highlighting, and students' poor command of the vocabulary used in the error messages. In recommending solutions, we considered three key principles:

- Many developers contribute to DrScheme. Error-message conventions need to be easy for multiple developers to follow.
- Error messages should not propose solutions. Even though some errors have likely fixes (missing close parentheses in particular places, for example), those fixes will not cover all cases. Given students' tendencies to view DrScheme as an oracle, proposed solutions could lead them down the wrong path; even error systems designed for experts sometimes follow this principle [8]. This principle directly contradicts requests of the students we interviewed, who had learned common fixes to common errors and wanted the messages to propose corrections.
- Error messages should not prompt students towards incorrect edits. This is related to, yet distinct from, the previous principle.

The first is particularly pertinent to addressing problems with the highlighting semantics. One could propose changing the color of the highlight based on its semantics. This would violate the first constraint, as it requires developers to interpret those semantics (additional problems make the proposal a poor choice). The second warns against proposing corrections to syntax errors. The third reminds us to carefully consider how students might interpret a highlight.

With these principles in hand, we have three recommendations:

Simplify the vocabulary in the error messages. DrScheme's messages often try too hard to be thorough, such as distinguishing between selectors and predicates in error messages that expect functions. The semantic distinctions between these terms are often irrelevant to students, particularly in the early weeks. We have simplified the terminology in Beginner Language messages and will be testing it on students in the fall. If this simplification is effective, the DrScheme developers may want to consider breaking Beginner Language into sublanguages based on error terminology, in addition to provided constructs.

Help students match terms in error messages to code fragments. Error messages contain many definite references, such as "the function body" or "found one extra part". As instructors, we often help students by connecting these references to the corresponding pieces of code. Sometimes, DrScheme's highlighting achieves this effect, too (as with unbound identifiers or unmatched parentheses). However, messages often contain multiple terms, while DrScheme currently highlights only one code fragment.

Treat error messages as an integral part of course design. IDE developers should apply the common curricular concerns of consistency, complexity and learning curves to the design of error messages. Professors must ensure their curriculum aligns with the content of the error messages, just like math professors ensure their notation matches that of the textbook.

The second recommendation suggests a new presentation for error messages: highlight every definite reference with a distinct color. Figure 9 shows a preliminary mockup of this idea. Each definite reference in the message uses color to point to a specific code fragment (colors are outlined with different line styles for black-and-white viewing). This design has several benefits: it resolves the ambiguity about highlighting (since highlights correspond exactly to terms in the message), it eliminates ambiguous references (as seen in Figure 2), and it gives students a chance to learn the vocabulary by example (in Figure 9, the meaning of the word "clause"). This design naturally highlights both the definition and the use on an inconsistency error (since both are referred to by the text of the error messages), which should avoid triggering the over-focusing behavior we observed. Early versions of this design heavily influenced our stated principles. For example, we briefly considered highlighting indefinite references (such as "question" in Figure 9) until we realized it violated the third principle. We are currently refining this design with intent to deploy it experimentally next year.

In addition, we intend to develop vocabulary conventions for talking about Beginner Student Language code. This convention will cover both the needs of the error messages and the needs of educators. The convention document will help maintain consistency across all the authors of libraries intended to be used in BSL, as well as between the classroom and the error messages.

Our recommendations about color-coded highlights and consistent vocabulary are not specific to Scheme. They should apply just as well in any other programming language used for teaching, including those with graphical syntaxes, to the extent that they have error messages.

9. RELATED WORK

The principles of HCI frame general discussions on the design of pedagogic programming languages [27], as well as on the design of error messages specifically [33]. These reflections informed our work.

Alice [23] and BlueJ [13] are two widely used pedagogic IDEs. Both environments show students the error messages generated by full-fledged Java compilers. In independent evaluations involving interviews with students, the difficulty of interpreting the error messages fared amongst the students' primary complaints [13] [31]. These difficulties have led professors to develop supplemental material simply to teach students how to understand the error messages [1]. One evaluation of BlueJ asked the students whether they found the messages useful [34]. Most did, but it is unclear what this means, given that they were not offered an alternative. The students we interviewed were similarly appreciative of the error messages of DrScheme, despite their struggles to respond to them. That said, our study shows that DrScheme's errors are still a long way from helping the students, and other recent work [7] also presents evidence of this.

There are still relatively few efforts to evaluate the learning impact of pedagogic IDEs [29]. Gross and Powers survey recent efforts [12], including, notably, those on Lego mindstorms [9] and on Jeliot 2000 [22]. Unlike these other evaluations, we did not evaluate the impact of the IDE as a whole. Rather, we attempted to tease out the effect of individual components.

A number of different groups have tried to rewrite the error messages of professional Java compilers to be more suitable for beginners. The rewritten error messages of the Gauntlet project [11], which have a humorously combative tone, explain errors and provide guidance. The design was not driven by any observational study; a follow-up study discovered that Gauntlet was not addressing the most common error messages [17]. The Karel++ IDE adds a spellchecker [3], and STLFilt rewrites the error messages of C++; neither has been evaluated formally [36].

Early work on the pedagogy of programming sought to classify the errors novice programmers make when using assembly [4] or Pascal [32]. More recent work along the same lines studies BlueJ [30] [18], Gauntlet [17] Eiffel [25], and Helium [14]. Others have studied novices' behavior during programming sessions. This brought insight on novices' debugging strategies [24], cognitive inclination [19], and development processes [20]. Our work differs in not studying the students' behavior in isolation; rather, we focus on how the error messages influence the students' behavior.

Coull [6], as well as Lane and VanLehn [21] have also defined subjective rubrics, though they evaluate the students' programming sessions rather than the success of individual error messages. In addition, vocabulary and highlighting were not in the range of considered factors affecting student responses to errors. Coull also added explanatory notes to the error messages of the standard Java compiler based on their observations. These notes made experimental subjects significantly more likely to achieve an ideal solution to short exercises.

Nienaltowski et al. [26] compared the impact of adding long-form explanation to an error message, and of adding a highlight on three different error messages, in a short web-based experiment. They found that the former has no impact, while the later impairs performance slightly. Unfortunately, the experiment's design has many threats to validity, some of which the paper acknowledged.

10. ACKNOWLEDGMENTS

Fatih Köksal generously provided his recording and playback software. Nate Krach, Janice Gobert, and Ryan S. de Baker offered extensive advice about social science methodology, analysis, and study design. Tamara Munzner discussed study design and provided useful references. Jesse Tov, Glynis Hamel, and Amy Greenwald generously gave class time and followup information for our vocabulary quiz. Matthias Felleisen offered valuable discussion and comments on an early version of the paper. Several U.S. National Science Foundation grants supported the research leading up to and including this project.

11. REFERENCES

[1] Ben-Ari, M.M. 2007. Compile and Runtime Errors in Java. *http://stwww.weizmann.ac.il/g-cs/benari/oop/errors.pdf, accessed June 15, 2010.*

[2] Bloch, S. *Picturing Programs*. College Publications (publication pending).

[3] Burrell, C. and Melchert, M. 2007. Augmenting compiler error reporting in the Karel++ microworld. *Proceedings of the Conference of the National Advisory Committee on Computing Qualifications* (2007), 41–46.

[4] Chabert, J.M. and Higginbotham, T.F. 1976. An Investigation of Novice Programmer Errors in IBM 370 (OS) Assembly Language. *Proceedings of the ACM Southeast Regional Conference* (1976), 319-323.

[5] Cohen, J. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*. 20, 1 (1960), 37–46.

[6] Coull, N.J. 2008. *SNOOPIE: development of a learning support tool for novice programmers within a conceptual framework*. PhD Thesis, School of Computer Science, University Of St. Andrews.

[7] Crestani, M. and Sperber, M. 2010. Experience Report: Growing Programming Languages for Beginning Students. *Proceedings of the International Conference on Functional Programming* (2010).

[8] Culpepper, R. and Felleisen, M. 2010. Fortifying Macros. *Proceedings of the International Conference on Functional Programming* (2010).

[9] Fagin, B.S. and Merkle, L. 2002. Quantitative analysis of the effects of robots on introductory Computer Science

education. *Journal on Educational Resources in Computing*. 2, 4 (2002), 1-18.

[10] Findler, R.B., Clements, J., et al. 2002. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*. 12, 02 (2002), 159–182.

[11] Flowers, T., Carver, C., et al. 2004. Empowering students and building confidence in novice programmers through Gauntlet. *Frontiers in Education*. 1, (2004), T3H/10 - T3H/13.

[12] Gross, P. and Powers, K. 2005. Evaluating assessments of novice programming environments. *Proceedings of the International Workshop on Computing Education Research*. (2005), 99-110.

[13] Hagan, D. and Markham, S. 2000. Teaching Java with the BlueJ environment. *Proceedings of Australasian Society for Computers in Learning in Tertiary Education Conference* (2000).

[14] Hage, J. and Keeken, P.V. Mining Helium programs with Neon. *Technical Report, Department of Information and Computing Sciences, Utrecht University*.

[15] Holt, R.C., Wortman, D.B., et al. 1977. SP/k: a system for teaching computer programming. *Communications of the ACM*. 20, 5 (1977), 301–309.

[16] Hristova, M., Misra, A., et al. 2003. Identifying and correcting Java programming errors for introductory computer science students. *Proceedings of the Symposium on Computer Science Education* (2003), 153–156.

[17] Jackson, J., Cobb, M., et al. 2005. Identifying top Java errors for novice programmers. *Proceedings of the Frontiers in Education Conference* (2005), T4C–24.

[18] Jadud, M.C. 2005. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*. 15, 1 (2005), 25–40.

[19] Jadud, M.C. 2006. Methods and tools for exploring novice compilation behaviour. *Proceedings of the International Workshop on Computing Education Research* (2006), 73–84.

[20] Köksal, M.F., Başar, R.E., et al. 2009. Screen-Replay: A Session Recording and Analysis Tool for DrScheme. *Proceedings of the Scheme and Functional Programming Workshop, Technical Report, California Polytechnic State University, CPSLO-CSC-09-03* (2009), 103-110.

[21] Lane, H.C. and VanLehn, K. 2005. Intention-based scoring: An approach to measuring success at solving the composition problem. *ACM SIGCSE Bulletin*. 37, 1 (2005), 373-377.

[22] Levy, R.B., Ben-Ari, M., et al. 2003. The Jeliot 2000 program animation system. *Computers & Education*. 40, 1 (2003), 1-15.

[23] Moskal, B., Lurie, D., et al. 2004. Evaluating the effectiveness of a new instructional approach. *Proceedings of the Symposium on Computer Science Education*. 35, (2004), 75-79.

[24] Murphy, L., Lewandowski, G., et al. 2008. Debugging: the good, the bad, and the quirky — a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin*. 40, 1 (2008), 163-167.

[25] Ng Cheong Vee, M., Mannock, K., et al. 2006. Empirical study of novice errors and error paths in object-oriented programming. *Proceedings of the Conference of the Higher Education Academy, Subject Centre for Information and Computer Sciences* (2006), 54-58.

[26] Nienaltowski, M., Pedroni, M., et al. 2008. Compiler Error Messages: What Can Help Novices? *Proceedings of the Technical Symposium on Computer Science Education*. 39, (2008), 168-172.

[27] Pane, J., Myers, B.A., et al. 2002. Using HCI Techniques to Design a More Usable Programming System. *Proceedings of the Symposia on Human Centric Computing Languages and Environments* (2002), 198-206.

[28] Pavlik, P.I., Presson, N., et al. 2007. The FaCT (fact and concept) system: A new tool linking cognitive science with educators. *Proceedings of the Conference of the Cognitive Science Society* (2007), 397-402.

[29] Pears, A., Seidman, S., et al. 2007. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*. 39, 4 (2007), 204-223.

[30] Ragonis, N. and Ben-Ari, M. 2005. On understanding the statics and dynamics of object-oriented programs. *ACM SIGCSE Bulletin*. 37, 1 (2005), 226-230.

[31] Rey, J.S. 2009. *From Alice to BlueJ: a transition to Java*. Master's thesis, School of Computing, Robert Gordon University.

[32] Spohrer, J.C. and Soloway, E. 1986. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*. 29, 7 (1986).

[33] Traver, V.J. 2010. On compiler error messages: what they say and what they mean. *Technical Report, Computer Languages and Systems Department, Jaume-I University* (2010).

[34] Van Haaster, K. and Hagan, D. 2004. Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool. *Information Science + Information Technology Education Joint Conference* (2004).

[35] Wirth, N. 1971. The Programming Language Pascal. *Acta Informatica*. 1, (1971), 35-63.

[36] Zolman, L. 2005. STLFilt: An STL error message decryptor for C++. *http://www.bdsoft.com/tools/stlfilt.html, accessed June 10, 2010* (2005).

12. APPENDIX A — VOCABULARY QUIZ

Circle one instance of each vocabulary term on the code below. Label each circle with the question number. For example, the circle labeled Q0 is an instance of the term "Return Type".

If you do not know what a term means, write a big "X" on it (in the left column). The right column gives examples of each term as used in DrScheme's error messages. The errors are irrelevant otherwise.

	Vocabulary term	Sample usage
Q1.	Argument	>: expects at least 2 **arguments**, given 1
Q2.	Selector	this selector expects 1 argument, here it is provided 0 arguments
Q3.	Procedure	this procedure expects 2 arguments, here it is provided 0 arguments
Q4.	Expression	expected at least two expressions after `and', but found only one expression
Q5.	Predicate	this predicate expects 1 argument, here it is provided 2 arguments

```
;; (make-book number string string number number bst bst)
(define-struct book (isbn title author year copies left right))

                                                  Q0
;; this-edition?:  bst number number -> boolean
;; Consumes a binary search tree, an ISBN number, and a year, and produces true
;; if the book with the given ISBN number was published in the given year
(define (this-edition? a-bst isbn-num year)
  (cond [(symbol? a-bst) false]
        [(book? a-bst)
         (cond [(= isbn-num (book-isbn a-bst))
                (= year (book-year a-bst))]
               [(< isbn-num (book-isbn a-bst))
                (this-edition? (book-left a-bst) isbn-num year)]
               [else (this-edition? (book-right a-bst) isbn-num year)])]))
```

13. APPENDIX B — ERROR MESSAGE DETAILS FOR TABLE 1

Read:
```
   read: bad syntax `#1\n'
   read: expected a closing '\"'; newline within string suggests a missing '\"' on line 20
   read: illegal use of \".\"
   read: illegal use of backquote
   read: illegal use of comma
```

Definitions / duplicate:
```
   babel: this name was defined previously and cannot be re-defined
```

Definitions / ordering:
```
  "reference to an identifier before its definition: liberal
```

Unbound id.:
```
  "~a: name is not defined, not a parameter, and not a primitive name
```

Argument count:
```
   and: expected at least two expressions after `and', but found only one expression
   check-expect: check-expect requires two expressions. Try (check-expect test expected).
   ~a: this procedure expects 3 arguments, here it is provided 1 argument
   or: expected at least two expressions after `or', but found only one expression
   string?: expects 1 argument, given 2: \"bob\" \"m\"
```

Syntax / function call:
```
   =: this primitive operator must be applied to arguments; expected an open parenthesis before the primitive operator name
   and: found a use of `and' that does not follow an open parenthesis
   cond: found a use of `cond' that does not follow an open parenthesis
   function call: expected a defined name or a primitive operation name after an open parenthesis, but found a function argument name
   function call: expected a defined name or a primitive operation name after an open parenthesis, but found a number
   function call: expected a defined name or a primitive operation name after an open parenthesis, but found something else
   function call: expected a defined name or a primitive operation name after an open parenthesis, but nothing's there
   or: found a use of `or' that does not follow an open parenthesis
   political-label: this is a procedure, so it must be applied to arguments (which requires using a parenthesis before the name)
   string-one-of?: this is a procedure, so it must be applied to arguments (which requires using a parenthesis before the name)
   string=?: this primitive operator must be applied to arguments; expected an open parenthesis before the primitive operator name
   string?: this primitive operator must be applied to arguments; expected an open parenthesis before the primitive operator name
   word01: this is a procedure, so it must be applied to arguments (which requires using a parenthesis before the name)
```

Parenthesis matching:
```
   read: expected `)' to close `(' on line 19, found instead `]'; indentation suggests a missing `)' before line 20
   read: expected `)' to close `(' on line 31, found instead `]'
   read: expected `)' to close preceding `(', found instead `]'
   read: expected a `)' to close `('
   read: expected a `)' to close `('; indentation suggests a missing `]' before line 20
   read: expected a `]' to close `['
   read: expected a `]' to close `['; indentation suggests a missing `)' before line 20
   read: missing `)' to close `(' on line 20, found instead `]'
   read: missing `)' to close `(' on line 39, found instead `]'; indentation suggests a missing `)' before line 41
   read: missing `)' to close preceding `(', found instead `]'
   read: missing `)' to close preceding `(', found instead `]'; indentation suggests a missing `)' before line 20
   read: missing `]' to close `[' on line 21, found instead `)'; indentation suggests a missing `)' before line 22
   read: missing `]' to close `[' on line 33, found instead `)'
   read: missing `]' to close preceding `[', found instead `)'
   read: missing `]' to close preceding `[', found instead `)'; indentation suggests a missing `)' before line 27
   read: unexpected `)'
   read: unexpected `]'"))
```

Syntax / if:
```
   if: expected one question expression and two answer expressions, but found 1 expression
   if: expected one question expression and two answer expressions, but found 2 expressions
```

Syntax / cond:
```
   cond: expected a clause with a question and answer, but found a clause with only one part
   cond: expected a clause with one question and one answer, but found a clause with 3 parts
   cond: expected a clause with one question and one answer, but found a clause with 4 parts
   cond: expected a question--answer clause, but found something else
   else: not allowed here, because this is not an immediate question in a `cond' clause
```

Syntax / define:
```
   define: expected a function name, constant name, or function header for `define', but found something else
   define: expected a name for a function, but found a string
   define: expected a name for a function, but found something else
   define: expected a name for the function's 1st argument, but found a string
   define: expected a name for the function's 1st argument, but found something else
   define: expected an expression for the function body, but nothing's there
   define: expected at least one argument name after the function name, but found none
   define: expected only one expression after the defined name label-near?, but found at least one extra part
   define: expected only one expression after the defined name label-near?, but found one extra part
   define: expected only one expression for the function body, but found at least one extra part
   define: expected only one expression for the function body, but found one extra part
```

Runtime / cond:
```
   cond: all question results were false
```

Runtime / type:

```
   and: question result is not true or false: \"true\"
   or: question result is not true or false: \"conservative\"
   string=?: expects type <string> as 1st argument, given: 'french; other arguments were: 'spanish
   string=?: expects type <string> as 1st argument, given: 2; other arguments were: 1 1 1 3
```

List of unbound identifiers:

```
                                                  ele                                               sybol=?
  /1.0                                            els                                               symbol-?
  ==                                              flase                                             symbol=2
  >label-near1?                                   hallo                                             synbol=?
  >label-near?                                    j                                                 temp
  Define                                          label-near1                                       test-expect
  Edit                                            label-near?                                       to-look-for
  Ryan                                            label                                             true
  Smith                                           labelwordwordwordname                             ture
  activity-type                                   land                                              tv
  actvity-type                                    liberal                                           word-to-look-for
  bable                                           love                                              word1
  celsis->fahrenheit                              me                                                word1orword2orword3
  celsius->fhrenheit                              name1                                             word1word2word3
  celsius-fahrenheit                              political-label                                   word
  celsius>fahrenheit                              political                                         yes
  celssius->fahrenheit                            senate
  dedfine                                         str=?
  dfine                                           string-locale=?
```