Error Messages Are Classifiers

A Process to Design and Evaluate Error Messages

John Wrenn
Computer Science
Brown University
USA
jswrenn@cs.brown.edu

Shriram Krishnamurthi
Computer Science
Brown University
USA
sk@cs.brown.edu

Abstract

This paper presents a lightweight process to guide error report authoring. We take the perspective that error reports are really *classifiers* of program information. They should therefore be subjected to the same measures as other classifiers (e.g., precision and recall). We formalize this perspective as a process for assessing error reports, describe our application of this process to an actual programming language, and present a preliminary study on the utility of the resulting error reports.

CCS Concepts • **Software and its engineering** → *General programming languages*; *Language features*; *Compilers*;

Keywords error messages, highlighting, classifiers, precision and recall, Pyret

ACM Reference Format:
John Wrenn and Shriram Krishnamurthi. 2017. Error Messages Are Classifiers: A Process to Design and Evaluate Error Messages. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3133850.3133862

1 Introduction

Error messages are a crucial communication channel between the computer and programmer. They generally indicate that something has gone wrong, what has gone wrong, where, and try to offer context or other information to help the programmer correct the error. Over the decades, error messages have become progressively more elaborate, from error codes that needed to be looked up in manuals to having multi-colored, hypertext, expanding, and other elements.

Onward!'17, October 25–27, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5530-8/17/10…$15.00
https://doi.org/10.1145/3133850.3133862

Error messages are particularly important to beginning programmers. First, they may already be intimidated by the act of learning programming, so these messages must not further turn them off. Second, errors are one of their first experiences with the computer as a harsh taskmaster: prior to that, their linguistic communication is handled by humans, who can often fill in missing context or (mis)interpretation. Third, they have a more limited vocabulary and knowledge than the experienced programmer, making the act of reading messages that much harder. Finally, there are often moments when beginners realize they not only have a mistake in their program, but may have misunderstood the functioning of the *language* itself. For all these reasons, error messages have been a focus in computer science education [10, 15, 17–19].

Because error messages are a human-computer interaction element, they should be subject to user studies and other forms of evaluation. Indeed, many researchers have performed extensive studies of how programmers (particularly students) react to different forms of messages [10, 15, 19, 20]. These studies have shown how students struggle with the vocabulary of errors [18], how the affect of actors involved in presenting the errors matters [10, 15, 20], and so on. One particularly significant finding is that, when an error message points to a fragment of code (as most do), students tend to adopt an "edit here" mentality [18]. Unfortunately, they behave this way even when the error message text might suggest otherwise, and in fact the highlighted portion is the only part of the program that is definitely correct, thereby making their program significantly worse with every edit!

Such studies show the importance of performing rigorous user studies on errors after they have been deployed. However, this creates a chicken-and-egg problem. To do these studies, one usually needs a sizable base of users who have had some experience with the language—but getting to such a community of programmers often depends on the language being usable in the first place, including in its presentation of errors. Therefore, waiting for a study gives a language creator no guidance while they are building their implementation. Furthermore, many users may have had to endure a significantly sub-standard experience until a study was executed and its findings implemented. Finally, studies are expensive, difficult to administer well, and time-consuming, so it can be hard to get feedback in a rapid prototyping cycle.

Fortunately, there are also other traditions in human-computer interaction for evaluating interfaces, and in many areas, researchers have created lightweight evaluation rubrics that a design team can itself implement and apply during the development or upgrade process. A good illustrative example is the work on "cognitive dimensions of notations" [4], which offers over a dozen dimensions along which a notation can be evaluated. Other good examples include "cognitive walkthroughs" [21], the "system usability scale" [5], and so on, which range in weight of effort and in how early in the process they can be administered. These techniques are not intended to replace a thorough user study, but, rather, to complement it and in some cases offer tools that can be used during the design and implementation phases, resulting in the deployment of better artifacts.

Our goal in this paper is to provide a similar process that can be used by error report designers during the design and implementation phases. We present this in multiple stages. First we present our perspective of error reports as *classifiers of information* (Section 2). Next, we discuss the types of information that comprise an error, which must somehow be presented to the user (Section 3). We formalize our perspective as a process for assessing error reports (Section 4), demonstrate its application on a report for a simple error (Sections 5 and 6), and adapt it to the assessment of a highlighting-based program selection mechanism (Section 7).

While the content of this paper is applicable to any programming environment that reports errors, we concreteize these issues in the context of an actual language: Pyret [6]. We conclude with a discussion of how we applied this framework to overhaul of Pyret's error reports, and a preliminary study to confirm that the consequences of following this process did not negatively affect users (Section 9).

2 Shifting From Text to Classification

Some past research has presented expansive, itemized prescriptions for how compilers should communicate errors to users: for instance, Horning [13], Shneiderman [23], and Traver [25]. These efforts to identify principles that characterize good reports occupy the same space of broad interface design principles for error reports as frameworks like cognitive dimensions do for notation.

However, although Horning and Traver argue that error messages should first direct the programmer accurately to "the location" where the error occurred, the bulk of all three authors' arguments is that information should be swaddled in a "good" explanation, for various definitions of that term. Indeed, many error messages projects follow a similar (sometimes unwritten) focus on the *language* of the error message: e.g., Shneiderman writes [23],

> Writing good messages, like writing poems, essays, or advertisements, requires experience, practice, and a sensitivity to how the reader will react.

and that focus persists into modern times [15, 18].

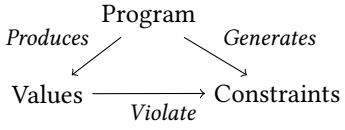
However, studies show that students often do not understand the vocabulary of messages [18], and anyway, obey an "edit here" mentality based on what code is highlighted [18]. Therefore, an equally important component of error reports is what code they highlight. The act of selecting code implicitly draws the reader's attention to those fragments *and* takes attention away from the fragments *not* highlighted. In short, an error message report is a *classifier* of code: it effectively classifies code as "look here to start fixing the error" and "don't look there to start fixing the error".

What properties might we want of such a classifier? Ideally, we would expect it to satisfy both *completeness* and *soundness*: it should highlight all relevant code, and highlight nothing but relevant code. If it fails to highlight all relevant code, it might not call attention to the place where the problem lies (as previous studies have shown). Similarly, if it highlights code not relevant to the error, it might cause a blameless fragment to be unnecessarily altered. While this paper gives special consideration to selecting program fragments, we apply this principle broadly: an error report should present relevant information and should not present irrelevant information.

Soundness and completeness are, however, too binary. A better measure would be their numeric counterparts, *precision* and *recall*, which are anyway standard measures applied to classification algorithms. Furthermore, real error message authoring involves tradeoffs, and authors sometimes sacrifice one property for the other based on their judgment (and, presumably, subject to subsequent studies). Therefore, the value of these measures is as a guideline to authors, in particular helping quantify differences between choices. This approach complements, but does not replace, other lightweight evaluation frameworks. An error report author must decide *what* information to present and *how* to present it. The cognitive dimensions of notations, for instance, address the latter; our work addresses the former.

3 A Model of Error Information

In concluding that an error has occurred, the language considers at least three information sources, with interesting interactions. Consider the case of run-time errors: there is the source *program*, there are some run-time *values* computed by terms in that program, and there are *constraints*—coming from the semantics—that those values violated, resulting in the error. Pictorially,



While the values arise directly from terms in the program, the constraints may cross-cut the values (e.g., an array's size and an index are each fine on their own, but their out-of-bounds juxtaposition results in a violation).

The error report's job is to present all this information to the user in a way that is accurate, useful, and (hopefully) not overwhelming. Of these three, however, neither the evaluator nor the semantics can be changed, so values and constraints are effectively fixed; the programmer can only alter the program. This is why selecting the right program fragments is so important; however, the report must also relate these fragments to values and constraints to guide the programmer's edits.

Though this paper focuses on run-time errors, the above model applies more generally. For instance, type-checking errors also fall into this framework, with values being replaced by types. This also applies to static well-formedness checks (e.g., no duplicate binding of identifiers), where values are replaced by program fragments, and a program "contains" these fragments rather than "producing" them.

4 An Error Assessment Process

Given this information model, we propose a process for designing and evaluating error reports. This has four steps. First, the author of the error report *represents* the available information. Second, they must review these representations and *classify* each information element as relevant (or not) to repairing the error. Third, given this classification, they must *select* the information that is relevant and present it to the user by composing an error report. Finally, they evaluate the soundness and completeness of the selection with respect to the classification via the measures of precision and recall.

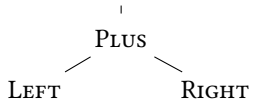
For instance, because errors should be reported in terms of the programs that caused them, we need a representation of these programs. At error authoring time we do not have a concrete program available, but we do have an understanding of the situation in which the error occurs. We *represent* this using a *partial and parametric AST* (P-AST): it is partial because it refers to just a fragment of the program, and it is parametric because it has holes that need to be filled in by the specific program instance. We see examples of this later in the paper (Sections 5 and 7). The P-AST represents the syntactic information available during error authoring. The author still has to *classify* which parts of the P-AST are relevant and which are not. Given the classification, we *select* the relevant fragments of the program by composing an error report. There are many technologies for selection. As discussed in Sections 5 to 7, some of them have better fidelity than others in presenting the classification, and must hence themselves be subject to precision and recall evaluation.

How to determine relevance is outside the scope of this paper as it is highly dependent on context. While a *perfect* heuristic would identify as 'relevant' *exactly* the set of information that is actually used to correct the error, predicting this is often infeasible: any given type of error can usually be corrected in several ways and the strategy that is ultimately employed will depend on the user's intent. If the designer lacks a good heuristic for relevance—as may be the case when there is limited structured information available (e.g., parse errors) or in complex algorithmic systems with non-local effects (e.g., full-program type inferencing)—then our process will be of limited use.

5 A Simple Application of the Process

To illustrate how to use the process, we begin with an extremely simple example. Consider a language with an overloaded binary + operator. Suppose in this language either both operands must be numbers, or if the left operand is a string, the right can be any value. These two constraints thus restrict the set of valid expressions. When the constraints are not satisfied, an error is reported. A program that induces this error and a concrete error report for that program are shown in Figure 1.

We *represent* the syntactic features that characterize programs which cause this error with a P-AST:



Using the information model presented in Section 3, we *classify* the available information as relevant or irrelevant and enumerate it. We identify relevant terms:

1. Plus
2. Left
3. Right

...constraints:

4. Add
5. Concat

...values:

6. v_{Left} (the value of the left operand)
7. v_{Right} (the value of the right operand)

...relationships between terms and constraints:

8. Plus imposes Add
9. Plus imposes Concat

...relationships between constraints and values:

10. Add constrains v_{Left}
11. Add constrains v_{Right}
12. Concat constrains v_{Left}

...and relationships between terms and values:

13. Left produces v_{Left}
14. Right produces v_{Right}

We then compare the information elements selected in Figure 1 against our enumeration of relevant information. This report does not convey any irrelevant information, therefore it is sound (a precision of 1). However, this report includes only some of the information that we identified as relevant: for instance, it omits all information relating to constraints (information elements 4, 5, 8, 9, 10, 11, and 12 are missing). In this context, precision is the fraction of information appearing in the report that also appears in our list of relevant

```
1  a = 5
2  b = 10
3  c = "cat"
4  d = "hello"
5  print(a + b +
6        c + d)
```

The binary plus expression failed on the values:
- 15
- *"cat"*

Figure 1. Running this program induces an error because a number is added to a string. The error report localizes the problem in the code with a (red) gutter-marker on line 5.

elements, and recall is the fraction of relevant elements that are conveyed by the report. Recall thus quantifies the incompleteness of this report:

$$Recall = \frac{|\text{relevant} \cap \text{selected}|}{|\text{relevant}|} = \frac{7}{14}$$

This simple example merely illustrates the process at work. In Section 7 we will see more sophisticated examples involving trade-offs.

6 Location Locution

We have made two insidious assumptions in our analysis of this report. First, we assumed that the readers of this report will be able to identify the relationships between the content of the report and syntactic terms of the program as well as we did. For instance, we assume that the readers of this report will infer that the value 15 corresponds to the left operand and not the right operand. We could precede each value with a phrase like "the left operand was", but that would assume that the reader could identify those referents in the source code—a tenuous assumption if the reader is, say, a novice programmer and unfamiliar with the meaning of "operand".

Second, we have assumed—despite knowing little about the lexical structure of the program—that the gutter marker on line 5 of Figure 1 is a sufficiently precise selector of code. Not only does line 5 contain more than one + operator, but the plus expression that failed is actually the one spread across two different lines.

These issues reflect two ways in which error report authors can fail to effectively select program fragments:

1. failure to effectively select (e.g., with gutter markers), because of assumptions about syntactic and lexical structure, and
2. failure to effectively select with textual references, because of assumptions about the reader.

In principle, the latter pitfall can be addressed by forcing all connections to be explicit. Unfortunately, this also has serious problems. If we were to follow *every* program reference with the corresponding source location, for messages that contained many textual references, this would dramatically increase the size and detail of error messages. For instance, the revised error report of Figure 1 would look like:

The binary plus expression at `some/directory/and/filename.arr:5:6-6:7` failed.

The value of the left side at `some/directory/and/filename.arr:5:6-5:11` was:
15

The value of the right side at `some/directory/and/filename.arr:6:6-6:7` was:
"cat"

A binary plus expression expects that either:
- that the left side is a string, or
- that both the left side and right side are numbers.

(This doesn't even follow *every* reference with its location!) We attempted such a system for Pyret but it rapidly became untenable. This experience suggests a tension between not providing enough information and not overwhelming the user. Instead, we implemented a variation of a system suggested by Marceau et al. [18]: in lieu of source locations, phrases in the error report prose that refer to syntactic fragments are visually associated with their referents by highlighting both the code *and* the phrase in a shared color. Figure 2 demonstrates this.

While, in principle, highlighting allows us to simultaneously connect *every* textual reference to a syntactic structure, technical limitations may prevent this from happening in full. Our implementation of highlighting did not support nested highlights, so a highlight over a term would prevent concurrent highlights in its subterms. In Figure 2, we resolve this by making "binary plus expression" an *on-demand* highlight: when the mouse is moved over this phrase, the other highlights in the code are hidden and the entire plus expression is highlighted.

7 Applying the Process to Highlights

With this highlighting mechanism, highlighting design decisions must be considered in two contexts. First, highlights of program fragments may be analyzed as a distinct channel of communication with the user. If selections of program fragments are interpreted by the user in lieu of the error report's prose [18], these selections should then be analyzed as if they *were* the error report prose. Second, highlighting can be analyzed in their role as a clarifier of the information provided in the error report prose; in the highlighting

```
1  a = 5
2  b = 10
3  c = "cat"
4  d = "hello"
5  print( a + b  +
6         c  + d)
```

The binary plus expression failed.
The value of the left side was:
15
The value of the right side was:
"cat"
A binary plus expression expects that either:

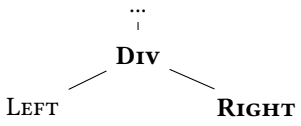
- that the left side is a string, or
- that both the left side and right side are numbers.

Figure 2. This error report uses highlighting to connect textual references to their referents in the program source. Clicking on any textual reference brings the relevant code into view in the editor.

mechanism we used, highlighting decisions affect both both the program and the prose. We therefore need a method to evaluate highlighting decisions in each of these contexts. Fortunately, our existing evaluation process can also be applied here. In particular, the P-AST is a good representation for reasoning about the consequences of highlighting.

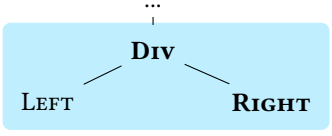
7.1 Represent & Classify

As before, we *represent* the available information as a P-AST, and classify its elements as relevant or irrelevant. All programs whose execution causes a divide-by-zero error will have at least one syntactic structure in common: a division expression. We represent this as a P-AST; the terms classified as relevant are denoted in bold:



7.2 Select

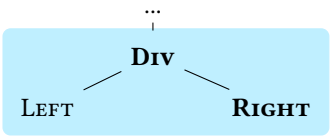
In what follows, we will assume that the compiler tags each node of the AST with the range of corresponding characters in the program. The edges of the AST and P-AST therefore represent both the structural and lexical 'contains' relationship of nodes to their children; i.e., the subterms of a node are *within* the lexical extent of that node. Consequently, when a structure is highlighted, its subterms will *also* be within the visual highlight. This consequence of highlighting is made evident by representing a highlight of a term on the P-AST as a box draw around that term's entire subtree. For instance, a highlight over the Div subtree will unavoidably envelope both the Left and Right subtrees:



7.3 Analyzing Program Highlights Independently

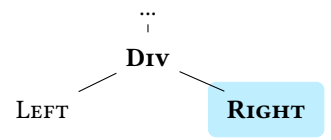
To analyze program highlighting decisions in the context of an "edit-here" mentality, we evaluate program highlights as if they *were* the error report: by computing the precision and recall of the program selections. In this context, precision reflects the fraction of highlighted P-AST nodes that are relevant and recall reflects the fraction of relevant P-AST nodes that are highlighted. In contrast to our application of the process in Section 5, selection decisions here often involve tradeoffs: a highlight of a relevant term may sweep up irrelevant elements in that term's subtree and our limitation of no-nested-highlights constrains the number of distinct highlights we are able to make.

For the P-AST of the divide-by-zero error, there are two promising ways that the P-AST may be highlighted: First, we could highlight the entire Div expression, thereby achieving perfect recall at the expense of precision:



$$Precision = \frac{2}{2+1} = \frac{2}{3} \qquad Recall = \frac{2}{2+0} = \frac{2}{2}$$

Alternatively, we could highlight only the Right subtree and achieve perfect precision at the expense of recall:



$$Precision = \frac{1}{1+0} = \frac{1}{1} \qquad Recall = \frac{1}{1+1} = \frac{1}{2}$$

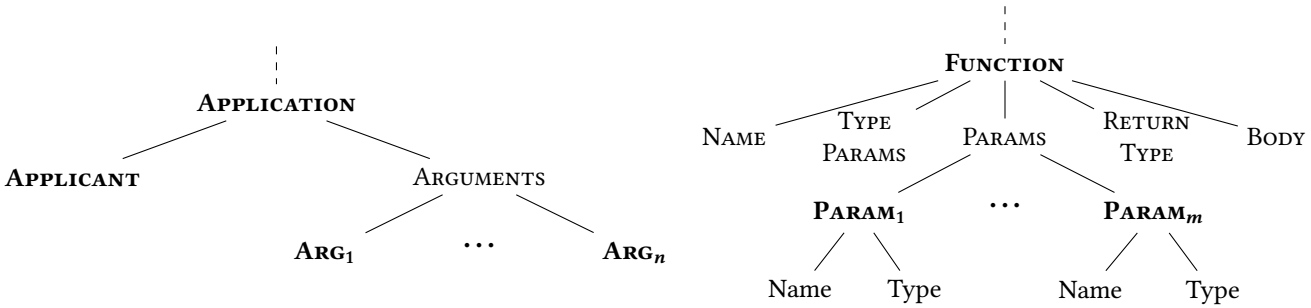


Figure 3. A P-AST for the application and definition sites of an ARITY-MISMATCH error. The relevant terms are in bold.

7.4 Analyzing Highlights as a Clarifier of Prose

In this context, we use a measure of recall to quantify how program highlighting decisions impact the highlighting of the prose. Here, the recall of a potential highlighting strategy is computed over the highlights of the prose, rather than over the highlights of the program source. For instance, Figure 3 depicts the P-AST for programs that induce an arity-mismatch error. (In contrast to the previous examples we have considered, this P-AST consists of two distinct components—a function definition *and* an application site.) A possible report for this error—whose prose includes references to all relevant syntactic elements—is given below. The references to syntactic elements are underlined and the name of their referent is set beneath each underline.

> Evaluating a function application expression failed.
> (APPLICATION)
>
> *n* arguments were passed to the left side.
> (ARG$_1$...ARG$_n$) (APPLICANT)
>
> The left side was a function accepting *m* arguments.
> (FUNCTION) (PARAM$_1$...PARAM$_m$)
>
> A function application expression expects that the left side evaluates to a function which accepts exactly the same number of arguments as are passed to it.
> (APPLICATION) (APPLICANT) (PARAM$_1$...PARAM$_m$) (ARG$_1$...ARG$_n$)

For this error, technical limitations rule out highlighting strategies that would initially highlight *every* textual reference in that message; e.g. an initial highlight of "function application expression" would rule out initially highlighting "left side", "*n* arguments" and "passed" since APPLICANT and ARG$_1$...ARG$_n$ are in the subtree of APPLICATION. While the mechanism of on-demand highlights enables us to augment *all* references to syntactic elements in the prose—though perhaps not immediately or simultaneously—it raises a design decision: which textual references in the report prose should be initially connected to their referent via a highlight, and which connections should be shown on-demand?

We evaluated these decisions using a measure of recall we dubbed *immediate actionability*: the fraction of textual references to distinct syntactic fragments that are *initially* highlighted. In essence, this is the fraction of references in the prose that a user can connect at a glance to the piece of syntax they reference—*without* activating any on-demand highlights. In this example, the prose contains five distinct references to syntactic fragments:

1. APPLICATION
2. ARG$_1$...ARG$_n$
3. APPLICANT
4. FUNCTION
5. PARAM$_1$...PARAM$_m$

The denominator of immediate actionability scores for this error report will be five. Figure 4 depicts two potential highlighting strategies for this arity mismatch error and compares their immediate actionability scores.

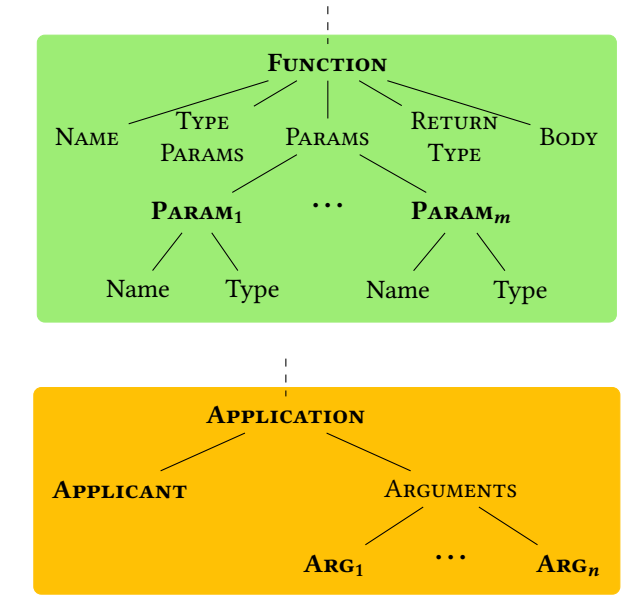
8 Challenges

We now discuss several challenges we encountered in applying our framework to Pyret's error reports.

8.1 The Rainbow Objection

The assignment of colors to highlights poses practical challenges. If an error report contains many references to distinct syntactic fragments, it may not be possible to choose colors with sufficient contrast. If multiple error reports are displayed, the user may conflate the highlights of one error report with those of another.

In practice, we encountered no error report requiring more than three distinct colors: our highlighting strategy for arity-mismatch errors (see Figure 4b). Note that although the number of highlighted syntactic elements for this error grows linearly with respect to the number of formal and actual parameters present in the program, the number of references to these elements in the prose of the report remains constant. To prevent confusion between different error reports, our system only displayed the highlights for one error report at a time. Clicking on an non-highlighted error report hides the existing highlights and displays the highlights for that report.



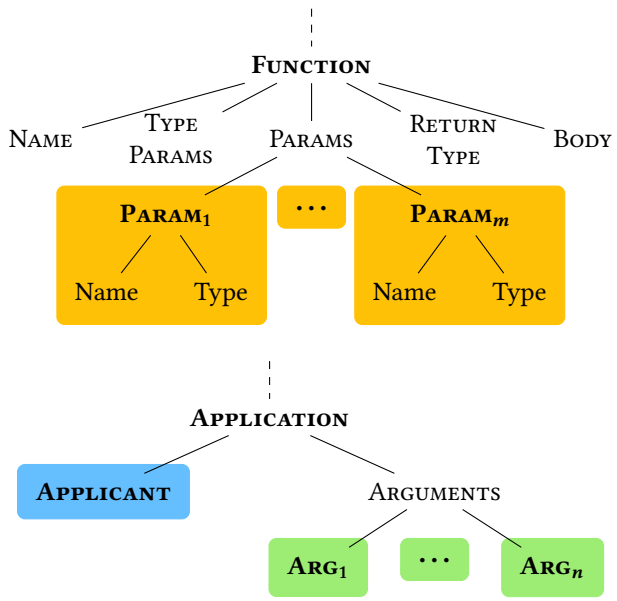
Evaluating a function application expression failed.

3 arguments were passed to the left side.

The left side was a function accepting two arguments.

A function application expression expects that the left side evaluates to a function which accepts exactly the same number of arguments as are passed to it.

(a) In this highlighting strategy, only Applicant and Application are highlighted initially. This highlighting strategy precludes initially highlighting any of the other three syntactic elements referenced by this prose. Consequently, the immediate actionability of this highlighting strategy is $\frac{2}{5}$.



Evaluating a function application expression failed.

3 arguments were passed to the left side.

The left side was a function accepting two arguments.

A function application expression expects that the left side evaluates to a function which accepts exactly the same number of arguments as are passed to it.

(b) In this highlighting strategy, all textual references *except* Applicant and Application are highlighted initially. This highlighting strategy precludes initial highlights of only Applicant and Application. Consequently, the immediate actionability of this highlighting strategy is $\frac{3}{5}$.

Figure 4. Two potential highlighting strategies for an arity-mismatch error. From top to bottom: the highlighting plan depicted on the p-ast of the definition site, the highlighting plan depicted on the p-ast of the application site, the consequences of the program highlighting plan on the highlighting of the error report prose. The caption of each highlighting plan describes the plan and quantifies its impact on the highlighting of the prose with an immediate actionability score. The colors in this diagram reflect the initially highlighted set of phrases and program fragments. The underlined phrases indicate the references to syntactic elements that are highlighted on-demand.

8.2 Relationship Problems

The p-ast for Arity-Mismatch depicted in Figure 3 has two disjoint components: the function application and the function definition. We left the relationship between these components unspecified; the application site could feasibly be either a descendant of the function definition (e.g., a recursive function), or a collateral relative (i.e., a term that is neither a direct ancestor or descendant of the function definition).

If a p-ast involves multiple disjoint components, the relationship between the components should be considered when designing highlights. For instance, the highlighting strategy presented by Figure 4a is actually invalid in our implementation of highlighting: if the Application term is within the subtree of the Body term, then the highlight over the Function term precludes displaying any initial highlights on any of the terms in the Application subtree.

8.3 Selection Influences Classification

We used the classification of terms as relevant or irrelevant to directly inform the design of highlights (i.e., we tried to highlight the relevant terms, and tried to not highlight the irrelevant ones). Our very use of highlights, however, may influence how we classify syntactic structures as relevant and how we weigh the tradeoffs of highlighting strategies.

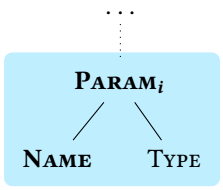
For example, Pyret reports a Shadowed-Param error if the name of a formal parameter of a function conflicts with a name already in scope. In the program fragment below (assuming `a` is bound and visible), the name of the parameter is clearly relevant (since we can resolve the error by changing the name) and is therefore highlighted:

```
1  fun foo( a  :: Number):
2    ...
3  end
```

Given this classification, the highlights depicted above are both sound and complete. However, another possible resolution strategy involves deleting the parameter entirely. Unfortunately, a student who applies an edit-here mentality to the above program may delete the name but not its type annotation—thereby inducing a syntax error. To accommodate this behavior, it is reasonable to conclude that the *entire* parameter, as a unit, is relevant, and consider a strategy that renders a highlight around it:

```
1  fun foo( a :: Number ):
2    ...
3  end
```

Under this revised classification, the initial highlighting scheme we considered remains sound, but is no longer complete. The second highlighting scheme we considered is complete at the expense of soundness (the type annotation of the parameter remains irrelevant to this error):



$$Precision = \frac{2}{3}$$

$$Recall = \frac{2}{2}$$

8.4 Optional Syntactic Structures

In that last example, a loss of precision is effected by the presence of a Type annotation. However, type annotations are optional in Pyret. If we omit optional structures from our p-asts, the estimates of precision we make based on those p-asts will be inflated. Duplicate-Param is an extreme case of this: if omit the Type term from the above p-ast because it is optional, we would have failed to detect any loss of precision whatsoever. To make the most pessimal estimates of precision and recall, optional structures must be included in the p-ast.

8.5 Selection Bias

Although suggesting a rule for classification is outside the scope of this paper, the classifications used in our work roughly follow a simple rule: a term is relevant if there is an edit strategy associated with it that could resolve the error. However, some edit strategies fundamentally cannot be represented on a p-ast or have their associated terms highlighted simply because they involve *adding* terms.

For example, an Unbound-ID error may be resolved by introducing a new binding to the program. The previously-discussed Shadowed-Param error can be resolved by introducing the keyword `shadow` before the name of the parameter. A term which does not exist cannot be highlighted.

8.6 Empty Syntactic Structures

We assume that, when we highlight a structure, we can visibly highlight corresponding code. While this is usually true, it can break down, e.g., when a highlighted structure contains no subterms. Figure 5 depicts an instance of this issue: note that there are no pink arguments highlighted in the code.

These situations can be accommodated with careful observations about the lexical structure of the error-inducing programs. For example, the error report of Figure 5 could be improved by highlighting the `()` tokens if there are no arguments, providing a target for the selection.

A reader might thus wonder why the original report highlighted individual arguments at all (as seen in highlighting strategy of Figure 4b). This is done because the user (particularly a novice) may not fully grasp the language's lexical structure. The split highlighting helps the user see where each argument begins and ends, resolving this and perhaps also, subtly, improving their grasp of the syntax. Another option is thus to use distinct colors for the number and for the word "arguments", using the latter to always highlight the `()`, though this would result in two more colors (one for formal, one for actual).

8.7 Implicit Structures

In addition to zero-width structures, programming languages may contain *implicit* structures. For instance, Pyret requires that method declarations include an explicit "self" parameter as the first formal parameter (à la Python). This argument is *implicitly* passed during method invocation; e.g., if a method is defined with two arguments, the length of the argument list at a call-site should be one. Supplying the wrong number of arguments results in a confusing error report, as Figure 6 shows, because the green highlight refers to three arguments but highlights only two.

9 Preliminary User Study

We applied this process to contribute an overhaul of Pyret's error reports. We rewrote all but a few of Pyret's 87 distinct

```
> fun foo(a, b, c):
    a + b + c
  end

  foo()
```

Evaluating this function application expression errored:

interactions://11:4:0-4:5

```
5 foo()
```

0 arguments were passed to the left side.

The left side was a function accepting 3 arguments:

interactions://11:0:0-1:2

```
1 fun foo(a, b, c):
2   a + b + c
```

(Show program execution trace...)

Figure 5. An error report for Arity-Mismatch with the textual reference to the actual parameters highlighted in pink. The program source does not contain a corresponding highlight because the source contains no parameters on which a highlight can be displayed.

error report types. This framework also identified numerous errors that were insufficiently specific. As a result we refactored the existing errors, resulting in 50 new types of error reports.

While the perspective of error-messages-are-classifiers does not prescribe how error reports should be presented, adhering to this methodology resulted in significant visual changes to Pyret's error reports. A preference towards notifications with high recall (and therefore many information elements) substantially increased the length of most error messages. Our liberal use of highlighting strategies with high immediate-actionability could be visually intense. Respectively, we feared that our revised notifications might be too much to read, and too much to look at (especially with the use of multiple colors to distinguish selectors). We therefore, needed to evaluate the new error messages actual users.

The revised notifications were previewed by teachers during August 2016. Since they were received well, the Pyret team decided to deploy our revised errors to the public that month. In the fall semester that followed, Pyret was used by several hundred students ranging from the high-school to graduate level. During these periods, we did not receive any criticisms relating to the revised notifications, and received modest positive feedback from educators.

Despite this lack of negative response, we wanted to verify directly that student interactions with the revised notifications were positive and effective. At the end of the Fall

```
> data Foo:
    foo
  sharing:
    method bar(self, baz):
      baz
    end
  end

  foo.bar(1, 2)
```

Evaluating this function application expression errored:

interactions://15:8:0-8:13

```
9 foo.bar(1, 2)
```

3 arguments were passed to the left side.

The left side was a function defined accepting 2 arguments:

interactions://15:3:2-5:5

```
4   method bar(self, baz):
5     baz
6   end
```

(Show program execution trace...)

Figure 6. An implicit structure causes an apparently contradictory message: the message references "3 arguments", but the corresponding highlight selects only two arguments. A 'self' parameter has been passed implicitly to the method in the application. The presence of an implicit structure creates an apparent discrepancy between the message text and its corresponding highlight.

2016 semester, we conducted a preliminary study of student sentiment and their interaction with the revised reports. Forty-eight students in an accelerated college-level introductory programming course participated in an optional lab section for extra credit, during which we screen-captured their progress on two programming problems, then administered a survey to solicit feedback regarding error reports.

9.1 Programming Sessions

In the programming portion of this study, we asked students to implement two simple functions. These problems were selected on the basis that they could be both succinctly described and implemented, and because we suspected that the implementation process for these problems would be error-prone. In this order, students were asked to:

1. "Design a function called rainfall that consumes a list of numbers representing daily rainfall amounts as entered by a user. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to

[DEL]	Deletes the problematic code wholesale.
[UNR]	Unrelated to the error report, and does not help.
[DIFF]	Unrelated to the error report, but it correctly addresses a different error or makes progress in some other way.
[PART]	Evidence that the student has understood the error report (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 7. Rubric for edits made in response to errors.

Phase	% of Errors	B
Run-time	50.5	12.3
Well-Formedness	21.9	11.4
Parse	26.2	5.4

Figure 8. Distribution of errors by phase.

the first -999 (if it shows up). There may be negative numbers other than -999 in the list." [9, 24]

2. "Design a function called `argmin` that consumes a list of numbers and produces the index of the smallest number in the list."

We additionally requested that students self-verify the correctness of their implementations by writing tests. Students were permitted to use standard library routines in their implementations, but were asked to not look up solutions online. Participants received the same amount of extra-credit regardless of the completeness or correctness of their implementation. Technical difficulties resulted in the loss of 4 of the 96 recordings. Twelve recordings did not include any encounters with errors.

Across the 80 recordings in which errors were encountered, 283 compilation or execution attempts resulted in an error. We analyzed each student response to these using the coding rubric of Marceau et al. [17] (shown in Figure 7). Three coders applied the rubric independently, then compared their results. A consensus was reached in all instances of disagreement.

We adopted Marceau's measure for the percent of errors that were responded to badly according to this rubric:

$$B = \frac{[\text{UNR}] + [\text{PART}]}{[\text{FIX}] + [\text{UNR}] + [\text{PART}]}$$

This metric does not include [DEL] or [DIFF]; neither coding provide a clear indication as to whether or not the user was responding to the proximate error. We calculated B across all error encounters, for each compilation or execution phase errors were induced in, and for each particular type of error. We discuss these results below.

Error Type	Phase	% Errors	% Bad
Test-Mismatch	R	20.8	18.5
Unbound-Id	R	10.2	14.3
Arity-Mismatch	R	8.8	0.0
Annotation	R	6.7	5.3
Field-Not-Found	R	6.7	27.8
Div-By-Zero	R	4.6	8.3
Shadowed-Id	W	4.2	0.0
Missing-Colon	P	3.9	9.1
Missing-Comma	P	3.9	0.0
Empty-Block	W	3.6	25.0

Figure 9. Summary of coding results for the ten most-encountered errors. The letters *R*, *W*, and *P*, respectively denote the *run-time*, *well-formedness*, and *parsing* phases of execution and compilation.

```
> fun identity(value):
    value
  where:
    # A commented-out test makes this
    # `where` block empty:
    # identity(1) is 1
  end
```

This block is empty:

interactions://13:1:2-1:7

```
2   value
```

A block should end with an expression.

Figure 10. A bug causes Pyret to erroneously highlight a structure other than the one referred to by the message for this type of error.

Our technique of highlighting program structures could not be applied to parse errors, since no structured representation of the program could be constructed. However, the grammar of languages that Pyret is able to parse is more liberal than the grammar of executable Pyret programs. Many syntactic properties are verified during a well-formedness-checking phase that operates over an AST. Errors induced during both this phase of compilation and during program execution were able to highlight program structures.

In all, 10.5% error reports were responded to poorly according to our rubric. Errors induced during phases in which structural highlighting could be applied accounted for approximately 72% of error encounters, of which 12.5% received bad responses. Figure 8 summarizes the coding results by compilation and execution phase.

In the process of reviewing student responses, we discovered two scenarios in which Pyret's error reports performed poorly. The errors induced by these scenarios were among

the then most commonly encountered errors, the coding results of which are summarized in Figure 9. First, we discovered that many students encountered some degree of difficulties with Pyret's module system. These mistakes (caused, for example, by typing `list` instead of `lists`) often surfaced as seemingly-orthogonal Unbound-ID ($B = 14.3\%$) and Field-Not-Found ($B = 27.8\%$) errors. Second, we discovered that the Empty-Block error ($B = 25.0\%$)—induced by programs in which a structure that expects to contain one or more expressions contains no expressions—produced a report that highlighted whatever structure immediately *preceded* the actual problematic structure. Figure 10 depicts an instance of this bug.

9.2 Survey Responses

Following the programming portion of the study, all participants answered the following questions in writing:

1. Do you usually read error messages? Why or why not?
2. What about the error messages do you find helpful?
3. What about the error messages do you find unhelpful or frustrating?
4. When a message refers to your code, are you usually able to find what it is referring to?
5. Do you find the highlights in error messages helpful?
6. Do you have any further comments?

In their answers, participants drew on most of a semester's experience with the Pyret ecosystem. Across all questions, 87.5% of respondents (all but three) commented positively regarding highlighting. Of these, 72.9% of respondents commented *only* in positive terms, while 14.6% included at least one critical or constructive comment about highlighting. Our concerns of verbosity and visual intensity were not substantiated by survey feedback. The use of vivid colors was positively cited by two participants:

1. "I like how it highlights it in a particular color which makes it rather easy to locate."
2. "I love how it highlights the various parts of the application expression in different colors. pleasing to look at and helpful for understanding what went wrong."

All participants additionally indicated that they read error messages, and two participants directly referenced the 'verbosity' of the messages as a helpful aspect:

1. survey: *What about the error messages do you find helpful?*
   student: They are generally verbose and allow my to understand the problem.
2. survey: *Do you usually read error messages? Why or why not?*
   student: Yes, every time! Error messages are an integral part of debugging and, in my case, understanding the development of my program. Error messages don't simply report a failure of a project. In my style of coding, I often use intentional errors as stop points, concept checks, or as a means of redirecting my efforts when I get lost. Therefore, the verbosity of error messages is essential for my process.

9.3 Criticisms

The survey responses of the three students that provided *only* critical or constructive feedback are representative of the criticisms we received. We are also optimistic that each of these frustrations can be entirely addressed through bug fixes and minor interface adjustments:

Incorrect Highlights *Student 1* indicated that highlights were cause for outright frustration, but we believe their frustration was tied to the Empty-Block highlighting bug:

survey: *Do you find the highlights in error messages helpful?*
student: No I find them extremely annoying.
survey: *What about the error messages do you find unhelpful or frustrating?*
student: I frequently comment out my test cases in where blocks and forget to write `nothing`. The error message when you do this is extremely confusing and has given me several headaches because for some reason I never remember that this always happens.

Failure-to-Highlight *Student 2* expressed ambivalence towards highlighting, and cited a scenario in which the localizing benefits of highlights were nullified because highlights could not be rendered for references to imported code:

survey: *Do you find the highlights in error messages helpful?*
student: not really
survey: *What about the error messages do you find unhelpful or frustrating?*
student: If you have multiple files, you have to switch between the testing file and the program file

Several other respondents also referenced this as a frustration. This temporary shortcoming of our implementation would have been particularly visible to this group of students, as the homework assignments asked them to maintain separate files for their implementation and tests.

UI of Highlight References *Student 3* suggested that we remove the "pulsating" visual effect we had applied to on-demand highlights: when an on-demand-highlighted textual reference was moused-over, its corresponding highlight not only appeared, but also blinked.

9.4 Limitations

Though we were unable to substantiate our concerns or identify other systemic issues in Pyret's revised error reports, this study certainly has limitations:

- The college-age participants of this study may be less vulnerable to verbose or overly colorful notifications than younger students.
- The participants of this study were capable users of Pyret, having used it extensively for a semester. The frustrations expressed by students at the end of a semester are likely different than the frustrations that would be expressed at other points in the semester.
- It is also possible that the issues which users reported (many of which were unrelated to highlighting, or even error messages) caused other problems to remain unreported.
- Testing students in a lab imposes constraints (time, motivation, interest, etc.) that may be artificial relative to other environments.

Nevertheless, this study suggests that it is reasonable to consider verbose and colorful messages, subject to further investigation.

10 Related Work

In taking the position that error reports are classifiers of program information, we gained insight on how reports can be assessed, but not on how the designer should decide which information is relevant, nor on what specific mechanism should be used to present that information, nor whether any of these questions are essential to the experience of programming. These questions are open, and some are even controversial. Especially for novices, although it is widely accepted that encounters with errors are formative [7, 15, 18, 19], there is substantial disagreement about the role they should play in learning to program.

Suggesting Fixes There is no consensus on whether error messages should suggest resolution strategies, with literature arguing both for [10, 11, 26] and against [18, 22]. For instance, the design of GREATERP [22], a LISP tutor, was guided by a belief in the pedagogical importance of having students independently formulate error resolutions:

> The tutor is designed to provide only as much guidance as necessary while encouraging the student to generate as much of the solution as possible. Thus, the tutor generally tries to provide hints rather than actual solutions.

The designers of BlueFix [26] took the opposite position, also citing pedagogical implications:

> BlueFix places an emphasis on teaching programming students how to resolve errors by example, and therefore suggests methods to resolve syntax errors using a database of crowd-sourced error fixes.

Once a position on such issues of relevance is taken, our classifier model can be applied to guide the design and assessment of error reports. However, the utility of applying this model at authorship-time is a function of the degree to which the feedback provided by error reports is *predictable*. Environments where correction suggestions are generated via static analysis (e.g., [2]) are highly predictable, and therefore amenable to authorship-time assessment. If suggestions are provided dynamically (e.g., via crowd-sourcing [11, 26]), an authorship-time assessment may not be as informative. The typical real-world performance of these dynamic systems can instead be evaluated by applying precision and recall to the actual suggestions provided for concrete, buggy programs [26].

Presentation Perspectives A classifier model of error reports suggests only how error reports should be assessed, not how the relevant information ought to be presented. For this, authors may draw from both general interface design guidelines (e.g., cognitive dimensions of notation) and recommendations tailored to error reports (e.g., [1, 3, 13, 23, 25]). Barik et al. [3]'s "interaction-first" approach to structuring error reports has designers taxonomize both errors and their associated resolution tasks according to similarities. For instance, the errors induced by a duplicate `cases` patterns in Haskell and duplicate method names in Java might both be taxonomized as a 'Clash', and the applicable editing strategies for these errors may be drawn from a resolution taxonomy (e.g., Remove(X), Replace(X,Y)). These taxonomies aid with the design of highly consistent interfaces for error reporting and interactive resolution.

Authors may implement their design principles with a growing vocabulary of visual notations. The highlighting system used in this work is a variation of one proposed by Marceau et al. [18] and subsequently employed by WeScheme [27]. Our use of highlighting differs slightly from these systems in not supporting nested highlights; we used on-demand highlights instead. Marceau's highlighting system innovates on the previous uses of highlighting for program selection (e.g., [8]) with the key insight that it can *additionally* be used to select error message prose, and that these two uses of highlights can be associated with a common color.

Selection notations can be enriched with additional semantic cues particular to the context in which they are used. MrSpidey, a static debugger for Scheme, uses green and red highlights to annotate operations whose safety can or cannot, respectively, be statically proved [8]. Our use of highlighting did not use colors in this manner. In fact, we selected the colors of highlights *randomly* to dissuade users from attributing meaning to highlights on the basis of their color. In contrast, recent work by Barik [1] presents a semantics-laden program annotation system that performs the dual role of program selector and error explanation. Using such semantically-rich notations may reduce the subjectivity of determining whether a presentation soundly and completely conveys relevant information.

Eliminate Error Reports? If encounters with error reports are daunting to novices, we could try to limit encounters altogether by designing languages and environments which are error-preventing and error-tolerant [14]. Environments such as Scratch [16] and Tern [12] provide programming interfaces that preclude the construction of syntactically invalid programs. In an attempt to further reduce error reports, Scratch also extends this principle to the run-time semantics, and "attempts to do something sensible even when presented with out-of-range inputs" [16].

11 Conclusion

This paper proposes that we shift our perspective on error reports. We complement the broad literature on error message text with a new perspective: that error reports are program classifiers, and as such should be evaluated using the metrics applied to classification. Adopting this perspective both gives us a diagnostic process that we can apply to error reports even before a language has been implemented, and gives us a numeric basis we can include when we compare different error reporting strategies. (We note that our perspective is not limited to *error* reporting; the same principle applies to any form of feedback about program source.)

We develop this perspective through a simple example. We then show its application to the error feedback in the Pyret programming language. Doing this resulted in a substantial revision to Pyret's reporting, but with the potential for negative consequences. We report on a user study that shows that the messages were effective; we observed positive effects on the student users while finding almost no negative impacts.

Acknowledgments

This work is partially supported by the US National Science Foundation. We thank the anonymous reviewers for their comments, which helped improve this paper. We are grateful to our shepherd, Karim Ali, for responding to our queries and helping us work through the reviews in the process of revising the paper. We are especially grateful to Ben Lerner and Joe Politz of the Pyret team for their help with implementing the new error message presentation, to Peter Hahn and Raghu Nimmagadda for their help reviewing screen captures, and to the students who participated in the studies and gave us valuable feedback.

References

[1] Titus Barik. 2014. Improving error notification comprehension through visual overlays in IDEs. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 177–178. DOI: http://dx.doi.org/10.1109/VLHCC.2014.6883043

[2] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. 2016. From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. DOI: http://dx.doi.org/10.1109/icsme.2016.63

[3] Titus Barik, Jim Witschey, Brittany Johnson, and Emerson Murphy-Hill. 2014. Compiler Error Notifications Revisited: An Interaction-first Approach for Helping Developers More Effectively Comprehend and Resolve Error Notifications. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 536–539. DOI: http://dx.doi.org/10.1145/2591062.2591124

[4] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and Richard M. Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Proceedings of the 4th International Conference on Cognitive Technology: Instruments of Mind (CT '01)*. Springer-Verlag, London, UK, UK, 325–341. http://dl.acm.org/citation.cfm?id=647492.727492

[5] John Brooke. 1996. SUS-A quick and dirty usability scale. In *Usability Evaluation in Industry*, Patrick W. Jordan, Thomas, Ian Lyall McClelland, and Bernard Weerdmeester (Eds.). CRC Press, Chapter 21, 189–194.

[6] Pyret Developers. 2016. Pyret Programming Language. (2016). http://www.pyret.org/.

[7] Benedict du Boulay and Ian Matthew. 1984. Fatal Error in Pass Zero: How Not to Confuse Novices. In *Proc. Of the 2nd European Conference on Readings on Cognitive Ergonomics - Mind and Computers*. Springer-Verlag New York, Inc., New York, NY, USA, 132–141. http://dl.acm.org/citation.cfm?id=2815.2825

[8] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1997. *DrScheme: A pedagogic programming environment for scheme*. Springer Berlin Heidelberg, Berlin, Heidelberg, 369–388. DOI: http://dx.doi.org/10.1007/BFb0033856

[9] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14)*. ACM, New York, NY, USA, 35–42. DOI: http://dx.doi.org/10.1145/2632320.2632346

[10] Thomas Flowers, Curtis A. Carver, and James W. Jackson. 2004. Empowering students and building confidence in novice programmers through Gauntlet. In *Frontiers in Education*. T3H/10–T3H/13. DOI: http://dx.doi.org/10.1109/FIE.2004.1408551

[11] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1019–1028. DOI: http://dx.doi.org/10.1145/1753326.1753478

[12] Michael S. Horn and Robert J. K. Jacob. 2007. Tangible Programming in the Classroom with Tern. In *CHI '07 Extended Abstracts on Human Factors in Computing Systems (CHI EA '07)*. ACM, New York, NY, USA, 1965–1970. DOI: http://dx.doi.org/10.1145/1240866.1240933

[13] James J. Horning. 1976. What the Compiler Should Tell the User. In *Compiler Construction, An Advanced Course, 2Nd Ed.* Springer-Verlag, London, UK, UK, 525–548. http://dl.acm.org/citation.cfm?id=647431.723720

[14] Andrew J. Ko. 2009. Attitudes and self-efficacy in young adults' computing autobiographies. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 67–74. DOI: http://dx.doi.org/10.1109/VLHCC.2009.5295297

[15] Michael J. Lee and Andrew J. Ko. 2011. Personifying Programming Tool Feedback Improves Novice Programmers' Learning. In *Proceedings of the Seventh International Workshop on Computing Education Research (ICER '11)*. ACM, New York, NY, USA, 109–116. DOI: http://dx.doi.org/10.1145/2016911.2016934

[16] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *Trans. Comput. Educ.* 10, 4, Article 16 (Nov. 2010), 15 pages. DOI: http://dx.doi.org/10.1145/1868358.1868363

[17] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In *Special Interest Group on Computer Science Education*. ACM, New York, NY, USA, 499–504. DOI: http://dx.doi.org/10.1145/1953163.1953308

[18] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *ACM International Symposium On New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2011)*. ACM, New York, NY, USA, 3–18. DOI: http://dx.doi.org/10.1145/2048237.2048241

[19] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler Error Messages: What Can Help Novices?. In *Proceedings of SIGCSE (SIGCSE '08)*. ACM, New York, NY, USA, 168–172. DOI: http://dx.doi.org/10.1145/1352135.1352192

[20] S. Joon Park, Craig M. MacDonald, and Michael Khoo. 2012. Do You Care if a Computer Says Sorry?: User Experience Design Through Affective Messages. In *Proceedings of the Designing Interactive Systems Conference (DIS '12)*. ACM, New York, NY, USA, 731–740. DOI: http://dx.doi.org/10.1145/2317956.2318067

[21] Peter G. Polson, Clayton Lewis, John Rieman, and Cathleen Wharton. 1992. Cognitive Walkthroughs: A Method for Theory-based Evaluation of User Interfaces. *Int. J. Man-Mach. Stud.* 36, 5 (May 1992), 741–773. DOI: http://dx.doi.org/10.1016/0020-7373(92)90039-N

[22] Brian J. Reiser, John R. Anderson, and Robert G. Farrell. 1985. Dynamic Student Modelling in an Intelligent Tutor for LISP Programming. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1 (IJCAI'85)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 8–14. http://dl.acm.org/citation.cfm?id=1625135.1625137

[23] Ben Shneiderman. 1982. Designing Computer System Messages. *Commun. ACM* 25 (Sept. 1982), 610–611. DOI: http://dx.doi.org/10.1145/358628.358639

[24] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858. DOI: http://dx.doi.org/10.1145/6592.6594

[25] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Adv. in Hum.-Comp. Int.* 2010 (Jan. 2010), 3:1–3:26. DOI: http://dx.doi.org/10.1155/2010/602570

[26] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. 2012. *BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair*. Springer Berlin Heidelberg, Berlin, Heidelberg, 228–239. DOI: http://dx.doi.org/10.1007/978-3-642-33642-3_25

[27] Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. 2011. WeScheme: The Browser is Your Programming Environment. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. ACM, New York, NY, USA, 163–167. DOI: http://dx.doi.org/10.1145/1999747.1999795