Error Messages Are Classifiers

A Process to Design and Evaluate Error Messages

John Wrenn
Brown University

Shriram Krishnamurthi
Brown University

October 27, 2017

Transcript

I'm John Wrenn, a PhD student at Brown University, and I hack on the error messages of Pyret—a programming language designed expressly to be an excellent choice for teaching, that is used by thousands of middle schoolers, high schoolers and college students every year.

I think we've all been at one time or another so frustrated by an error that we just wanted to give up computers. The fact that we're all here today is a testament to how forgiving—or conditioned—we are to the quirks of software. Pyret serves audience that lacks this understanding, and we don't want their first experience with computing to be their last!

Needless to say, we sought out the wisdom of the ancients to design our errors.

> Writing good messages, like writing poems, essays, or advertisements, requires experience, practice, and a sensitivity to how the reader will react.
>
> B. Shneiderman,
> *Designing Computer System Messages* (1982)

Transcript

Computer scientists have been complaining about error messages for as long as there have been error messages, and offered good advice on how to improve them.

Ben Shneiderman, lamenting a lack of care taken in writing error reports, suggested qualitative prescriptions to language developers, likening the process to that of writing poems or essays.

> 1. Clarity & Brevity
> 2. Specificity
> 3. Locality
> 4. Proper phrasing:
>
>   4.1 Positive tone
>   4.2 Constructive guidance
>   4.3 Programmer language
>
> J. Traver,
> *On Compiler Error Messages* (2010)

Transcript

Many computer scientists have specified analytical enumerations on the characters of "good" error reports, from JJ. Horning in 1976 who wrote an entire chapter on writing *readable* messages, to Javier Traver in 2010, who felt that good messages are, among other things, clear, brief, and positive.

Measuring the Effectiveness of Error Messages Designed for Novice Programmers

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

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

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

ABSTRACT
Good error messages are critical for novice programmers. Recognizing this, the DrRacket programming environment provides a series of pedagogically-inspired language subsets with error messages customized to each subset. We apply human-factors research methods to explore the effectiveness of these messages. Unlike existing work in this area, we study messages at a fine-grained level by analyzing the edits students make in response to various classes of errors. We present a rubric (which is not language specific) to evaluate student responses, apply it to a course-worth of student lab work, and describe what we have learned about using the rubric effectively. We also discuss some concrete observations on the effectiveness of these messages.

Categories and Subject Descriptors K.3.2 [Computer and Education]: Computer and Information Science Education—Computer science education; H.5.2 [User Interfaces]: Evaluation/Methodology

General Terms Experimentation, Human Factors

Keywords Error messages, Novice programmers, User-studies

1. INTRODUCTION

In a compiler or programming environment, error messages are one of the most important points of contact between the system and the programmer. This is all the more critical in tools for novice programmers, who lack the experience to decipher complicated or poorly-constructed feedback. Thus, many research efforts have sought to make professional compilers more suitable for teaching by rewriting their error messages [10] or supplementing

make sense at that point, and similarly customizes error messages. The levels and messages have evolved over a decade of observation in lab, class, and office settings.

Despite this care, we still see novices struggle to work effectively with the messages. To understand why, we logged students' edits in response to errors over an entire college-level introductory course and coded whether the edits reflected understanding of the error message. Our work is novel in using fine-grained data about edits to assess the effectiveness of individual classes of error messages. Our coding rubric for assessing the performance of error messages through edits is a key contribution of this work. Our observations about how to use the coding results to reflect on our course is another. Finally, we also present some concrete observations on how students respond to these messages.

2. RESPONSES TO ERROR MESSAGES

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

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

Transcript

There have also been a handful of empirical studies.

Several years ago, a team of researchers at WPI and Brown—including my co-author Shriram—set out to investigate some unexpected difficulties students seemed to be having with Racket—a language designed expressly to excel at teaching novices computer programming.

They recorded how students edited their programs in response to errors and calculated—for each error type—the percent of responses that implied an either incomplete or incorrect understanding the error report.

What they found alarmed them: among the most frequently encountered types of errors, upwards of 50% of responses implied an incomplete or incorrect understanding.

> Writing good messages, like writing poems, essays, or advertisements, requires experience, practice, and a sensitivity to how the reader will react.
>
> B. Shneiderman,
> *Designing Computer System Messages* (1982)

Transcript

Is it that Racket's designers lacked experience, practice, or a sensitivity to how users would react?

A Cautionary Tale

> 1. Clarity & Brevity
> 2. Specificity
> 3. Locality
> 4. Proper phrasing:
>   - 4.1 Positive tone
>   - 4.2 Constructive guidance
>   - 4.3 Programmer language
>
> J. Traver,
> *On Compiler Error Messages* (2010)

Transcript

Of course not! And it's hard to argue that they were insufficiently principled; Racket's messages *are* brief, they *are* consistent, they *are* polite, and usually *do* localize the problem. They were designed with principles!

In actuality, at least two things went wrong.

First, it's one thing to hold the principle that messages should be "readable", but who's trying to write *unreadable* error messages? *All* error messages are readable to the person who authored them.

For instance, the wording of error messages was something that Racket's developers had paid particularly close attention to, but but Marceau et. al observed that novices were struggling with the vocabulary.

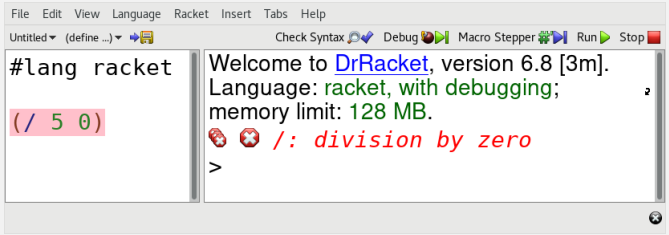
Principles are *not* Heuristics

Transcript

Principles are neither users, methods or measures. There must be a clear translation between a principle to a method or measure.

Moreover, that method or measure needs to have some relationship to user behavior.

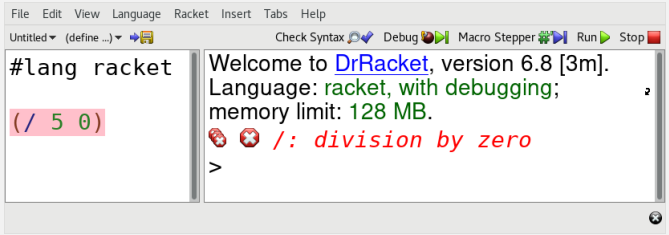
1. This expression raised a runtime exception



Transcript

When an error is reported in Racket, a pink highlight is rendered over some span of code. Like, if an expression throws an exception, you highlight it.

1. This expression raised a runtime exception

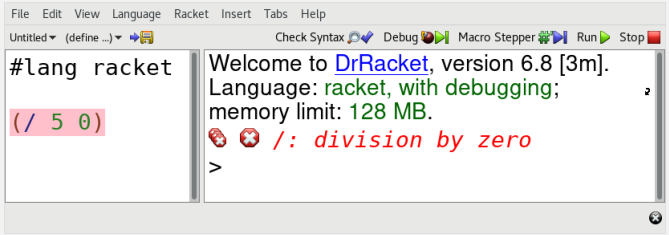


2. The parser did not expect to find this
3. This expression is inconsistent with another part of the code
4. The parser expected to see something after this, but nothing is there
5. This parenthesis is unmatched.

Transcript

Second, these researchers actually identified five distinct intentions with which Racket's developers used highlighting. But when users encountered highlights, they tended applied an "edit-here" interpretation...

1. This expression raised a runtime exception



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

Transcript

...even when the message implied that the highlighted code was the *only* correct part of the program! (As is often the case the parentheses that is unmatched is highlighted.)

Intention $\neq$ Interpretation

Transcript

The intention of the designers, wasn't the interpretation that users adopted.

Our Perspective

Transcript

Principles are essential for guiding the design of anything, but we want our principled, analytic perspective to avoid these pitfalls. It should accomodate the user's interpretation, and it should be clearly translatable to a concrete method and measure.

We have therefore taken a perspective that is directly informed by Guillaume's observation of an "edit-here" interpretation: The act of selecting code implicitly draws the reader's attention to those fragments *and* takes attention away from the fragments *not* highlighted. In this respect, error 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".

Perspective

Program

Transcript

And we apply this lesson more generally, too. Yes, an an error report is a classifier of the program...

Perspective

Program

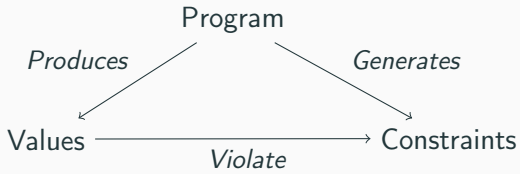
Values

Constraints

Transcript

...but it's also an explanation of a failure that includes things like constraits, the values that violated them...

Perspective



Transcript

...and how all of these things relate to each other, because you need to know that to inform where you make your edit!

We want our error report—given all of the information available when the report is produced—to be *sound* and *complete*.

```
> 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.

Transcript

We want it to be *sound*—i.e., to not include irrelevant information—because if users are presented with irrelevant information—say, a highlight of an irrelevant fragment of the program—because the user might act on that information.

```
> segmentation fault
```

Transcript

And we want it to be *complete*—to include all relevant information—because we can't expect that users will act on information that they don't have.

Method

1. *Identify* available information.

Transcript

This perspective implies a method for developing error messages.

We identify the information we have available...

Method

1. *Identify* available information.
2. *Classify* information as relevant (or not).

Transcript

...judge which elements of it are relevant (or not)...

Method

1. *Identify* available information.
2. *Classify* information as relevant (or not).
3. *Select* information.

Transcript

...and ‘select’ this information by presenting it in an error report.

Measures

$$\text{Soundness} \approx \text{Precision} \qquad = \frac{|\text{relevant} \cap \text{selected}|}{|\text{selected}|}$$

Transcript

Moverover, our principles imply concrete, measures. Taking a page from the the information retrieval book of tricks, we quantify the degree to which an error report is a sound selection via *precision*, the fraction of selected things which are relevant.

Measures

$$\text{Soundness} \approx \text{Precision} = \frac{|\text{relevant} \cap \text{selected}|}{|\text{selected}|}$$

$$\text{Completeness} \approx \text{Recall} = \frac{|\text{relevant} \cap \text{selected}|}{|\text{relevant}|}$$

Transcript

And we quantify the completeness of the message via recall, the fraction of relevant things which are selected.

```
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"*

1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

Because our goal is to improve Pyret's error reporting, we applied this methodology to improve errors like this one.

For the sake of demonstration, let's say our language doesn't permit adding a number and a string. Something like this is pretty typical of what a language implementor might whip up quickly and then move on to more important things.

Program

Values

Constraints

1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

In our framework, the first thing we do to evaluate is actually enumerate all the things we think are relevant.

What do we know, and is it relevant?

Constraints

1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

We know that our language allows using plus for both addition and concatenation, and that these two use cases have different constraints associated with them.

Values

1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

From how our language checks constraints, we know we will have some values at hand when the error is detected.

Program

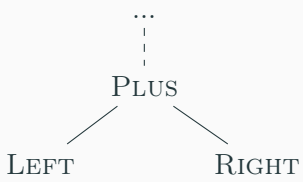
1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

Figuring out what we know about the user's program is a little more subtle.

We aren't designing an error report for a particular, concrete programs; we're designing one for *all* programs that cause this error.

The key observation is that programs which cause the same types of errors usually contain the same sort of syntactic structures...



1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

And we represent what we know about the commonalities of programs with failing binary plus operations by drawing out a *partial* and *parametric* AST. It's partial because we've left large fragments of the program unelaborated, and it's parametric because some details about this diagram—like just how many arguments there will be—are concretized by the instance in which the error is actually reported.

1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

And lastly, we need to consider how all of these types of information relate to each other. Why are those constraints relevant? Because a plus expression appeared in the program.

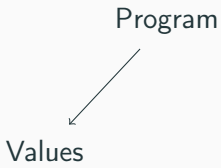
Program

Values ⟶ Constraints

1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

Why are those constraints relevant? Because we have some set of values that violated them.



Constraints

1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

And to fix this error, we might need make a change that altes the values that were ultimately produced. To do this, you need to have an understanding of how those values were produced by terms.

```
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"*

1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

This is a far richer—and longer—set of relevant information than was perhaps immediately obvious.

```
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"*

1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

And we find that it has some striking deficiencies, even if we're generous in what we count as 'included' in the message.

Our error report here doesn't even mention that plus has these left and right operands. It doesn't relate the values reported in the message to those operands. And it doesn't mention constraints at all. This is a great message—if you already know what you did wrong.

$$Recall = \frac{|\text{relevant} \cap \text{selected}|}{|\text{relevant}|}$$
$$\approx 36\%$$

The binary plus expression failed on the values:

- 15
- *"cat"*

1 Plus
2 Left
3 Right
4 Add
5 Concat
6 v_{Left}
7 v_{Right}
8 Plus imposes Add
9 Plus imposes Concat
10 Add constrains v_{Left}
11 Add constrains v_{Right}
12 Concat constrains Left
13 Left produces v_{Left}
14 Right produces v_{Right}

Transcript

And these extreme deficiencies of completeness are clearly reflected if we actually go and calculate out recall.

Building a Better Error

```
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"*

Transcript

But, having enumerated exactly what we hope this error message should convey...

Building a Better Error

```
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 operand was:
15

The value of the right operand was:
"cat"

A binary plus expression expects that either:

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

Transcript

...we could use our enumeration of relevant elements to construct a new, uber-complete message. We relate those values to the operands they came from, and we introduce a whole new dicussion of the constraints that need to be satisfy.

But—at least for Pyret—that's *still* not enough.

This message assuming that between the gutter marker...

Building a Better Error

```
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 operand was:
15

The value of the right operand was:
"cat"

A binary plus expression expects that either:

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

Transcript

...and these three textual references to syntactic locations, you'll know where to look.

But what if you do not know what an "operand" is? Phrases like "left operand" and "right operand" are going to be more frustrating than helpful.

And even if you do suss that out, this message is referencing three distinct parts of the program, but we only have one gutter marker. And *even* if you do infer that the gutter marker localizing the "binary plus expression" that failed, 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.

But there's a classic solution to this problem: we can follow each of these syntactic references with a source location.

Building a Better Error

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

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

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

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

A binary plus expression expects that either:

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

Transcript

...for messages that contain many references, this can dramatically increase size of the explanatory text, not to mention it forces you to constantly context switch between english and location notation. This is not a message you can naturally read aloud.

In prioritizing complete information selections, we've ended up with *many* references to syntax (even for 'simple' reports like this one), and our trusty presentation mechanisms simply were not cut out for it.

Building a Better Error

```
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 operand was:
15

The value of the right operand was:
"cat"

A binary plus expression expects that either:

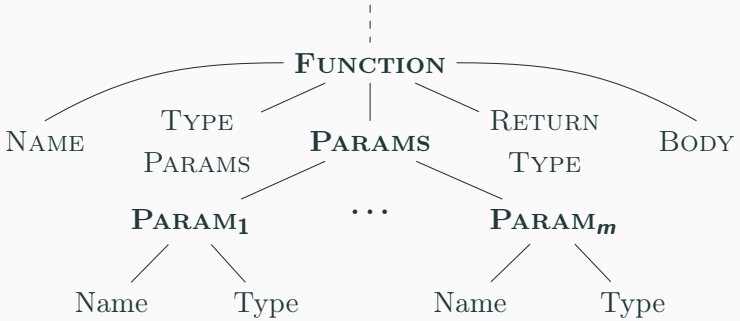
- that the left operand is a string, or
- that both the left operand and right operand are numbers.

Transcript

Again, we drew inspiration from the Racket researchers. Trying to address the issue of students not knowing the vocabulary of error messages, they suggested that program highlighting can be applied to *both* the source code *and* the prose of the message. Whenever the prose references something in the program source, you highlight both that term and the prose in the same color.

With a variation of this highlighting scheme, we were able to remove nearly all source locations from the error reports and craft these detailed but naturally readable diagnostics.

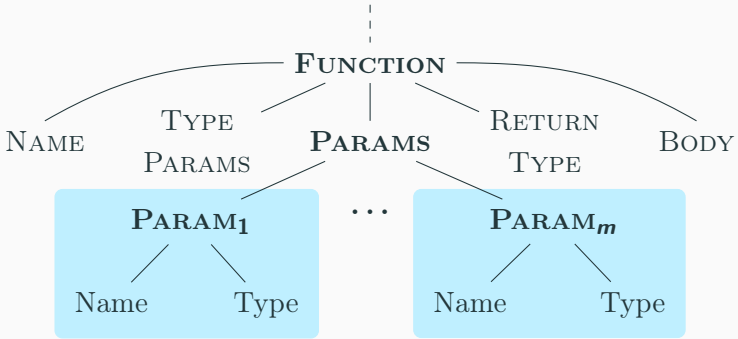
Moreover, we were able to reuse the same machinery we employed for figuring out the information content of error reports to plan out highlighting decisions.



Transcript

because in following this process, we had *already* constructed a representation of everything we know about the syntactic structure of the programs: the partial and parametric AST diagram!

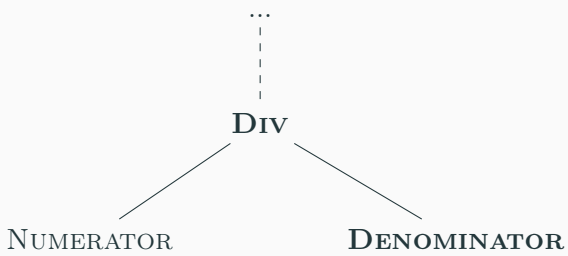
The edges of an AST naturally represent both a structural and syntactic contains relationship of nodes to their children.



Transcript

We can exploit this similaritiy by representing the visual scope of highlights as a box around the subtree of the terms we want to select.

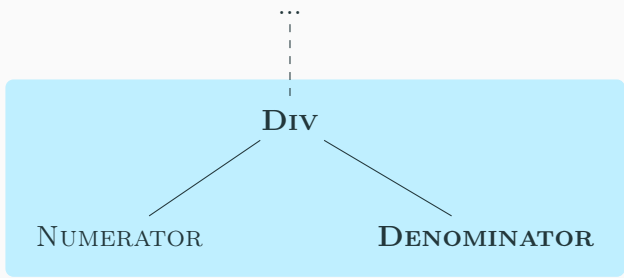
We say a node is 'selected' if it is within the bounds of a highlight.



Transcript

Having a notion of selection, we analyze the soundness and completeness of these highlights over the code with precision and recall. For a division-by- zero error, the whole expression is relevant and the denominator is definitely relevant, but there's no edit you can make to the numerator that will fix this bug.

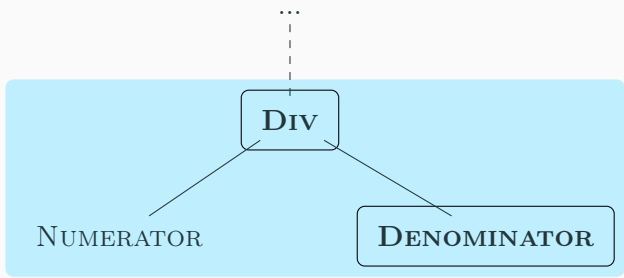
If we highlight the entire division expression...



$\mathit{Recall} = 1$

Transcript

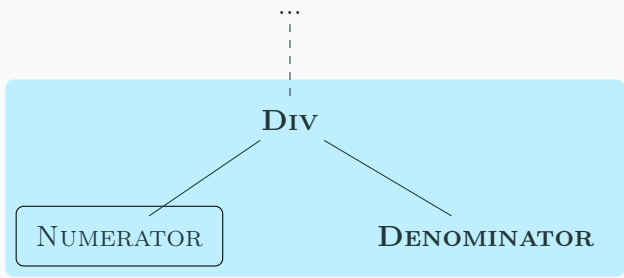
...we achieve perfect recall...



$Recall = 1$

Transcript

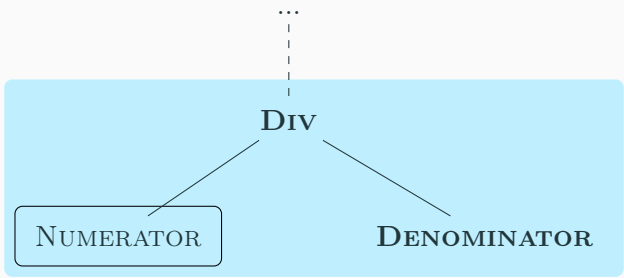
...because both relevant terms are selected.



$Recall = 1$

Transcript

...but an irrelevant term is also included in the selection,

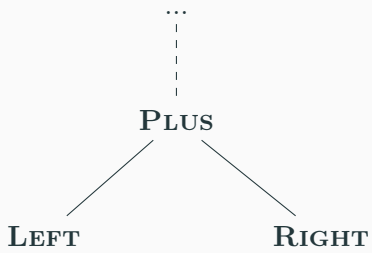


$$Precision = \frac{2}{3}$$

Transcript

...and this is reflected as a loss of precision.

```
1  c = a + b
```

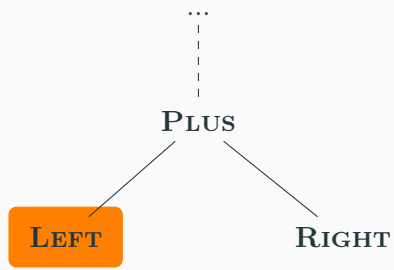


Transcript

There's one thing that looks good on paper—and when drawn over a PAST—that doesn't work in practice: nested highlights.

Say we've decided the left operand is relevant.

```
1  c = a + b
```

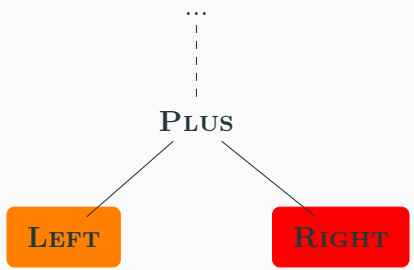


Transcript

so let's highlight that.

The right operand is relevant...

```
1  c = a + b
```

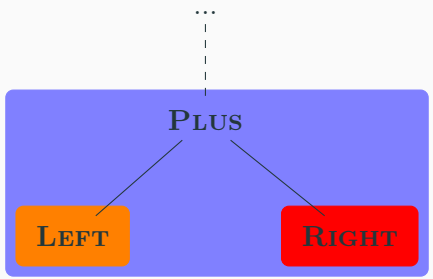


Transcript

so we'll highlight that, too.

And the fact that this error happens in the context of a Plus expression is relevant

```
1  c = a + b
```



Transcript

so let's highlight that in blue.

This highlighting strategy seems great over an AST because the nested structure of these highlights is clearly conveyed.

But in most text editors, the spacing between lines and characters is something sacred, and you end up with something that looks like the three *adjacent* colors, rather than a nested structure of expression and operands.

So we settled upon a rule: no nested highlights.

On-Demand Highlights

```
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 operand was:
15

The value of the right operand was:
"cat"

A binary plus expression expects that either:

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

Transcript

At first glance, this restrictions seems like we need to make choices between highlighting strategies with major tradeoffs of precision and recall.

We could highlight each of the operands of this plus,

On-Demand Highlights

```
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 operand was:
15

The value of the right operand was:
"cat"

A binary plus expression expects that either:

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

Transcript

...or we could highlight the whole expression.

On-Demand Highlights

```
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 operand was:
15

The value of the right operand was:
"cat"

A binary plus expression expects that either:

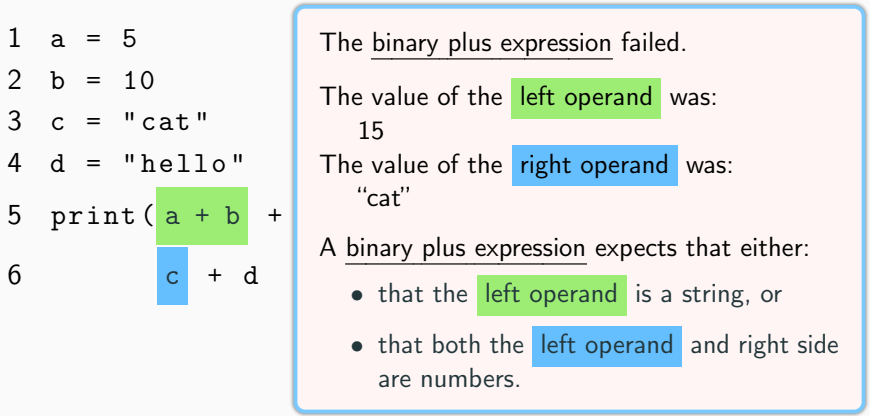
- that the left operand is a string, or
- that both the left operand and right side are numbers.

Transcript

We solve this problem by settling on some particular set of highlights that are shown immediately

And make whatever is left over an *on-demand* highlight...

On-Demand Highlights

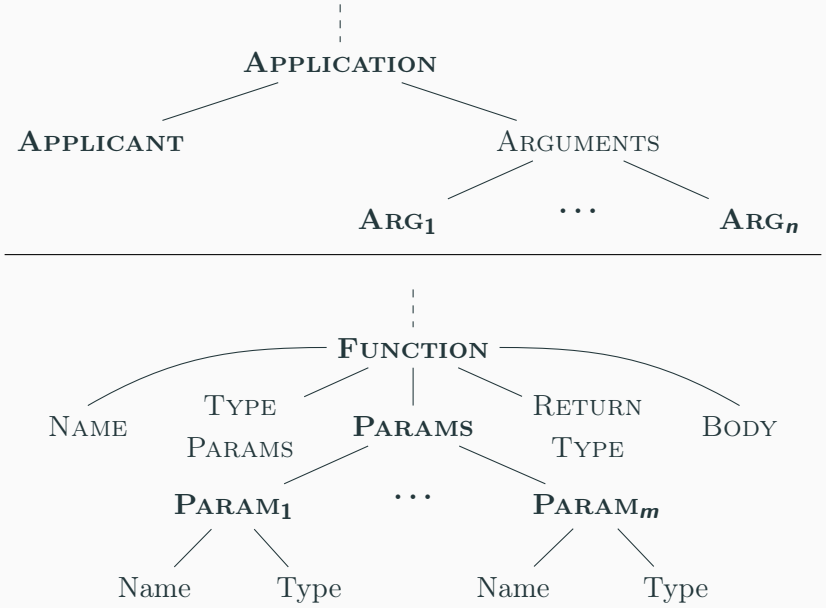


Transcript

...a hyper-link that, when hover overed, temporarily hides the other highlights and shows its own highlight.

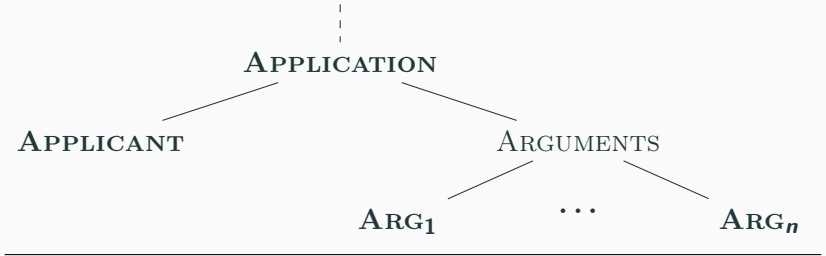
By interacting with the error message, the user can now disambiguate *all* of the source code references, so maybe it doesn't make sense to calculate recall of the highlights over the program source, because we're not excluding this information from the error report—they just have to ask.

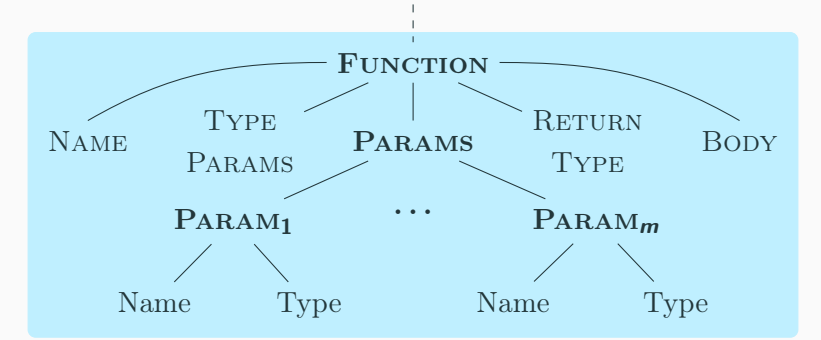
So we instead guide our highlighting decisions by calculating recall over the highlights applied to the *prose* of the error report. By making "binary plus expression" an on-demand highlight, two-thirds of the references in this prose are immediately connected to the source code, as opposed to only one-third if made the operands on-demand highlights.



Transcript

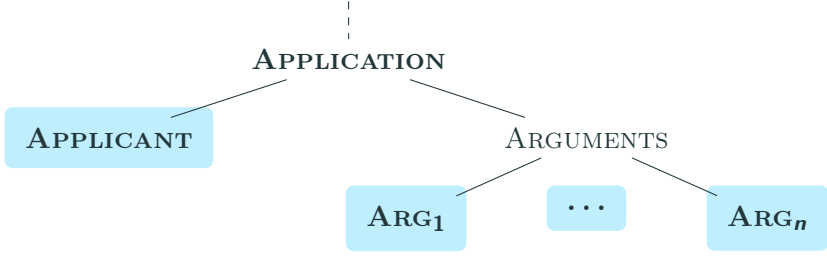
There are still ways in which we can make faulty assumptions about error-inducing programs. If an error may involve more than one interacting syntactic component, we may need to consider the structural relationship between the two.





Transcript

For instance: in an arity mismatch error, if we highlighted the entire function definition, that would rule out *any* highlights of the application sight, since the application could occur inside the function definition, and that would be a nested highlight.



Transcript

And we are also still not completely immune from faulty lexical assumptions. An AST is a good structure on which to plan highlights, but doing so comes with the assumption that those nodes in the AST actually take up space.

We might decide, for instance, to highlight each argument of an arity mismatch error individually...

```
> 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
```

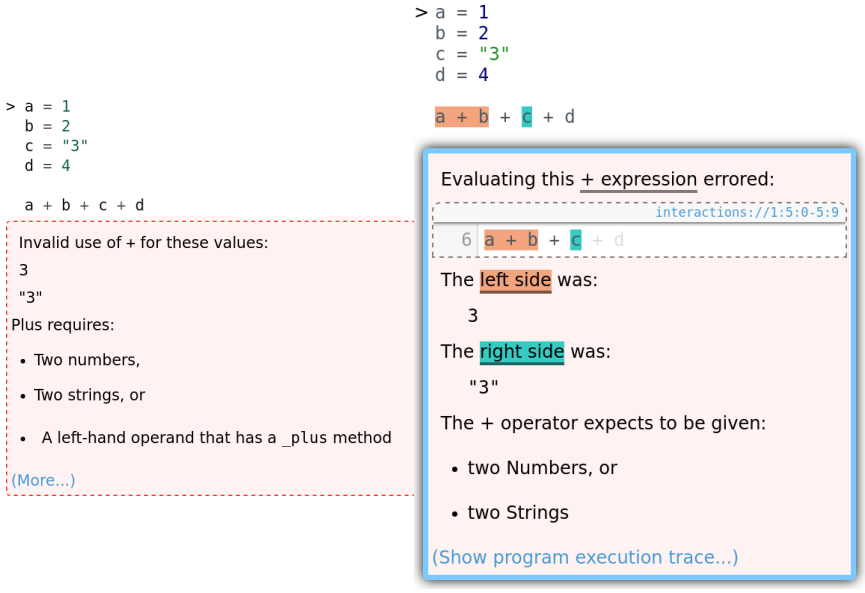
(Show program execution trace...)

Transcript

But if we fail to anticipate the possibility that there could be no arguments, you end up with a highlighted reference in the prose that has no corresponding highlight in the program source.

Transcript

I applied this pipeline to contribute an overhaul of Pyret's error reports. I rewrote all but a few of Pyret's 87 distinct error report types. Some error types were insufficiently specific and broken into multiple errors; 50 new error types now exist.



Transcript

This is a typical transformation.

The revised messages favored an explicit style over an implicit one. For example, it's more explicit which values correspond to which operands. There was some fear that the messages were now too wordy.

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

  foo(1, 2)
```

Evaluating this function application expression errored:

interactions://9:4:0-4:9

```
5 foo(1, 2)
```

2 arguments were passed to the left side.

The left side was a function accepting 3 arguments:

interactions://9:0:0-1:2

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

(Show program execution trace...)

Transcript

An emphasis on high recall and highlighting strategies that had many distinct highlights could be visually intense. The highlighting strategy I adopted for arity mismatch errors entailed four distinct colors.

Respectively, we feared that our revised notifications might be too much to read, and too much to look at.

The revised notifications were previewed by teachers during August 2016. They were received well and deployed to the public that month.

Pyret was used by several hundred students in the fall semester that followed. We did not receive any criticisms relating to the revised notifications, and received modest positive feedback from educators.

User Study

1. Programming
2. Survey

Transcript

Of course, principles are not users. If I wanted to gain any confidence that student interactions with the revised notifications were positive and effective, I needed to actually observe people using them.

So at the end of last semester, I conducted a preliminary study of student sentiment and their interaction with the revised reports.

Forty-eight CS19 students participated in an optional lab section for extra credit, during which I screen-captured their progress on two programming problems, then administered a survey to solicit feedback regarding error reports.

Edit Coding

> 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 the first -999 (if it shows up). There may be negative numbers other than -999 in the list. [Fis14, Sol86]

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

Transcript

During the programming portion, 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

Edit Coding

[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).

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

[MFK11]

Transcript

We reviewed these recordings, and coded each edit response according to the same rubric applied by the researchers at WPI and Brown.

Likewise, we calculated the faction of edit responses that implied an imcomplete or incorrect understanding of the message.

Edit Coding

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

Transcript

In all, 10.5% error reports were responded to poorly according to our rubric. However, the scope of my work was mostly limited to runtime and well-formedness errors–phases in which the program is parsable. In these phases 12.5% received bad responses.

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

The letters *R*, *W*, and *P*, respectively denote the *run-time*, *well-formedness*, and *parsing* phases of execution and compilation.

Transcript

We also calculated *B* for each type of error. Although of the of the ten most commonly encountered types of errors, none were responded to poorly more than 28% of the time, there's a small number of errors that seem to be responded to disproportionately poorly.

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

The letters *R*, *W*, and *P*, respectively denote the *run-time*, *well-formedness*, and *parsing* phases of execution and compilation.

Transcript

In reviewing the videos, we discovered that students were encountering difficulties with Pyret's module system, and that these mistakes surfaced as seemingly orthogonal Unbound-ID and Field-Not-Found errors.

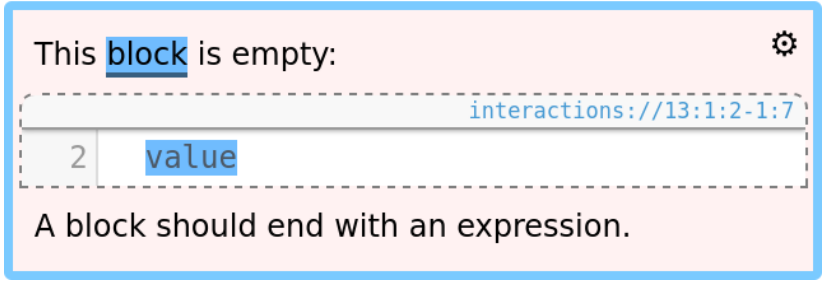
Error Type	Phase	% Errors	% Bad
Test-Mismatch	R	20.8	18.5
Unbound-Id	R	10.2	14.3
Arity-Mismatch	R	8.8	0.0
Annotation	R	6.7	5.3
Field-Not-Found	R	6.7	27.8
Div-By-Zero	R	4.6	8.3
Shadowed-Id	W	4.2	0.0
Missing-Colon	P	3.9	9.1
Missing-Comma	P	3.9	0.0
Empty-Block	W	3.6	25.0

The letters *R*, *W*, and *P*, respectively denote the *run-time*, *well-formedness*, and *parsing* phases of execution and compilation.

Transcript

Second, we discovered that the Empty-Block error—induced by programs in which a structure that expects to contain one or more expressions contains no expressions...

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



Transcript

...produced a report that highlighted whatever structure immediately *preceded* the actual problematic structure.

While this was fruitful for identifying some painpoints with Pyret, it did not reveal any systemic issues with students failing to interpret messages.

Survey

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?

Transcript

Following the programming portion of the study, all participants completed a questionairre about hteir interactions with error messages. There responses drew on a semester's worth of experiences with Pyret.

Across all questions, 88% of respondents (all but three) commented positively regarding highlighting.

> I like how it highlights it in a particular color which makes it rather easy to locate.

Transcript

Students enthusiastically noted the use of colors—probably the most obvious thing distinguishing Pyret's messages from other languages.

> They are generally verbose and allow my to understand the problem.

Transcript

And two students specifically noted the verbosity of the error messages as a helpful feature.

> Sometimes clicking on the message (which is supposed to move the code window to the appropriate line) **doesn't actually move it to the right place**, though.

Transcript

However, students *did* notice, and even found it annoying when highlights failed to work as expected.

A Very Critical Response

> No I find them **extremely annoying**. [...]
> 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.

Transcript

Only one student expressed *only* frustration regarding highlighting, and it appears that this frustration relates to the empty-where-blcok error report, where the entirely wrong thing is highlighted.

The takeaway from these responses is that highlights play an unusually central role. They are attention-grabbing. They are the chief localizers. Without them, the textual references of the message would be completely ambiguous. So when highlights fail, in any way, it is *really* noticed.

Error Messages Are Classifiers

A Process to Design and Evaluate Error Messages

Jack Wrenn
me@jswrenn.com

Shriram Krishnamurthi
sk@cs.brown.edu

Transcript

This work is ongoing, these results are encouraging.

Adopting this information-centric perspective of error messages has armed the Pyret development team with a concrete method and measure, where previously our process was ‘write well and hope for the best’.

Error Messages Are Classifiers

A Process to Design and Evaluate Error Messages

Jack Wrenn
me@jswrenn.com

Shriram Krishnamurthi
sk@cs.brown.edu

Transcript

Coming from a tradition of terse error messages, many of us had severe reservations of such information-dense reports and presenting them was challening. This vibrant multi-color highlighting system let us cut out all source locations from our error reports, and we were able to directly reuse the machinery developed for our information-centric methodology to help us plan these highlights out.

While we have always told our students to "read the error message", it's only now that we've given them error messages that can actually be read in the usual sense.

References i

- Kathi Fisler, *The recurring rainfall problem*, Proceedings of the Tenth Annual Conference on International Computing Education Research (New York, NY, USA), ICER '14, ACM, 2014, pp. 35–42.
- Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi, *Measuring the effectiveness of error messages designed for novice programmers*, Special Interest Group on Computer Science Education (New York, NY, USA), ACM, 2011, pp. 499–504.
- E. Soloway, *Learning to program = learning to construct mechanisms and explanations*, Commun. ACM **29** (1986), no. 9, 850–858.

Transcript

Transcript

 V. Javier Traver, *On compiler error messages: What they say and what they mean*, Adv. in Hum.-Comp. Int. **2010** (2010), 3:1–3:26.