Mind Your Language: On Novices' Interactions with Error Messages

Guillaume Marceau, Kathi Fisler
WPI Department of Computer Science
Worcester, MA USA
{gmarceau,kfisler}@cs.wpi.edu

Shriram Krishnamurthi
Brown University Dept of Computer Science
Providence, RI, USA
sk@cs.brown.edu

Abstract

Error messages are one of the most important tools that a language offers its programmers. For novices, this feedback is especially critical. Error messages typically contain both a textual description of the problem and an indication of where in the code the error occurred. This paper reports on a series of studies that explore beginning students' interactions with the vocabulary and source-expression highlighting in DrRacket. Our findings demonstrate that the error message significantly fail to convey information accurately to students, while also suggesting alternative designs that might address these problems.

Categories and Subject Descriptors D.3.m [**Programming Languages**]: Miscellaneous; K.3.2 [**Computer and Education**]: Computer and Information Science Education—Computer science

General Terms Languages, Human Factors.

Keywords *Novice programmers, error message design, beginner-friendly IDEs, user-studies.*

1. Introduction

Error messages are one of the most critical user experience elements for programmers. These messages play at least two critical roles: as a programming tool, they should help the user progress towards a working program; as a pedagogic tool, they should help the user understand the problem that led to the error. In addition, they must avoid frustrating the user further, either by being too hard to understand or by leading a user down the wrong path to correcting the problem.

Yet, ask any experienced programmer about the quality of error messages in their programming environments, and you will often get an embarrassed laugh. In every environment, a mature programmer can usually point to at least a handful of "favorite" bad error responses. When they find out that the same environment is being used by novices, their laugh often hardens.

If you agree with the previous paragraphs, ask yourself this: when's the last time you saw a paper with ***rigorous human-factors evaluation*** *on this topic?* Everyone knows user evaluation matters, especially with novice programmers. Future-work sections of papers say "we really should do some user evaluation"—always in the future. Living in glass houses, we don't ask each other hard questions about this. Most of us would not even know where to begin addressing the question. It's time to change that.

Many researchers in programming languages and IDE design have long felt that students and professional programmers need different levels or forms of feedback. IDEs geared at beginning programmers take various approaches to this problem, from supporting custom languages for beginners to clarifying their error messages. DrRacket's language-levels provide a hybrid model in which a full-fledged language is staged into sub-languages that tailor the available constructs (and error messages) to what students will have seen at different points in the course. In this approach, programs that are legal in advanced levels (such as functions with no arguments) may be illegal in beginning levels. In addition, the sub-languages elide many advanced constructs (such as macros and contracts) that are available in the professional Racket programming language.

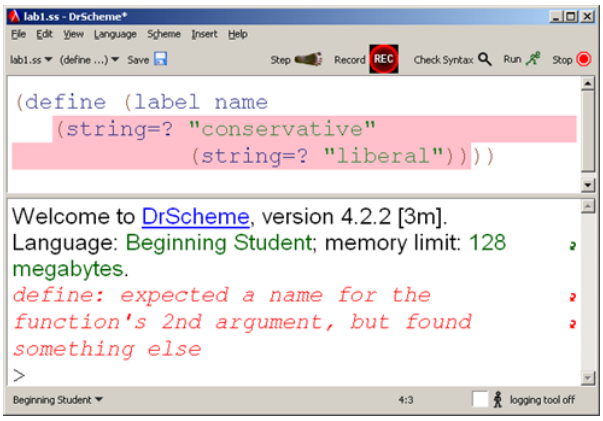
The DrRacket development team put considerable effort into the design of the error messages. They carefully considered both form and terminology, and refined the messages many times over the years based on their observations of students. Each message presents a textual description of the problem and highlights a relevant expression in the source code. In 2009, curious about difficulties students seemed to be having with responding to the messages, we began formal user-based studies of the problems. We logged students' programming sessions and explored the effectiveness of the error messages at helping students make progress. The results have been humbling, as well as a source of many interesting questions.

At a low level, the results show that students struggle with the carefully-designed vocabulary of the error messages and often misinterpret the source highlighting. At a high level, this work reveals that the DrRacket team (which includes the authors of this paper) lacked a clear model of errors and feedback to guide their design. Given the tight connection between error reporting and linguistic decisions such as parsing strategies, this is clearly a Programming Languages problem as much as one of HCI.

This paper describes a series of formal studies on novice students' understanding of vocabulary and source highlighting. These illustrate the kinds of useful information that arise from formal studies, particularly those that inform a larger model of how students should interact with the error messages. Our observations are scoped narrowly: we focus on error message text and error highlighting without considering other mechanisms such as debuggers, dynamic error stack contexts, etc. Nevertheless, we do present some recommendations for IDE designers. Overall, we hope to help raise the level of discourse among PL researchers about how to effectively consider the impact of our work on users.

SIGPLAN'05 June 12–15, 2005, Location, State, Country.

(a)



(b)

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

Figure 1. (a) A student's program and its error message, (b) The student's response to the error message

2. Exploring Students' Responses to Errors

Figure 1(a) shows a screenshot of DrRacket presenting an error message. The student has written the program in the upper window and hit the "Run" button located in the upper-right of the window. DrRacket reports an error to the student through the text in the lower window (starting with "define:") and highlights an expression in the source code. In this case, DrRacket is reporting a parsing error: the student did not close the parameter list for the `label` function after the `name` parameter, so DrRacket is trying to treat the highlighted expression as a parameter name (presumably the student meant for that expression to be the function body). The error message text reports this problem, referring to the highlighted expression as "something else".

How do we gauge whether this error message helped the student correct the problem? The edit that the student performed in response is a good indicator. In this case (Figure 1 (b)), the student inserted the identifier `name` after the leading `string=?` in the highlighted expression. Knowing that the error is reporting on a parsing problem within a parameter list, this seems an odd edit. However, depending on how the student interpreted the message, the action might make sense. DrRacket has highlighted a chunk of text and reported that something is wrong with the "2nd argument". The student has inserted a term (in this case, `name`) into the second position of the highlighted expression. Whether the student chose `name` based on the existing parameter or the use of "name" in the error prose is unclear.

One could certainly argue that the error message was not effective in this case because the student's edit did not address the underlying problem (the student will get the same error for the same reason if he tries to run the edited code). Before recommending mitigations for this situation, however, we should assess the frequency of the problem across a larger pool of students. To this end, we logged students' interactions with DrRacket at the keystroke level, including timing data. In addition, each time a student received an error message, we saved a copy of her program. This combination of logged keystrokes and file snapshots lets us replay students' responses to error messages: we see what error they received, where they edited in response, and how much editing they did before attempting to run the program again.

The example in Figure 1 is one of the many programs we collected. It occurred during the first lab session of WPI's introductory course for novice programmers. We collected data from 60 students (out of 120, self-selected via consent to participate in our study) in the course. We collected data during lab sessions, which ran 50 minutes per week for 6 weeks.

To analyze this data, two experienced instructors independently assessed the extent to which individual edits addressed the reported problems. The marking rubric (validated for inter-rater reliability) and detailed results are reported elsewhere [1]. Our initial goal was to identify specific error messages that performed poorly in practice. The results emphasized that the "performance of a message" had to be considered in context of the course. For example, we saw a surge in errors associated with the wrong numbers of arguments to functions in the fifth week of the course, but those traced to students making mistakes while building examples of n-ary trees (which involve lots of nested calls to constructors). Even in the general context, however, the data suggest that students often fail to respond to errors effectively. The more interesting question, then, is why.

3. Digging Deeper: Interviews About Errors

The edits alone do not indicate what a student was thinking while processing an error message. For that, we require dialog with students as they work with the messages. To that end, we recruited four students from the original study to do individual hour-long talk-aloud sessions with the study team. We gave each student a series of programs from the data set and asked them to correct the error, talking about how they interpreted the error as they worked on the problem. We first presented each student with a few programs they had written during their first lab, then we presented programs from a common set of other, often subtle, errors made during Lab #1 by their classmates in the study. At various points during each interview, we asked each student about their interpretations of the messages. We audio-recorded and transcribed each of these sessions.

Inspired by examples such as that in Figure 1, we asked students about their interpretations of the highlighting. The following excerpts are quoted from the transcripts:

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 DrRacket highlights something, what does it highlight?*

```
(define (string-one-of? check-for-match stringOne stringTwo)
  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
```

Figure 2. Troublesome Fragment

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.*

All three of these excerpts suggest that students initially interpreted the highlighting as saying "edit here", or "look inside here for the problem". As we discuss in Section 4. , the actual semantics of DrRacket's highlights are more subtle than this.

During the interviews, we observed that students misused words, or used long and inaccurate phrases instead of using the precise technical terms when describing code. 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.

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 2. 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 DrRacket 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 (not the locus of the error).

We also noticed that students tended to 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 felt prompted to provide the missing part, though this might not be the correct fix. This could explain the edit from Figure 1 where the student took the expression "expected a name" to mean "insert 'name' here", while the actual fix was to add a parenthesis.

What the Interviews Tell Us

We intended the interviews to be formative, simply suggesting issues that warranted further exploration as we tried to understand how students respond to DrRacket's error messages. Accordingly, the small number of students we interviewed is not scientifically problematic (especially once multiple interviews point to common issues).

Students' weak facility with DrRacket's technical vocabulary and misinterpretation of the highlight stood out as both common and fundamental problems. The vocabulary problems were particularly surprising, given the relatively few concepts required for programming in DrRacket and its designers' efforts to choose vocabulary carefully. As a result, we looked at both highlighting and vocabulary more carefully.

4. Semantics of the Highlight

Informally, the highlight means "this expression or parenthesis is related to the error". The common "edit here" interpretation ascribes a more precise semantics to the highlight. Through manual inspection of all of the error messages in the Beginning Student language, we found five different meanings for DrRacket's highlights, depending on the error:

1. This expression raised a runtime exception
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 "edit here" interpretation 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, which the highlighting does not identify explicitly). 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.

As computer scientists, we understand the genesis of these different semantics. Highlights arise from either compile-time or runtime errors. In the case of a runtime error, the expression that failed to evaluate properly gets highlighted (the first and fifth cases in our list). In the case of a parsing error, the highlighting reflects where the source code is inconsistent with the language grammar. Here, the third case is particularly interesting, as DrRacket must highlight an expression to explain the absence of another expression. In some IDEs, the errors focus on source locations, giving errors such as

`expected to see` *a token such as if, {, or return* `here`

where the visual aid points to a particular character position within the source code. DrRacket instead chooses to highlight expressions, presumably in an attempt to help students focus on the problematic expression, more than the location. Which style of errors gets produced is closely tied to the parsing strategy that the language employs.

In the absence of explicit instruction about how to work with the highlights, this discussion suggests that students need to understand (or infer) the parsing strategy. 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 DrRacket's behavior. DrRacket does not explain highlighting to students. If a class does not take care to explain the highlighting (as we admit we never thought to do in our own classes), students are on their own to deduce its meaning. 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.

That said, highlights do provide visually distinctive patterns for certain classes of errors. Mismatched-parenthesis errors highlight a single parenthesis. Unbound-identifier errors highlight a single identifier. Students quickly learn the highlighting seman-

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

Table 1. Vocabulary words

tics of these patterns. Distinguishing cases in which entire expressions are highlighted requires students to look more closely at a combination of the structure of the highlighted expression and the accompanying error text. It is in these more complicated cases that students need help understanding highlighting semantics.

Simple instructions such as "find the phrase in the error text that matches the highlight" are unlikely to suffice. While most of DrRacket's error messages reference the highlighted code (e.g., "something else" in Figure 1), some do not. Furthermore, the correspondence is sometimes ambiguous in potentially misleading ways. In Figure 1, the phrase "the function's second argument" could refer to the function being defined or the function being called. The latter interpretation could help explain the student's mistaken response to this message.

What the Highlighting Observations (Do Not) Tell Us

While the interviews strongly suggest that students read expression highlights as saying "edit here", we don't know why. For example, students may view DrRacket as an oracle that "knows" how to fix their programs; they may be working quickly and thus try editing in the highlighting first (rather than think too hard about the problem until necessary); perhaps the semantics of the highlight is actually clear if students take the time to look carefully at the highlight in the context of the textual message. There may be obstacles preventing students from developing a more precise semantics, such as "the error mentions the highlight" interpretation. Each of these reasons, if dominant, would suggest different changes to the IDE.

5. Vocabulary

While vocabulary difficulties arose during the interviews, we wanted more extensive data about students' mastery of vocabulary before recommending IDE changes. To study this question in more detail, we extracted the terms used in the most frequently-presented error messages in our 6-week data set. Table 1 shows the 15 technical vocabulary words in the 90th-percentile of this list. We then developed a short quiz that asked students to circle instances of 5 specific words from this list in a simple piece of code. We administered the quiz at three different universities: WPI, Brown, and Northeastern, receiving 90, 32, and 41 responses respectively. At each university, students had used DrRacket for at least a couple of months before taking the quiz. As the quizzes were anonymous, we were not able to compare quiz performance with our coding data from the recorded editing sessions.

The results are roughly similar across all three universities (see Figure 3). 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 question whether students are able to make sense of the error messages. While students could have conceptual understanding of the messages without the declarative understanding of the vocabulary, our follow-up quiz

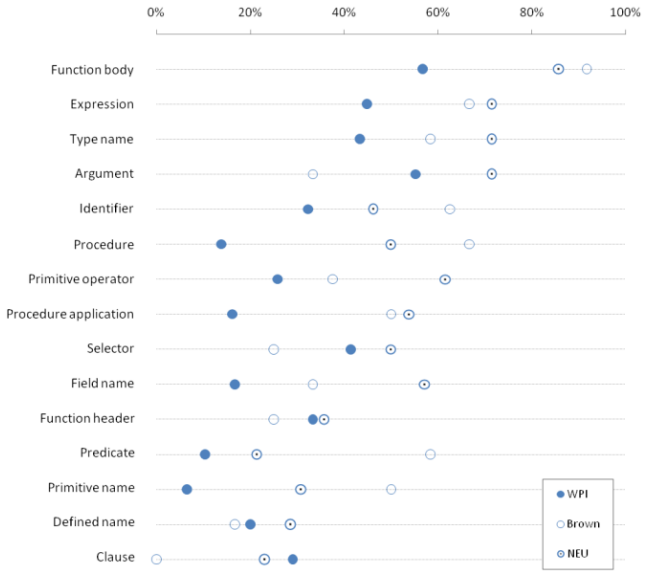


Figure 3. Average percent correct on the vocabulary quiz

(discussed in Section 6.) provides some evidence against this possibility.

Do Students Learn Vocabulary from Lectures?

Class lectures, as well as the IDE, discuss code via a set of terms. One might reasonably expect that students would perform better on such a quiz if the terminology used in their lectures matched those used in the IDE's error messages. To confirm this, we asked the professors who administered the quiz which of the terms from Table 2 they had used in class to describe code. Whenever a word used by DrRacket 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 to which this is caused by their poor command of the vocabulary?

	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 2. In-class word use

Number of questions of this type	*Syntax of cond (expected)* 15	*Syntax of cond (found)* 10	*Syntax of function calls* 14	*Syntax of define* 6	*Runtime type* 3
Correct	180	113	206	82	42
'Expected' mistake	96	N/A	N/A	N/A	N/A
Right-kind, wrong-instance	N/A	11	20	11	20
Wrong kind	91	120	122	56	15
No answer, inscrutable	26	18	21	9	4

Table 3. Results from the 'what's circled' questions. Each column corresponds to a different class of error messages; the number of quiz questions per class across the three quizzes appears under the class description. As 26 to 27 students completed each question, the total answers per column lies in the range of the number of questions multiplied by 26 or 27. The rows characterize the correctness of the answers. The term "kind" in the row descriptions refers to "grammatical term" (such as "argument" or "function call"). Details appear in the prose.

- 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 DrRacket 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 [2])? Lecture time is limited, as are homework contact hours; could the error messages help teach the vocabulary?

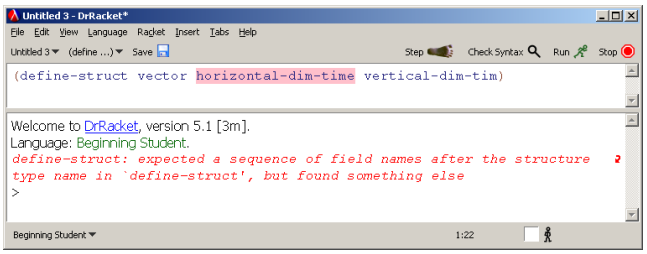
All three professors agreed that the mismatch between their vocabulary usage and DrRacket's was contrary to their efforts to use consistent language in class. Moreover, once the issue was pointed out to them, they all agreed that adjustments were needed. In general, we suspect professors tend to forget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

6. Vocabulary and Highlighting in Context

Our initial probes into highlighting and vocabulary raised additional questions that would impact recommendations for IDEs. We developed a quiz to explore two of these questions: (1) whether students understand what highlights refer to when explicitly asked about it, and (2) whether students can link vocabulary words to code fragments in the context of an explicit program and error message (with highlighting). The new quiz presents students with a series of error message screenshots, as shown in Figure 4. In the first kind of question (Figure 4(a)), students are asked to circle the phrase in the error message text that corresponds to what is highlighted. In the other kind of question (Figure 4(b)), we circled several terms in the error-message text and asked the students to box off the corresponding expression in the source program, or to cross out the term if it did not correspond to any particular expression.

To further explore the roles of vocabulary and highlighting, we

(a)



(b)

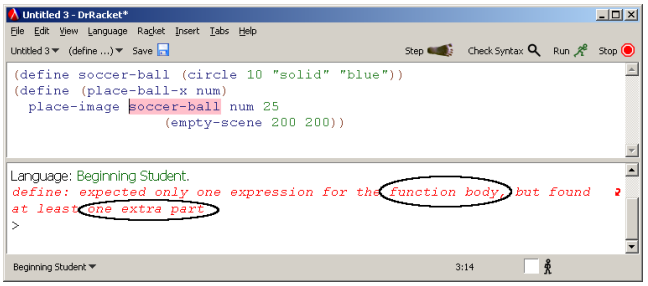


Figure 4. Highlight interpretation quiz: (a) What term is highlighted? (b) What (if anything) do these terms refer to?

also designed two different guides to accompany the quiz. One guide was a standard textual glossary of all vocabulary terms used in the messages within the quiz. The other guide gave two samples of code where each demonstrative noun phrase in the error message text was highlighted in the same color as the code fragment it was referring to; Figure 6 in Section 7 shows one of these samples. Each student received at most one of these two guides (i.e., some received none) along with their quiz.

Methodology

Nearly all programs used in this study were taken from our dataset of actual student errors. In one case, we created our own example with the same high-level structure, but a simplified key expression in order to assess the impact of a simpler term on error comprehension. We chose programs that exercised vocabulary terms that we knew to be problematic and programs on which we had seen poor student performance in the past. There were 20 programs with errors in our initial pool. We made three versions of the quiz, each asking 'what is highlighted' for 2 of these programs, and 'what code does this term reference' for 5 of these programs. For the latter, we asked about 2–4 terms per program, for a total of about 18 questions across 7 programs per quiz. We refer to these as 'what's circled' questions in the rest of this section. Combining each quiz version with each of the three possible guides (just described) yielded 9 distinct quizzes.

We administered the quizzes to students immediately after one of the weekly course labs at WPI; the quiz did not consume instructional time. Each participant received $5 in exchange for a completed quiz. All quizzes were anonymous. As our study was exempt from IRB procedures, we did not issue formal consent forms, though we did describe the study and our uses of data to students in the course via email the previous day. A total of 79 students submitted completed quizzes. The quizzes were done entirely on paper, rather than through DrRacket.

Results

In analyzing the data from the new quiz, we initially checked whether the style of the vocabulary guide affected the student's performance on the quiz. This question did not yield a statistically reliable answer: while the raw data suggested that some "better" answers were slightly more common for students who received the color-coded guide, statistical analysis could not demonstrate that those differences were due to more than chance. On the other hand, we did not have enough data to argue for a lack of effect either (this involves a different statistical calculation which depends on both the magnitude of the differences and the size of the data set). Given the lack of reliable results, we chose not to report raw data on the effect of the guides in this paper.

Subsequently, we ignored the different vocabulary guides and studied the data aggregated across the guides. Then we aggregated further by combining the data for problems with similar error messages (syntax errors on function definition, on conditionals, on structure definition, on function calls, and runtime errors). The results presented here are from the data aggregated across message types. Table 3 shows the results for the 'what's circled' questions, and Table 4 shows those for the 'what's highlighted'. We discuss the contents of the tables through the following descriptions of several interesting patterns in the data.

Found/Expected Confusion

As the examples in Figure 4 demonstrate, DrRacket's errors follow a common structure of

<construct>: *expected* <expr type> *but found* <X>

where <expr type> describes in grammar terms what the parser expected to find, or the expected data type for a runtime error; and <X> describes the kinds and number of expressions found, or a generic "something else", or, in case of a runtime error, the specific value given.

The grammatical structure of the message (in English) suggests that the terms in <expr type> do not appear in the code, while those in <X> do. We were therefore surprised at the frequency with which students thought the highlighting referred to something in the <expr type> portion (the orange highlighting in Table 4 (a) to (d)). Only one class of the 'what's circled' questions admitted a similar mistake (those that do not have "N/A" in the "expected mistake" row). Those questions asked students to identify which (if any) part of the code corresponds to the words *clause*, *question*, and *answer* in the expected half of the error message "*cond: expected a clause with one question and one answer, but found…"*. The leftmost data column of Table 3 shows that students answered correctly (no corresponding code) only 180 times; 96 times, they circled the right kind of expression but not

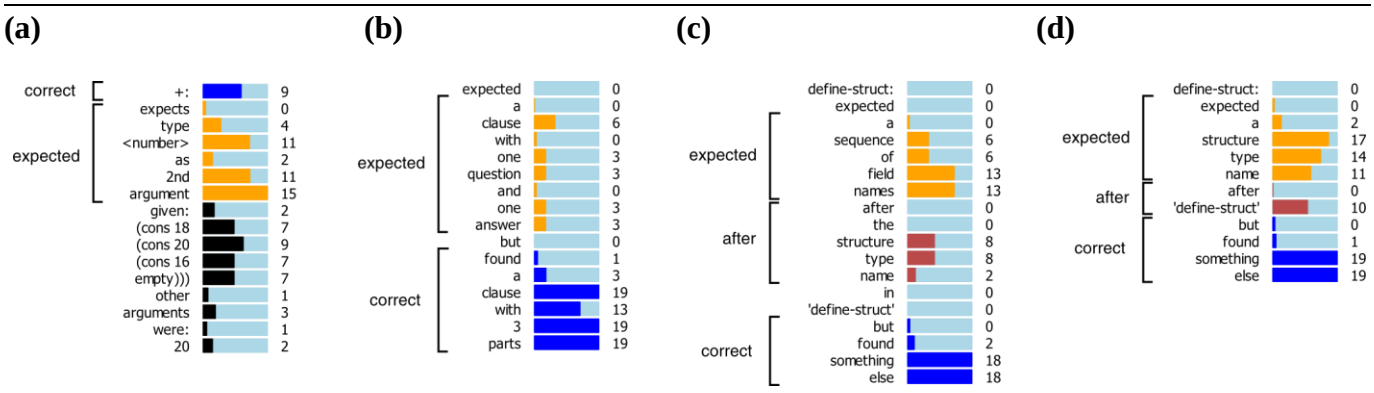


Table 4. Results from 'what's highlighted' questions. Both colors and brackets (for black-and-white viewing) characterize possible responses. Correct answers are in blue, answers in "expected" clauses are in orange, answers in a positional phrase (specifically, "after") are in red, and otherwise incorrect answers appear in black. Charts (a) and (c) each combine data from two questions with different code but the same error message.

the one referenced in the message, and 91 times they circled something completely unrelated to the term. This strongly suggests that many students are not grasping the grammar of the messages. A student remark in a comment area at the end of the quiz emphasizes this confusion:

> I wasn't confused with any of the words, but using language like "expected ____ found something else" is boths useless and insignifigant (sic)

Location versus Description

Two questions gave students programs with malformed structure definitions, including the one shown in Figure 4(a). The error messages concerning the syntax of defining structures take the form:

define-struct: expected <X> *after* <Y> *but found something else*

where Y refers either to the term define-struct itself, or to the type name introduced in the define-struct (and X is another term accordingly). Many students made the expected/found error just discussed on these questions, but a different answer was also common: circling "after <Y>" (Table 4(c) and (d)).

At first glance this seems incorrect, as the highlighting is meant to refer to "something else". However, there is some logic to a student selecting "after": it is an accurate description of the position where the highlight occurs, as opposed to a description of what is highlighted. None of the other error messages we tested had location terms. For students looking to an error for guidance as to where to edit, however, it makes sense that they would look for highlighting to reference location-oriented terms.

Vocabulary in Context

This new quiz gives us a second look at the students' command of the vocabulary. Our original vocabulary quiz (results in Figure 3) showed that the students had poor command of the vocabulary when queried outside of the context of an error message. If we see similar struggles when testing students on vocabulary in the context of an error message, then terminology is likely to be a factor in students handling error messages poorly.

Results on the 'what's circled' questions that did refer to source expressions appear in the four rightmost columns of Table 3. The percentage of correct answers hovers around or below 50% on each of these questions, but that figure tells only part of the story. Some questions had more than one source expression that illustrated the term we asked them to identify: answers that circled a valid instance of the term, but not the one referenced in the message, are tallied in the 'right kind, wrong instance' row. In contrast, the 'wrong kind' row tallies responses in which the circled code was not an instance of the requested term (such as a student circling a clause when asked for an argument). The 'wrong kind' mistakes dominate the 'right kind, wrong instance' ones. This gives us further confidence that the vocabulary truly is problematic for students, even in the context of error messages.

Highlights as Referents

One question, shown in Figure 5, yielded very few correct answers (data in Table 4(a)). Upon closer inspection, we realized that the question we asked had no good answer since the error message's text never refers to the highlighted expression. DrRacket highlights the entire call to +, but the closest referent to that is the very start of the error message (the "+: …" part).

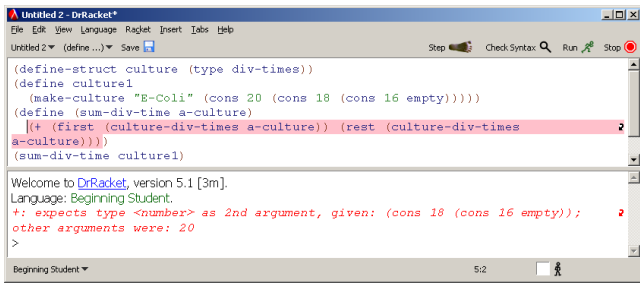


Figure 5. A highlight with no clear referent in the text

DrRacket has highlighted the expression whose evaluation generated a runtime error. While this is reasonable linguistically, it is inconsistent with the idea that highlights should always correspond to something mentioned in the message. To handle the latter, the error should highlight the second argument rather than the entire call to +. If we hope to teach students guidelines for working with error messages such as "figure out what term has been highlighted", we must ensure that our error messages only highlight such referents. More generally, we must ensure the highlights have a semantics that is both internally-consistent and consistent with the strategies we wish to teach students.

Lessons Learned

On the whole, students performed reasonably well on questions about what term was highlighted (the blue bars in Table 4). In most cases, correct answers dominate incorrect ones, with errors related to "expected/found" confusion dominating the errors. The one question with a large number of outright wrong answers combined both a runtime error (which students often struggle to understand) with no clear referent (as described in the previous observation). If we discount this question and focus on compile-time errors, students seem able to match highlights to error-message terms. This suggests that the "edit here" interpretation has some source other than mere confusion, such as students seeking an easy default behavior or giving too much attention to the highlight due to its strong visual appearance.

The vocabulary results, as already discussed, are weaker. This suggests that vocabulary and message grammar are perhaps bigger factors than highlighting in students' difficulties with responding to errors. Were this truly the case, however, we might have expected to yield statistically significant improvements from giving students vocabulary guides as part of the quiz. It would be worth conducting an additional study with a more carefully-designed vocabulary guide, just to double-check whether there is indeed an effect that either our quiz or data set size did not uncover.

Were we to conduct the concrete highlighting and vocabulary quiz again, we would also make several changes based on what we saw in the current data. We originally chose quiz questions to test a range of terms that we thought were problematic, as well as different kinds of highlighted expressions (concrete versus abstract, parentheses versus complex expressions, direct versus indirect references). In hindsight, we conflated runtime errors (which we suspect are generally problematic) with other problems (such as the question with no good answer). In a new quiz, we would isolate runtime errors, test a wider range of highlight meanings, and add more questions with both location and descriptive terms that could correspond to expressions. While the basic scientific principles about changing only one variable at a time sound easy in theory, we find that conflation of concerns often reveals itself only in the presence of concrete data.

7. Recommendations

Ultimately, we are pursuing this research to help improve how DrRacket and other IDEs for beginners handle error messages. While many research questions remain (as outlined in earlier sections), our results do suggest several concrete proposals that we intend to implement and validate in the near future.

In developing these proposals, we adhere to two fundamental principles about error message design:

- 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 DrRacket as an oracle, proposed solutions could lead them down the wrong path. 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. In particular, it reminds us to carefully consider how students might interpret a highlight.

With the principles in hand, we turn to our proposals:

Simplify the vocabulary in the error messages.

DrRacket'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. On the whole, DrRacket is accurate and consistent in its use of technical vocabulary. However, some of its terms are overly precise relative to terms that students already know. For example, DrRacket uses the term "identifier" rather than "variable", even though "variable" is the term students are used to from high school math classes. We suspect that beginning students would fare better with familiar vocabulary, even if it comes at a slight cost in precision (particularly, that "variable" should only apply to a name whose value can be changed via side-effects).

Table 5 shows a proposed simplification of the vocabulary that currently appears in DrRacket's teaching languages (Beginner through Advanced). We are preparing to deploy this simplified vocabulary to students. In the interest of timely progress, we have chosen not to do extensive validation of this simplified list. Doing such validation well would ideally require us to determine the "best" set of terms for introductory students, an effort that should look across languages, across high-school mathematics texts, and across the many populations of students who use beginner IDEs. While we feel this is important work (and hope someone else takes up the charge), it is tangential to our focus on understanding how students interact with error messages.

One could reasonably argue that the current (non-simplified) vocabulary is appropriate, or even desirable, for students at the end of CS1. Certainly, we should expect that CS1 students learn a good deal of vocabulary over the span of the course. We note that this same observation is the key principle underlying DrRacket's language levels: protect students from constructs that they have not yet learned. Perhaps the same principle should apply to the design of error messages within those language levels: use a vastly simplified vocabulary in the Beginner language, then relax the messages to use more complex but precise vocabulary as the course progresses.

Old term	New term
Procedure **Primitive name** **Primitive operator** **Predicate** **Selector** **Constructor**	Function
Name **Identifier** **Argument** **Defined name**	Variable, *for value definitions, mutable variables, and for formal arguments (in function definitions)* Argument, *for actual arguments (in function calls)*
Sequence	At least one
Structure type name	Structure name
Question—answer clause	A clause is expected to have a question and an answer
Function header **Primitive name** **Keyword** **Type**	*These words and notations have been removed entirely and reworded in terms of other vocabulary words.*
<type name> ***'literal word'***	*These punctuations are no longer used in the error messages.*
Function body **Expression** **…**	

Be explicit in errors about inconsistencies.

When a student uses a function or constructor inconsistently with its definition, the current DrRacket error messages highlight the expression for the usage, but not the definition. Such errors can be difficult to fix when the problem lies in the definition instead. The highlight effectively has an over-focusing effect, steering students away from the possibility that the problem lies in the other half. The best student we interviewed had learned to avoid the over-focusing effect, as noted in 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.*

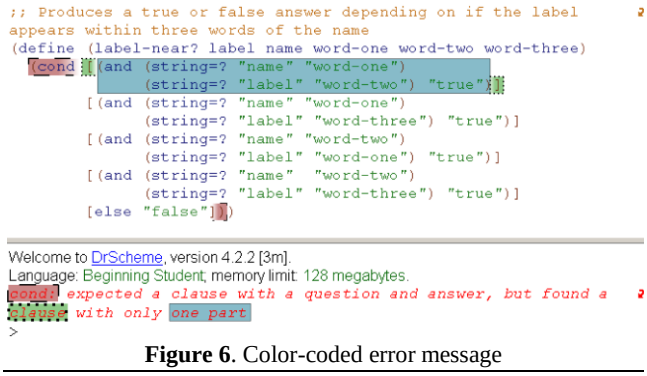


Figure 6. Color-coded error message

Once again, the student's response indicates an "edit here" instinct towards the highlight, even though he had learned that was not always the correct response. An IDE cannot be omniscient about which of the definition or use is incorrect in a program. It can, however, avoid biasing a student towards one of the two through its error-reporting mechanism.

The "*expected … found …*" structure of DrRacket's error messages adds to the bias imposed by the highlights in errors about inconsistencies. When a use contradicts a definition, the *expected* clause refers to the definition, which suggests that the definition is fine and the use problematic. Ideally, this form should be reserved for definitions provided by DrRacket or a library, but not used for inconsistencies arising from user-defined functions or constructors.

Inconsistencies (and biases in their presentation) arise in other circumstances outside of definitions and use of functions. The grammar of the language imposes constraints on the position where certain forms can appear. When one such constraint is violated, the error messages of DrRacket state that one endpoint of the constraint is correct and the other one is wrong, but the choice is either arbitrary or dictated by accidents of the parsing strategy used. For example, in the following code fragment (from a student not involved in our studies) the student has correctly defined a constant for an image, but erroneously wrapped the lookup of that constant with parentheses.

```
(define LIGHT_RED (circle 20 "outline" "red"))
(define TRAFFIC_LIGHT
    (place-image (LIGHT_RED) …)
```

The DrRacket error message reports:

> *function call: expected a defined name or a primitive operation name after an open parenthesis, but found something else*

This message assumes that the parentheses are correct, and that a function call, not the use of the constant, was intended. But the reversed assumption would be equally valid, and in this case it would match the student's intention. By favoring one half of the inconsistency, the message violated our core principle that error messages should not suggest a particular edit to the student. Indeed, in response to this error the student inlined the constant's definition. This left a function call where one "was expected" and made the error go away, but it also pushed the student away from a correctly designed program.

In the case of errors reporting inconsistencies, one option (suggested by Michael Jackson, the software engineering researcher) would be to simply ask the student what they were trying to do (e.g., use a constant or call a function), then give an appropriate detailed message based on that goal. Starting from student input would avoid having the error message prompt the student towards a particular solution, while letting us give a more precise message than "you've used a non-function constant as a function here". In general, we suspect that we should be more interactive in providing error feedback to students. This will be a key component of our work moving forward.

Help students match message terms to code fragments.

Error messages contain many demonstrative 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, DrRacket's highlighting achieves this effect, too (as with unbound identifiers or unmatched parentheses). However, messages often contain multiple terms, while DrRacket currently highlights only one code fragment.

We therefore propose that error messages highlight every definite reference and its corresponding code with a distinct color. The quiz guide discussed earlier (displayed in Figure 6) was taken from a preliminary mockup of this idea. Each demonstrative reference in the message uses a colored highlight to point to a specific code fragment (in this paper the colors were 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 1), and it gives students a chance to learn the vocabulary by example (in Figure 6, 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 6) until we realized it violated the second principle. We are currently refining this design with intent to deploy it experimentally next year.

Treat error messages as an integral part of course design.

Professors must ensure their curriculum aligns with the content of the error messages, just like math professors ensure their notation matches that of textbook. While this may sound obvious, calibrating lecture notes against DrRacket's error messages had not occurred to the instructors who administered our vocabulary quiz (all of whom are veteran CS1 professors).

In the specific case of DrRacket, which accompanies particular textbooks, we intend to develop vocabulary conventions for talking about Beginner Student Language (BSL) 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.

Teach highlighting (and other error components).

Instructors and books should explain the components of error messages to students. While this also may seem obvious, few of us who teach regularly with DrRacket had noticed that interpreting the highlights required explicit instruction. IDE developers should provide guides (not just documentation buried in some help menu) about the semantics of notations such as source highlighting.

Our results further suggest that students need help reflecting on the semantics of the textual message components as well.

Students' difficulties with the "expected/found" distinction, for example, are arguably as much about their failure to carefully read and reflect on the message text. However, as instructors we must bear in mind that the visual cues (such as highlights) may dominate the students' focus. Furthermore, errors are stated from the perspective of the IDE, not the student. When a student sees "expected/found", they might not initially understand the process through which DrRacket comes to "expect" anything in particular. Lacking a mental model of this process, students default to one (such as "edit here") that the messages then work against until the student's model matures.

More generally, there is a question about how to teach CS1 students about IDE features that arise from advanced CS concepts such as parsing. Programming texts frequently present formal grammars (through syntax diagrams [3] or textual BNF) to help explain language syntax; some include exercises on deciphering text through grammar rules [4]. Unfortunately, the highlighting undermines 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 an early stage of their computing education.

Beware Libraries.

As we examined error messages, we noticed a significant source of additional inconsistencies: libraries! We realized that DrRacket does not have style guidelines for library authors, who were hence each creating their own language universes. Clearly each programming environment needs an accompanying *Elements of Error Message Style*, and we have one in development for DrRacket: a four-page sheet that lists allowed words, disallowed words, and grammar rules for error messages.

8. Related Work

The principles of HCI frame general discussions on the design of pedagogic programming languages, as well as on the design of error messages specifically [5]. These reflections informed our work. Many researchers have studied various programming tools, such as debuggers; we focus here only on studies involving error messages themselves.

Alice [6] and BlueJ [7] 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 [7,8]. These difficulties have led professors to develop supplemental material simply to teach students how to understand the error messages [9]. One evaluation of BlueJ asked the students whether they found the messages useful [10]. 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 DrRacket, despite their struggles to respond to them. That said, our study shows that DrRacket's errors are still a long way from helping the students, and other recent work [11] also presents evidence of this.

There are still relatively few efforts to evaluate the learning impact of pedagogic IDEs [12]. Gross and Powers survey recent efforts [13], including, notably, those on Lego Mindstorms [14] and on Jeliot 2000 [15]. 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 [16], 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 [18], Lerner S. et al explains the type error messages of ML by stating a modification that would make the code type [19], and STLFilt rewrites the error messages of C++ [20]; none has been evaluated formally against real students behavior.

Early work on the pedagogy of programming sought to classify the errors novice programmers make when using assembly [21] or Pascal [22]. More recent work along the same lines studies BlueJ [23,24], Eiffel [25], and Helium [26]. Others have studied novices' behavior during programming sessions. This brought insight on novices' debugging strategies [27], cognitive inclination [28], and development processes [29]. Our work differs in not studying the students' behavior in isolation; rather, we focus on how the error messages influence the students' behavior.

Coull [30], as well as Lane and VanLehn [31] 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. [32] 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.

9. Perspective and Ongoing Work

Studying students' fine-grained interactions with error messages is humbling, but highly informative. The DrRacket team (to which we belong) has spent years developing an IDE tailored for beginning students. Many of DrRacket's current features arose from our observations of student struggles in classes and labs. However, it wasn't until we collected a large, concrete, detailed data set on students' interactions with errors that we appreciated the problems reported in this paper.

Our goal with this paper is twofold: first, to advocate for and illustrate this kind of work to others who develop IDEs for particularly audiences; second, to raise design issues that should apply broadly across IDEs. Our recommendations about color-coded highlights, consistent vocabulary, non-biased error messages, and pedagogy are not specific to Racket. 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).

Going forward, we need to deploy the recommendations presented here, then measure their impact on students. Impact can take several forms. At one level, we are simply interested as whether students make fewer errors using our revised IDE. We should also look at other metrics, however, such as whether the number of iterations needed to fix an error decreases and whether students experience less frustration with the revised messages. DrRacket also has a wide user base, including middle-school students in after-school programs, high school students, and college students. We cannot assume that one style of errors will fit all.

Our work also leaves open interesting questions about the interaction of parsing strategy and error message content. At a recent talk on this work, a colleague asked whether we had contrasted DrRacket's expression-highlighting approach to a more conventional compiler error based on expected tokens. We have also seen examples in our dataset on which the error message appears counterfactual unless the user understands the parsing strategy. The latter is obviously unsuitable for beginners, but the fix is not obvious, as a different parsing strategy would expose odd errors in other programs.

In general, the design of error messages and their interaction with programming language technology is a wide open and fascinating area. Developers of IDEs for beginners are working on several ideas that minimize the effect of programming errors, but these often come at a cost of scalability as students (quickly) advance to writing more sophisticated programs. Our work shows that IDE interface design is not simply an HCI question, but one with deep roots in programming languages technologies. We invite more PL researchers to join us for the ride.

Acknowledgments

Danielle Gilmore collected and coded the data for the quiz in Section 6. 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.

10. References

[1] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the Effectiveness of Error Messages Designed for Novice Programmers," *Proceedings of the Symposium on Computer Science Education*, 2011.

[2] P.I. Pavlik, N. Presson, G. Dozzi, and B. Macwhinney, "The FaCT (fact and concept) system: A new tool linking cognitive science with educators," *Proceedings of the Conference of the Cognitive Science Society*, D. McNamara & G. Trafton, eds., 2007, pp. 397-402.

[3] N. Wirth, "The Programming Language Pascal," *Acta Informatica*, vol. 1, 1971, pp. 35-63.

[4] S. Bloch, *Picturing Programs*, College Publications, 2011.

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

[6] B. Moskal, D. Lurie, and S. Cooper, "Evaluating the effectiveness of a new instructional approach," *Proceedings of the Symposium on Computer Science Education*, 2004, pp. 75-79.

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

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

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

[10] K. Van Haaster and D. Hagan, "Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool," *Issues in Informing Science and Information Technology*, vol. 1, 2004, pp. 455-470.

[11] M. Crestani, "Experience report: growing programming languages for beginning students," *Proceedings of the International Conference on Functional Programming*, 2010.

[12] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, "A survey of literature on the teaching of introductory programming," *ACM SIGCSE Bulletin*, vol. 39, Dec. 2007, pp. 204-223.

[13] P. Gross and K. Powers, "Evaluating assessments of novice programming environments," *Proceedings of the International Workshop on Computing Education Research*, New York, New York, USA: 2005, pp. 99-110.

[14] B.S. Fagin and L. Merkle, "Quantitative analysis of the effects of robots on introductory Computer Science education," *Journal on Educational Resources in Computing*, vol. 2, 2002, pp. 1-18.

[15] R.B.-B. Levy, M. Ben-Ari, and P.A. Uronen, "The Jeliot 2000 program animation system," *Computers & Education*, vol. 40, 2003, pp. 1-15.

[16] T. Flowers, C. Carver, and J. Jackson, "Empowering students and building confidence in novice programmers through Gauntlet," *Frontiers in Education*, vol. 1, 2004, p. T3H/10 - T3H/13.

[17] J. Jackson, M. Cobb, and C. Carver, "Identifying Top Java Errors for Novice Programmers," *Proceedings of the Frontiers in Education Conference*, 2005, p. T4C–24.

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

[19] B.S. Lerner, M. Flower, D. Grossman, and C. Chambers, "Searching for type-error messages," *Programming language design and implementation*, 2007, p. 425.

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

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

[22] J.C. Spohrer and E. Soloway, "Novice mistakes: are the folk wisdoms correct?," *Communications of the ACM*, vol. 29, 1986.

[23] N. Ragonis and M. Ben-Ari, "On understanding the statics and dynamics of object-oriented programs," *ACM SIGCSE Bulletin*, vol. 37, 2005, pp. 226-230.

[24] M.C. Jadud, "A First Look at Novice Compilation Behaviour Using BlueJ," *Computer Science Education*, vol. 15, Mar. 2005, p. 25–40.

[25] M.-H. Ng Cheong Vee, K. Mannock, and B. Meyer, "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, pp. 54-58.

[26] J. Hage and P.V. Keeken, "Mining Helium programs with Neon," *Technical Report, Department of Information and Computing Sciences, Utrecht University*, 2007.

[27] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander, "Debugging: the good, the bad, and the quirky — a qualitative analysis of novices' strategies," *Proceedings of the Symposium on Computer Science Education*, 2008, pp. 163-167.

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

[29] M.F. Köksal, R.E. Başar, and S. Üsküdarlı, "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, pp. 103-110.

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

[31] H. Chad Lane and K. VanLehn, "Intention-based scoring: An approach to measuring success at solving the composition problem," *Proceedings of the Symposium on Computer Science Education*, New York, New York, USA: 2005, pp. 373-377.

[32] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler Error Messages: What Can Help Novices?," *Proceedings of the Technical Symposium on Computer Science Education*, 2008, pp. 168-172.

11. 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))

;; 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)])]))
```

Q0