Contextualized Programming Language Documentation

Hannah Potter
hkpotter@cs.washington.edu
University of Washington
Seattle, Washington, USA

Ardi Madadi
ardier@cs.washington.edu
University of Washington
Seattle, Washington, USA

René Just
rjust@cs.washington.edu
University of Washington
Seattle, Washington, USA

Cyrus Omar
comar@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Abstract

Learning the syntax and semantics of a new programming language is a challenge. It is common for learners to refer to language documentation many times and in many contexts as they build comfort and understanding. We review existing functional language documentation, finding that it tends to be organized according to the structure of the language. Each section interleaves narrative explanations, which introduce precise terminology that is then used consistently, with code examples. Sections often start with simpler special cases of a construct before considering it in full generality.

To make use of language documentation, learners must step away from the code they are working with, e.g., in an exercise or tutorial, to locate and transfer knowledge from the documentation. We describe a system, ExplainThis, that automatically generates *contextualized language documentation*, structured based on our study of language documentation but specialized to the particular code at the cursor. This system is integrated into the structure editor of Hazel, a live functional environment. Documentation appears next to the editor and color is used as secondary notation to correlate the explanation with program terms. We also study syntactic and explanatory specificity with a formative user study. We find that participants desire documentation to be tailored to specific syntax of the code a user is working with, while allowing an adaptive level of specificity for code examples.

CCS Concepts: • **Software and its engineering → Development frameworks and environments**.

Onward! '22, December 8–10, 2022, Auckland, New Zealand
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9909-8/22/12.
https://doi.org/10.1145/3563835.3567654

Keywords: functional programming, documentation, programming education, structure editing

ACM Reference Format:
Hannah Potter, Ardi Madadi, René Just, and Cyrus Omar. 2022. Contextualized Programming Language Documentation. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '22), December 8–10, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3563835.3567654

1 Introduction

Learning a new programming language can be difficult, even for experienced programmers. The programmer must become familiar with new syntactic forms. A *syntactic form* refers to an element in a language grammar, such as the rule for a `let...in` expression. A *term* refers to an instance of a syntactic form. An instance of a `let...in` expression could be `let x = 1 in x + 1`. Consider this single-line code snippet, written in an ML-like language:

```
let x : Float = f 1 in x +. 2.
```

It involves different syntactic forms, including a `let...in` expression, a type annotation (`Float`), a function application of `f` to an integer literal (`1`), a floating point operator (`+.`), and a floating point literal (`2.`). These may be unfamiliar, even to a programmer familiar with popular imperative languages.

Most programmers learning a new language do so on their own [28]. They may only have access to static narrative and examples from online tutorials, books, help forums, and language references. Programmers report difficulties with finding relevant documentation, e.g., because they may not yet know the correct terminology to use in an online search [28]. Moreover, while always available, this help is not as adaptive as an expert may be to the specific learning context.

When a programmer is learning a new language, they ideally have access to an engaged expert, such as a teaching assistant or coworker, who can explain the syntax and semantics of code that the learner is trying to understand or write, providing context and varied explanations, reducing the need to search for and through documentation.

This paper proposes a paradigm shift for programming language documentation, away from static standalone documentation toward contextually adaptive systems. It also presents one such contextualized language documentation system, ExplainThis, integrated into a computational notebook style programming environment that displays both narrative and code snippets [11]. Code snippets can appear as read-only examples interleaved into a narrative tutorial, or they can be editable when the user is solving integrated coding exercises. The language documentation is adapted to explain the language constructs underlying the term at the user's cursor, at an adjustable level of specificity.

To inform the design of ExplainThis, we surveyed existing programming language documentation, focusing on functional languages (Sec. 2) and consulted prior work in this area (Sec. 7). We found that documentation emphasizes precise and consistent terminology for syntactic forms, which are organized into "features" largely corresponding to the structure of the language. We also found that narrative explanations tend to be interleaved with examples and that compound or complex language constructs, e.g., destructuring let expressions with support for mutual recursion, were introduced progressively using simpler special cases.

We implemented ExplainThis as a Hazel editor service, which we describe by example in Sec. 3. Hazel is a live functional programming environment with a structure editor that ensures that even partially completed code is syntactically well-formed [16, 18–20]. Because Hazel always provides a syntax tree, ExplainThis is applicable to any code snippet. The novel components of ExplainThis' interface are a syntactic specificity slider, an adaptive explanation system, and an adaptive collection of related examples. All of these are informed by the observations reported in the documentation survey (Sec. 2). While we chose Hazel, ExplainThis' core features could be implemented in editors for a variety of programming languages.

To evaluate and iteratively improve the design of ExplainThis, the narrative explanations, and the chosen examples, we conducted a formative study soliciting feedback from individuals involved or capable of assisting in teaching functional programming courses (Sec. 4 and Sec. 5). In particular, we focused on understanding the extent to which participants value explanations and examples specifically contextualized to the user's source code. We found that participants prefer (1) explanations and examples tailored to some extent to the specific syntax of the user's code, (2) a fixed level of specificity for code explanations, but (3) an adaptive level of specificity for code examples. Additionally, participants value precision and uniformity in the terminology that appears in explanations. Finally, participants value smaller, focused examples. These findings can guide the community in exploring the large design space of contextually adaptive documentation systems (Sec. 8).

2 Language Documentation Survey

Many programming languages provide online language documentation, ranging from informal quick reference guides to tutorials and in some cases, formal language definitions. We surveyed online documentation for three languages to better understand existing documentation practices.

We surveyed the documentation for OCaml [15], Reason [26], and Racket [24]. We chose these partially due to our familiarity with the languages and their documentation and because these are widely-used languages with functional elements that bear many similarities to Hazel.

Our starting point for this survey was each language's official website [15, 24, 26]. We focused on the language documentation that could be found from that starting point. Reading through some of this documentation, including in all cases documentation related to basic variable binding as a focal point, we made notes on various observed features:

- Each language offered different kinds of documentation for different use cases (e.g., quick references, tutorials, books, and formal language definitions). The different kinds of documentation differed in level of detail, assumed different background knowledge, and incorporated examples of varying complexity.
- Documentation was often organized into sections describing different "features" or "constructs" which generally followed the type structure of the language, i.e. it grouped together the introduction and elimination or pattern matching forms of a type (whether or not types were statically checked). Some general features, like let bindings, did not correspond directly to a single type, so they were described separately, often in some simplified form, e.g., let bindings without discussion of mutual recursion or destructuring, and then reoccurred, in specialized form or as general scaffolding in examples, throughout the documentation.
- Within a section, there was also sometimes a progression from simple special cases to more general or advanced modes of use of a construct.
- There was substantial use of links, which (1) often connected less formal and more formal documentation for the same concept and (2) were used to provide context for language constructs that appear incidentally in examples or that have some conceptual relationship to the construct of interest.
- There was extensive use of simple examples that appear to be carefully curated to highlight the construct of interest without requiring a deep understanding of many other constructs. However, there is also generally a progression of constructs such that examples for later constructs will use previous constructs, e.g., functions and let bindings appear in many subsequent examples.

- Introductory documentation often sought to help programmers who are new to the particular language or functional paradigm but not to programming in general by drawing comparisons to other languages or paradigms, e.g., explicit discussion of how functional let binding differs from imperative assignment.
- Precise and consistent use of terminology was used to refer to language constructs and positions within constructs (e.g., the "guard," "then branch," and "else branch" of a conditional expression).

3 Design of ExplainThis

ExplainThis is designed (1) to serve a similar role as quick reference documentation, so it incorporates many of the observations relevant to that kind of documentation as discussed in Sec. 2 as well as providing multiple levels of documentation for the same concepts, and (2) for users who are not novices to programming, but that may be novices to functional programming.

Potential use cases of ExplainThis include code reading and comprehension (e..g, a user trying to understand existing functions in a library or examples in instructor-created tutorials); writing new code (e.g., a user trying to modify starter code to complete an assignment); or debugging (e.g., a user trying to understand unexpected behavior of a function).

Here we present the initial design of our tool using a motivating example. We note that the design presented is preliminary, representing a first effort toward the vision presented in this paper, and our intention is to refine it based on the feedback we received in our user study (Sec. 5).

3.1 ExplainThis by Example

We explain our design for ExplainThis by example. The hypothetical subject of our use case is a student who has multiple years of experience programming with imperative programming languages, such as C++, and is now taking a course where they are introduced to the functional programming paradigm for the first time. In particular, the course uses Hazel, a functional programming language. The student has been given a formal introduction to the language in lecture and is now working through exercises to strengthen their understanding. The student is currently working through an exercise to practice using the standard `map` function. The starter code for the exercise can be seen in Fig. 1. As part of the starter code, the student is given a reference implementation for `map`[1]. Additionally, the student is given the skeleton of a function, `prod`. The exercise is for the student to implement the body of the `prod` function, which should take a `List` of `Int` two-tuples and return a `List` containing the products of the elements of each tuple.

The student first wants to make sure that they understand how the `map` function works. The student selects the body of the function and sees the syntactic form documentation shown in Fig. 1. The three dialogs that form the documentation system are the Syntactic Form, Code Explanation, and Code Examples dialog.

The Syntactic Form dialog presents the student with information about the syntax of the term on which their cursor is currently placed, which here is a `case` expression. The editor indicates which syntactic form is being documented with a green background on the parts of the syntax that are at the root of the term. The dialog displays varying levels of details about its subforms using meaningful labels which can be controlled by the user using the slider at the bottom of the box. In Fig. 1, the student is first shown the most general syntactic form of a `case` expression.

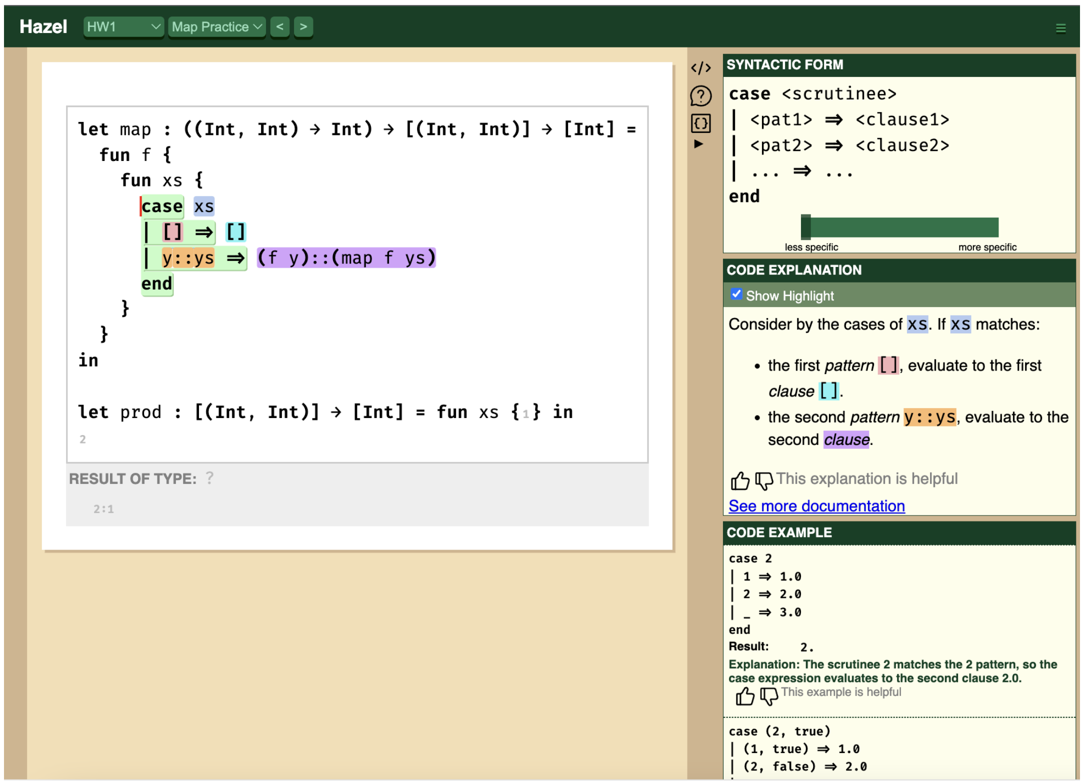
The Code Explanation dialog provides a natural language explanation of the code that the cursor is on relative to the level of specificity of the indicated syntactic form. Code highlighting uses matching colors to relate the parts of the explanation to the associated source code. Terminology, such as "clause", is indicated with italics. In Fig. 1, the student is first shown an explanation of the `case` expression that makes up the body of the `map` function. This explanation is relative to the general syntactic form of a case via the syntactic form slider, but is tailored to the particular `case` expression in the code by inlining small pieces of code, the color highlighting used, and the number of rules presented. Larger pieces of code are referred to using the corresponding natural language terminology, e.g., "clause."

The code example box provides one or more examples of the term the cursor is on, relative to the level of specificity of the indicated syntactic form. Examples include information about the result of evaluating the example and a natural language explanation. In Fig. 1, the student is presented with a few examples they can use to try to better understand the body of the `map` function and `case` expressions in general.

After reviewing the information of the syntactic form, code explanation, and code example boxes, the student feels that they better understand how general case analysis works in Hazel. However, they still do not fully understand how the structural pattern matching in the `map` body works. The student wants to understand a more specific syntactic form than general `case` expressions, so they move the syntactic form slider to "more specific" (Fig. 2a and Fig. 2b). The more specific syntactic form gives explanations and examples for cases of an empty list and a non-empty list. After seeing this more specific explanation and more examples, the student feels more confident in their understanding of how the pattern matching works in the body of the `map` function. The student can select any part of the `map` function to understand other pieces following the same process.

When the student feels more confident in their understanding of the `map` function, they feel ready to start implementing

[1]Polymorphism is not currently implemented in Hazel, so we use an implementation of `map` with the specific types needed for the example.



```
let map : ((Int, Int) → Int) → [(Int, Int)] → [Int] =
  fun f {
    fun xs {
      case xs
      | [] ⇒ []
      | y::ys ⇒ (f y)::(map f ys)
      end
    }
  }
in

let prod : [(Int, Int)] → [Int] = fun xs {1} in
2
```

RESULT OF TYPE: ?

2:1

SYNTACTIC FORM

```
case <scrutinee>
| <pat1> ⇒ <clause1>
| <pat2> ⇒ <clause2>
| ... ⇒ ...
end
```

CODE EXPLANATION

Show Highlight

Consider by the cases of `xs`. If `xs` matches:

- the first *pattern* `[]`, evaluate to the first *clause* `[]`.
- the second *pattern* `y::ys`, evaluate to the second *clause*.

This explanation is helpful

See more documentation

CODE EXAMPLE

```
case 2
| 1 ⇒ 1.0
| 2 ⇒ 2.0
| _ ⇒ 3.0
end
```

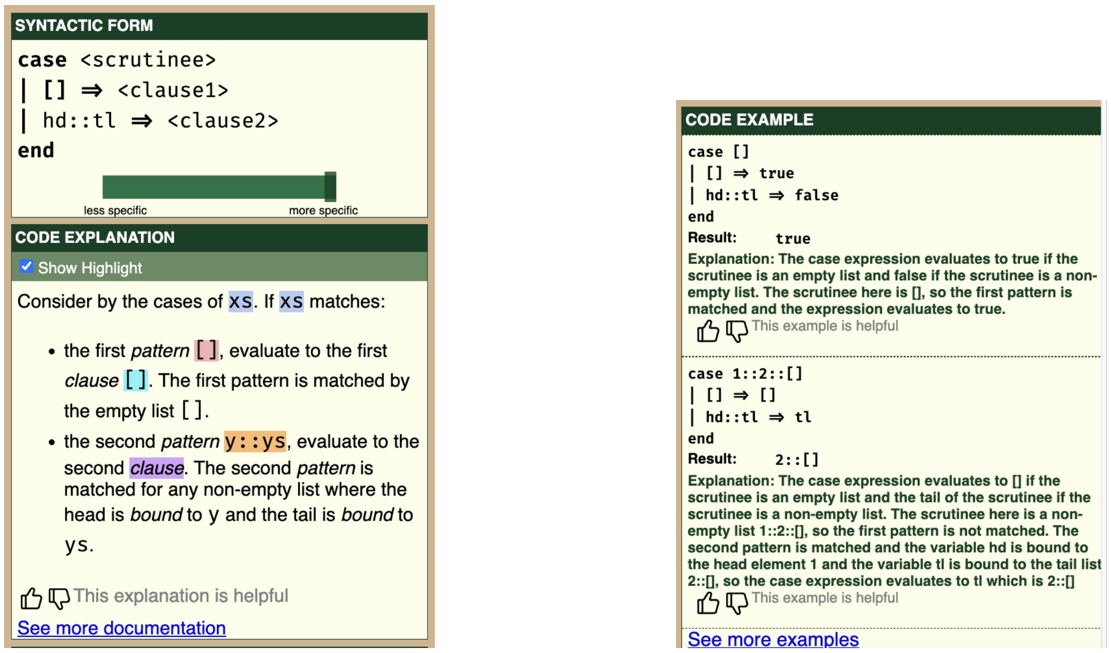
Result: 2.

Explanation: The scrutinee 2 matches the 2 pattern, so the case expression evaluates to the second clause 2.0.

This example is helpful

```
case (2, true)
| (1, true) ⇒ 1.0
| (2, false) ⇒ 2.0
```

Figure 1. ExplainThis for Hazel. The editor is shown on the left and ExplainThis is shown in the right panel. The programmer's cursor is at the beginning of the case expression. In Hazel, the term where the editor cursor is currently positioned is highlighted in light green. Holes are displayed with light-grey hole identifier numbers, such as the body of the `prod` function.



SYNTACTIC FORM

```
case <scrutinee>
| [] ⇒ <clause1>
| hd::tl ⇒ <clause2>
end
```

CODE EXPLANATION

Show Highlight

Consider by the cases of `xs`. If `xs` matches:

- the first *pattern* `[]`, evaluate to the first *clause* `[]`. The first pattern is matched by the empty list `[]`.
- the second *pattern* `y::ys`, evaluate to the second *clause*. The second *pattern* is matched for any non-empty list where the head is *bound* to `y` and the tail is *bound* to `ys`.

This explanation is helpful

See more documentation

CODE EXAMPLE

```
case []
| [] ⇒ true
| hd::tl ⇒ false
end
```

Result: true

Explanation: The case expression evaluates to true if the scrutinee is an empty list and false if the scrutinee is a non-empty list. The scrutinee here is [], so the first pattern is matched and the expression evaluates to true.

This example is helpful

```
case 1::2::[]
| [] ⇒ []
| hd::tl ⇒ tl
end
```

Result: 2::[]

Explanation: The case expression evaluates to [] if the scrutinee is an empty list and the tail of the scrutinee if the scrutinee is a non-empty list. The scrutinee here is a non-empty list 1::2::[], so the first pattern is not matched. The second pattern is matched and the variable hd is bound to the head element 1 and the variable tl is bound to the tail list 2::[], so the case expression evaluates to tl which is 2::[]

This example is helpful

See more examples

(a) More specific syntactic form and code explanation. **(b)** More specific code examples.

Figure 2. Syntactic form and code explanation (left) and code examples (right) of a more specific form of a case expression.

the prod function. The student stubs out the implementation, knowing that map needs to be called on some function and the input list to prod, xs . The student stubs out the function that will be used for map, following the other functions given in the starter code which use a single variable pattern (Fig. 3).

At this point, the student needs some more help understanding how to write the function that will be used by map. The documentation gives them specific information about functions that use a single variable pattern (Fig. 4). However, this very specific syntactic form of a function is not what the student wants because they need to extract the elements of the tuples in the lists. The student moves the syntactic form slider to the more general form of a function to try to understand how their inner function stub can be edited to have the functionality they need (Fig. 5). The student can see from this more general documentation that functions are not restricted to a single variable pattern for the argument, but rather can use a different pattern. Using this information, the student is able to edit their inner function stub to use a tuple pattern and from there is able to complete their implementation of the prod function.

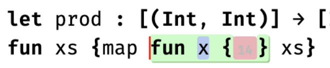
The subsequent sections provide a more detailed description of ExplainThis.

3.2 Hazel Tutor

Hazel is a browser-based system [16]. Editing in Hazel is done through a structure editor. One of the key features of Hazel is that all program states are syntactically well-formed and semantically meaningful, having both a type and a non-trivial result [18, 19]. This is accomplished by automatically inserting *empty holes* to stand for missing parts of a program as well as wrapping *non-empty holes* around parts of a program with semantic errors. With every program state being meaningful, the “gap-problem” is solved, where in traditional programming environments tools either only work partially or not at all for incomplete programs. In Hazel, tools are able to give feedback when programs are incomplete, which is when programmers often need assistance.

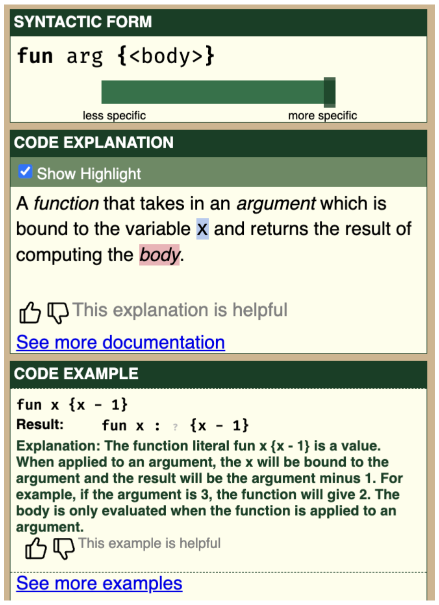
Hazel currently has tools that make use of typing information in the program [21]. These tools are the Cursor Inspector, which displays information about the expected and actual types of patterns and expressions in a program, and the Strategy Guide, which is designed to guide novices through a type-driven development strategy, presenting a hierarchical list of valid edits for empty holes. These tools are part of the Hazel Tutor project, which is focused on developing assistive technology specifically targeted at novices to typed functional programming.

Here we present the design for another tool in the Hazel Tutoring system, ExplainThis, to provide novices with contextualized programming language documentation. While the Cursor Inspector is designed to surface information about the static semantics of a program, this tool gives information about the syntax and dynamic semantics of a program. The



```
let prod : [(Int, Int)] → [Int] =
fun xs {map fun x {14} xs}
```

Figure 3. Initial stub for the function used by map which uses a single variable pattern for the argument.



SYNTACTIC FORM

```
fun arg {<body>}
```

CODE EXPLANATION

☑ Show Highlight

A *function* that takes in an *argument* which is bound to the variable x and returns the result of computing the *body*.

This explanation is helpful

See more documentation

CODE EXAMPLE

```
fun x {x - 1}
```

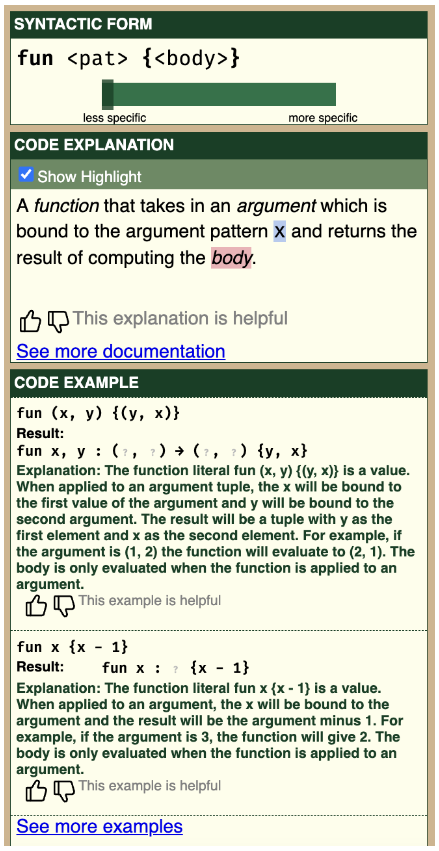
Result: `fun x : ? {x - 1}`

Explanation: The function literal fun x {x - 1} is a value. When applied to an argument, the x will be bound to the argument and the result will be the argument minus 1. For example, if the argument is 3, the function will give 2. The body is only evaluated when the function is applied to an argument.

This example is helpful

See more examples

Figure 4. Documentation for functions that use a single variable pattern for the argument.



SYNTACTIC FORM

```
fun <pat> {<body>}
```

CODE EXPLANATION

☑ Show Highlight

A *function* that takes in an *argument* which is bound to the argument pattern x and returns the result of computing the *body*.

This explanation is helpful

See more documentation

CODE EXAMPLE

```
fun (x, y) {(y, x)}
```

Result:

```
fun x, y : (?, ?) → (?, ?) {y, x}
```

Explanation: The function literal fun (x, y) {(y, x)} is a value. When applied to an argument tuple, the x will be bound to the first value of the argument and y will be bound to the second argument. The result will be a tuple with y as the first element and x as the second element. For example, if the argument is (1, 2) the function will evaluate to (2, 1). The body is only evaluated when the function is applied to an argument.

This example is helpful

```
fun x {x - 1}
```

Result: `fun x : ? {x - 1}`

Explanation: The function literal fun x {x - 1} is a value. When applied to an argument, the x will be bound to the argument and the result will be the argument minus 1. For example, if the argument is 3, the function will give 2. The body is only evaluated when the function is applied to an argument.

This example is helpful

See more examples

Figure 5. Documentation for general functions.

tool's interface will be displayed when a user requests documentation for the term at their current editor cursor position. One of the key goals of this interface is to give a novice to Hazel all the information they would need to understand a term that is present in their program without needing to consult external documentation.

3.3 Syntactic Form

In ExplainThis, the explanations and examples can be tailored to different level of specificity of the syntactic form. In general, the most specific syntactic form can be thought of as the most simple version of a form that would be used pedagogically to teach it. For instance, for a `let` expression, `let`-binding to a single variable is a specific version of a `let` expression that is often used to introduce this syntactic form. In contrast, a `let` expression, `let`-binding with patterns and mutual recursion is the general (least specific) syntactic form. A programmer can scrub back and forth on this slider to tune the explanations and examples to the level of specificity that is most useful to them for a given task.

The system attempts to start at a suitable level of generality that captures the particular syntax that the cursor is on, without including features that are unused. We are not at this time trying to incorporate any machine learning techniques to infer user intent or knowledge of a programming language. In the future, we may consider trying to model a user's current knowledge based on syntactic forms they have seen or used in the system.

3.4 Code Explanation

Explanations can be tailored to both the code on which a user's cursor is placed as well as the chosen level of specificity on the syntactic form slider. The explanations demonstrate correct uses of terminology used for the syntactic forms, such as can be seen by the italicized text. Code highlighting uses matching colors to relate the parts of the explanation to the associated source code. Similar highlighting is used in Pyret, a language designed for novices [22]. The code explanation highlighting can be toggled on and off to minimize potential distraction to the user.

3.5 Code Examples

Examples can also be tailored to both the code on which a user's cursor is placed as well as the chosen level of specificity on the syntactic form slider. The goal is to keep the examples relevant to the code that the programmer is actually trying to explore and understand rather than a small selection of general examples that may be difficult for the programmer to map back to their code context. Additionally, we want examples that highlight the particular aspects of the code the programmer is trying to understand, as informed by the user's selection with the syntactic form slider.

3.6 Other Features

Other features of the design include links to external documentation and feedback indicators. External documentation can be reached through either the link to see more documentation shown in the Code Explanation dialog or the link to see more examples shown in the Code Example dialog, both seen in Fig. 5. The external documentation will be static learning material created for each of the syntactic forms in Hazel. The goal is that the programmer will rarely need to consult this external material in order to understand the syntactic forms, but that they will have all of the information they need right there in the programming environment. The feedback indicators shown with the thumbs-up and thumbs-down icons seen in Fig. 5 allow users to quickly indicate if they found any part of the contextualized language documentation helpful or unhelpful. This feedback can be used to improve messaging.

4 Evaluation Methodology

A key goal of ExplainThis is to provide documentation that is similarly beneficial as hand-curated explanations by experts. We conducted a user study designed to explore the question of what properties of explanations and examples experts believe would be most beneficial to someone trying to understand a given syntactic form. Our participant pool consisted of 10 programmers who had the necessary knowledge of functional programming for our study. Our IRB-approved study consisted of three phases: a screening survey, a background survey, and an interview. Participants who completed all three phases were compensated with a $25 gift card.

4.1 Screening Survey

All participants completed a screening survey to ensure they had the required knowledge to participate in our study. The screening survey consisted of five multiple-choice, beginner-level functional programming questions to assess whether a candidate had a basic understanding of topics such as structural pattern-matching, curried functions, and `let` bindings.

4.2 Background Survey

Participants who received a satisfactory score, defined as missing no more than one question on the screening survey, were asked to complete a background survey to collect data on their functional language experience and demographics.

4.3 Interview

Participants who passed the screening survey were invited to complete an interview. We conducted three in-person and seven virtual interviews. Interviews were designed to last about 75 minutes and included a 15-minute instructional demo followed by a 60-minute survey. The participants' screen activity and audio were recorded during the interview.

4.3.1 Instructional Demo.

The demo was designed to acquaint the participants with the design space for ExplainThis. Each participant was briefed on the motivation behind ExplainThis and its goals.

4.3.2 Prompts.

Following this introduction, participants were asked to respond to a series of nine prompts. We designed the prompts to explore a range of explanations and examples to understand what properties participants considered when determining how beneficial they found the options. Additionally, we designed the prompts to provide insights into how a syntactic form is being used and how an indicated level of specificity for the syntactic form influenced participant rankings.

Each prompt included the following components:

1. **Prompt title:** A descriptive title of the prompt.
2. **Code snippet:** Uneditable code for which the participants were asked to rank a set of explanations and examples. The cursor was placed on a predefined term.
3. **Result:** The computed result of the code snippet.
4. **Syntactic form:** The syntactic form for the selected term in the code snippet. It displays the form along with a disabled slider which indicated the level of specificity of the syntactic form.
5. **Code explanation:** Various explanations for the code snippet which the participants were asked to rank. When hovered over, both the explanations and the code snippet were simultaneously color-highlighted with different colors mapping corresponding parts of the explanation and the code snippet together.
   Participants were asked to rank each explanation with respect to the code and the syntactic form using ranking drop downs which were displayed to the left of each explanation. Participants were asked to rank the options according to what they thought would be most beneficial to a programmer who is trying to understand the indicated syntactic form when looking at the specific code snippet. The explanations could be ranked from 1 to N, where N was the number of explanations. The participants were asked to avoid assigning identical scores to the different options within the same prompt. Each ranking drop down had an extra option for participants to mark the explanation as not useful. Additionally, a free response text box was provided for participants to give any alternative explanations that they thought would be more beneficial to the potential users of our tool than the options presented.
6. **Code example:** Various code examples for the source code were provided. The participants had the option to expand (and collapse) the examples to see the result of each example's computation and an individual explanation for that example.
   Similarly to the code explanations, code examples also included a ranking drop down as well as a text box for alternative examples and feedback.

Participants could navigate back and forth between the prompts using arrow buttons and a drop-down prompt selector. The order in which explanations and examples were presented was randomized.

4.3.3 Specificity Levels and Syntactic Forms.

We used nine prompts, including four sets of two prompts where the pairs of prompts were identical except for the level of specificity (syntactic form slider). Across the nine prompts, there were five different syntactic forms of focus: a `case` expression (**Case**), a function application with a tuple parameter (**FA w/ Tuple**), a curried function application (**FA Curried**), a function literal (**Fun Lit**), and a `let` expression (**Let**). The order of prompts was randomized except that the prompts that were identical modulo the specificity slider were presented in adjacent order, the less specific level always being presented first.

4.4 Artifact Availability

The user-study artifacts (e.g., surveys, scripts, executable) are available at: https://doi.org/10.6084/m9.figshare.21381864.

5 Results

5.1 Participant Makeup

We had 10 participants complete the interview phase of the study. Four of the participants self-identify as women. Seven of the participants are graduate students. All participants have taken or are currently taking at least 5 computer science courses, with eight participants having taken more than 10. Eight of the participants have some computer science industry experience (including internships). Four of the participants have formal teaching experience in a programming languages or programming paradigm course. Nine of the participants have some experience studying from a programming languages or programming paradigm course. The most common functional programming languages that participants reported having at least a little experience with were OCaml (7 participants), Racket (8 participants), and Haskell (8 participants). Four of the participants have at least some experience with structure editors.

5.2 Features of Explanations and Examples

Based on observations from the documentation survey, different designs and questions about good properties of explanations and examples we have considered, interviews with participants, and our own review of properties of the explanations and examples we gave participants, we derived labels of properties that capture characteristics of our explanations (see Table 1) and examples (see Table 2) of syntactic forms.

Table 1. Properties of code explanations. Syntactic form refers to the outermost form of the expression the cursor is on.

Group	Label	Sublabel	Meaning
Technicalities	*Gist/Intuition*		Trying to convey the essence of a syntactic form rather than technical details not necessarily needed for being able to work with the form in the language
	Technical Details		Uses technical language and details for information about the syntactic form
Levels of Explanation	*Explains Subforms*		Specializes messages based on subforms
		Separated Subform	Information about a subform is visually separated from the information about the outer forms
		Incorporated Subform	Information about a subform is incorporated into the information about the outer forms
	General		Uses an explanation of the general syntactic form
Verbalization	*Verbalization*		Gives a verbalization of the syntax, but does not add any additional explanatory information
Evaluation Order	*Out of Order*		Structure does not follow the evaluation order
	Explicit Order		Explicitly conveys the evaluation order

Table 2. Properties of code examples. Syntactic form refers to the outermost form of the expression the cursor is on.

Group	Label	Meaning
Match Code Snippet	*Matches Overall Subform Shape*	Matches overall shape (e.g., tuples of any size, curried function application) of the subterms of the code snippet
	Matches Context	Matches syntax of parts of code snippet outside the focused syntactic form (does not apply if no forms outside focused form)
	Matches Types	Matches all the types in the code snippet for the syntactic form (based on declarative typing)
	Varies Subform Shape	Uses different subform shapes from those used in the code snippet (e.g., tuples of different size)
	Varies Types	Uses different types from those used in the code snippet
Details	*Highlight Details*	Highlights details about the syntactic form, including potential sources of confusion or special cases
Complexity	*Uses Meaningful Let Bindings*	Uses let bindings to name pieces of the example to incorporate "meaning"
	Extra Forms	Wraps the syntactic form in other forms (excluding let bindings) or uses more complex subforms than in the code snippet (e.g., case)
	Use of Form	Highlights how a base (not operation) syntactic form is used (e.g., form is a function literal and the example shows it being applied)
	Concise	Uses a concise (minimal) example of the syntactic form—no extra wrapping forms nor complicated subforms
Evaluation	*Does Not Evaluate to Simple*	Evaluates to a function

Table 3. Properties and voter rankings of code explanations. Lower ranks indicate more preferred explanations. LS and MS are the ranking votes for *less* and *more* specific, respectively. (‘–’ = no prompt for the associated specificity level.)

	Technicalities		**Explanation**			**Verbalization**	**Evaluation Order**		**Rank**	
	Gist	*Details*	*Subforms*		*General*		*Out of Order*	*Explicit*	*LS*	*MS*
			Sep.	*Inc.*						
Case		✓	✓	✓					1	–
		✓	✓			✓			2	–
		✓			✓			✓	3	–
FA w/ Tuple		✓	✓						1	1
		✓			✓	✓			2	2
	✓			✓					3	3
FA Curried		✓		✓				✓	1	1
		✓	✓						2	2
		✓			✓	✓			3	3
	✓			✓					4	4
Fun Lit		✓			✓	✓	✓	✓	1	3
	✓			✓					2	2
		✓		✓			✓	✓	3	1
Let		✓			✓	✓			1	2
		✓		✓			✓		2	1
	✓			✓			✓		3	3

Table 4. Properties and voter rankings of code examples. Lower ranks indicate more preferred examples. LS and MS are the ranking votes for *less* and *more* specific, respectively. (*MS* = Matches Overall Subform Shape, *MC* = Matches Context, *MT* = Matches Types, *VS* = Varies Subform Shape, and *VT* = Varies Types; ‘–’ = no prompt for the associated specificity level.)

	Match Code Snippet					**Details**	**Complexity**				**Evaluation**	**Rank**	
	MS	*MC*	*MT*	*VS*	*VT*		*Let*	*Extra Forms*	*Use*	*Concise*		*LS*	*MS*
Case	✓		✓			✓						1	-
				✓	✓					✓		2	-
	✓				✓	✓						3	-
				✓	✓		✓	✓	✓			4	-
FA w/ Tuple	✓	✓		✓	✓		✓					1	1
		✓		✓	✓		✓					2	3
				✓	✓					✓		3	4
	✓	✓			✓		✓	✓				4	2
FA Curried	✓			✓	✓							1	1
	✓	✓		✓	✓		✓					2	3
	✓	✓			✓		✓	✓				3	2
		✓		✓	✓		✓					4	5
				✓	✓	✓					✓	5	4
				✓	✓					✓		6	6
Fun Lit				✓	✓	✓				✓	✓	1	3
	✓			✓	✓		✓	✓	✓			2	1
				✓	✓			✓	✓			3	4
	✓				✓		✓	✓	✓			4	2
				✓	✓		✓	✓	✓			5	5
Let	✓		✓									1	1
				✓	✓					✓		2	3
	✓				✓			✓				3	2

We then used these properties to label the explanations and examples shown to participants (see Table 3 and Table 4). The purpose of this labeling process is to better understand properties of explanations and examples that participants consider to be beneficial rather than develop a general labeling scheme.

5.3 Rankings of Explanations and Examples

To aggregate participants' votes for different explanations and examples, we used the Single Transferable Vote method, which can be used where voters rank candidates (in this case explanations and examples) according to their preferences [25]. We used the `vote` package in R to get the complete ranking of candidates running for N - 1 seats (where N is the number of explanations or examples for a given prompt). We allow for equal rankings, which automatically corrects rankings that would otherwise be considered invalid (e.g., rankings 2, 3 would be corrected to 1, 2). Explanations or examples that were not ranked or marked as not useful by participants are converted to "NA" votes. Votes for participants who have "NA" votes for all explanations and examples are considered invalid and will not be automatically corrected. Table 3 and Table 4 show the participants' vote rankings.

5.4 Discussion

Four total votes were invalid, all for the more specific syntactic form level: one for the examples for FA w/ Tuple, one for the explanations for FA w/ Tuple, and two for the examples for Fun Lit. Both of the invalid votes for FA w/ Tuple correspond to a participant not answering a prompt. The two invalid votes for Fun Lit examples both correspond to participants indicating that none of the given examples were useful. Both of these participants wanted an example that more closely matched the indicated level of specificity of the syntactic slider, in this case an unapplied function literal with a tuple argument pattern; two other participants gave similar free response examples. This aligns with our general observation that participants wanted the examples shown to be influenced by the level of specificity.

We observe that the level of specificity of the syntactic slider in general did not affect rankings as much for explanations as it did for examples, when considering total rankings. This is consistent with comments made by participants. One participant said, "...I'm kind of tempted to give the same rankings for all of the code explanations and then basically change the code examples for the specificities... I think that's what I've basically been doing... subconsciously...." Another participant when asked about these trends responded "... I think that [the] syntactic form should... change the examples more than the explanations...." For explanations, two of the four prompts that used multiple levels of specificity have identical rankings. However, we also observe that for the example rankings when only considering the top ranked example, three of the four applicable prompts are identical between syntactic form specificity levels. Aligned with these observations about the influence of the syntactic form specificity level, we observe that for explanations for a more specific syntactic level, all top ranking choices were specialized to explain subforms. These observations indicate that explanations and examples should be tailored both to code that a programmer is exploring and to particular questions that a programmer is asking about the code.

Our current contextualized programming language documentation design uses the syntactic form slider as a proxy for what a programmer is trying to understand about a piece of code. Some participants indicated that the current design of the slider may not be a natural way for a user to express that. Two participants indicated the syntactic form indicator did not fit how they consider code, stating it was not how their "brain understands code" and not the way their "brain works." One participant suggested, "...I feel like it might be helpful to have not like a linear sort of like less specific to more specific about the syntactic form, but like more selective which specific features are confusing...." This opens the design question for an intuitive way for programmers to express these questions, which is left to future work.

Considering properties of technical correctness, we observe that in general, participants did not prefer explanations that elided technical details; three of the four explanations that gave the gist/intuition of the syntactic form were given the lowest ranking for both levels of syntactic form specificity, with none receiving the highest ranking. These observations indicate that explanations should use correct and precise terminology.

For examples, we observed that participants generally do not prefer examples that used extra, complex syntactic forms, even when those forms gave more "meaningful" examples or demonstrated a use of a literal syntactic form. We observe that except for one example, no example that has the "Extra Forms" label has a top-ranking. Rather, participants preferred examples of the syntactic form with focused details relevant to how the form is actually used in the code and the level of specificity of the syntactic form slider.

We plan to refine our design and messages based on our observations and the feedback we received in the formative study, including developing a systematic way to define explanations and examples. We also plan to investigate how helpful these designs are to users in practice (see Sec. 6).

5.5 Validity

We will now discuss the validity of our study design. For external validity, we do acknowledge that our participant sampling method is primarily a convenience sample. This may impact the generalizability of our results. However, this sample should be a fairly good representation of the target population for the study, which are programmers with enough knowledge of functional language concepts to competently teach a course that uses a functional programming language. We did recruit from multiple labs and universities to get a more diverse sample of participants. For internal validity, we did screen our participants to make sure they had the necessary background knowledge to provide informative responses to our task prompts. For construct validity, our tasks are designed to cover a range of code explanations and examples. Relevant tasks are framed with approximately the same limited level of information about the code that ExplainThis will be working with.

One threat to validity is that some of the participants ran into a bug that required reloading the study infrastructure, which re-randomized prompts and made it so that participants were not able to look back on their previous answers. Additionally, some participants mixed up the order for which ranking is "better." We asked participants to flip their orders when we noticed this, but it is possible that some unintended orderings were made. Another threat to validity is that the questions asked by the interviewer may have swayed participant responses.

Our study is focused on ExplainThis, contextualized documentation for Hazel, aiming to generalize our results to the various syntactic forms in this particular environment. Our observations may not generalize to other programming paradigms, languages, or environments.

6 Future Work

There are many open avenues for further exploration of contextualized programming language documentation. In particular, it remains for future work to investigate use of ExplainThis by our target user: experienced programmers who are new to a particular programming language. Our formative study (Sec. 4) focused on properties of explanations and examples that experts would choose to provide a programmer learning a new language, as well as insights into our tool design. After design iteration based on the results of this study, the next step is to evaluate our tool and messages on those who fit our target audience. We plan to deploy an implementation of our tool to a university-level programming languages course, where many students are learning not only Hazel, but the functional paradigm for the first time. We can gather light-weight feedback from the students using the feedback indicators (Sec. 3.6), as well as analyzing logging information about tool use. This information will provide better understanding of the effectiveness and usability of our design and messages.

Along with exploring tool use from the learner perspective, an area for future work is investigating ExplainThis from the message author's point of view. Currently, all explanations and examples as well as the syntactic structures they each apply to is written into Hazel source code. It remains for future work to find easier ways for an educator to specify syntactic structures and specialize messages to suite particular educational needs and goals. Additionally, expanding our contextualized documentation system to give specialized messages for APIs or user-defined syntax, e.g., via macros [17], may be useful to programmers trying to work with unfamiliar APIs. It could be beneficial for authors of such APIs or newly defined syntax to easily be able to define their own documentation to work within our system.

Another line of work is looking at incorporating more information about a program into the contextualized documentation. Our current design and implementation only uses the syntactic structures of a program, but there are other forms of semantic information available (e.g., types) that could potentially be utilised to provide more contextualized, meaningful, and useful documentation to programmers.

Future iterations of Hazel could leverage contextualized documentation that considers the program context and allows the user to query for particular functionalities using natural language, similar to CodeMend [27]. Additionally, allowing users to interact with presented code examples, such as by stepping through execution and editing, may be a useful augmentation to the current presentation in ExplainThis. This feature was desired by one of our participants and is present in tools such as Nota [3] and Python Tutor [5].

7 Related Work

ExplainThis's design, and future refinements thereof, draws from research on novice programmers, contextualized explanations, and leveraging natural language knowledge for understanding code.

7.1 Novice Programmers

As a tool that is primarily targeted at programmers who are novices to a programming language or paradigm, this work has been influenced by research on novice programmers, as well as some influence early in the design process from [33].

One pedagogical device for teaching programming is the notional machine [4]. A notional machine is an abstract notion of a physical machine that only contains the level of detail needed to meet the desired level of understanding. A student then has a mental model of the notional machine, but this mental model may not be accurate. Thus, a goal of teaching is to align the student's mental model with the notional machine. Notional machines could be used to teach formal programming language semantics. A recent Dagstuhl Seminar looked at ideas around formal programming language semantics and notional machines [6]. One goal of ExplainThis is to help build and correct students' mental models of the semantics of syntactic forms in Hazel. In this way, the documentation tool is teaching a notional machine of Hazel.

The computer science education community studies introductory programming and its challenges more broadly. One literature review categorizes prior works' characterization of student knowledge into (1) syntactic knowledge, which is the most basic and involves knowledge of language features; (2) conceptual knowledge, which is related to mental models; and (3) strategic knowledge, which involves applying syntactic and conceptual knowledge to solve new problems [23]. Some of the misconceptions students may struggle with described in this literature review include cognitive load, mental models, and misuse of prior knowledge. Some mitigation strategies and tools described include visualizations, novice friendly IDEs and programming languages, worked examples, and teaching students to read code (not just write code). Another recent literature review on introductory programming focused on computing majors, not just computer science, and included similar ideas to help novice programmers such as teaching code-reading, not just code-writing, skills and discussion of cognitive load theory, which says that students perform worse when they need to remember more than can fit in their working memory [13]. We believe that ExplainThis may help with a variety of these identified struggles of novice coders, including learning to read code by providing natural language explanations of syntactic forms.

Drawing on ideas from cognitive load theory, a study looked for the modality effect in programming [14]. The modality effect posits that using multiple senses to communicate complementary, but not duplicated, information can improve learning. For instance, this may support the teaching practice of live-coding where a teacher talks (audio) about some code (visual). The study investigated if changing the modality of instruction (text, oral, or both) improved learning outcomes for information retention and transfer. Interestingly, they did not find supporting evidence that the modality principle held for programming. This may indicate that textual explanations of code, such as those proposed for Hazel, may still be effective for teaching students programming even though novices may view code as text (whereas experts view code more as diagrams) [14].

7.2 Contextualized Explanations

The design of ExplainThis is also influenced by recent work and design recommendations for contextualized explanations of technical information.

Recent work has gone into developing tools that give users contextualized explanations in a variety of domains. One such work is ScholarPhi [8]. This tool automatically generates context-sensitive information about nonce words in scientific papers. Nonce words are technical terms and symbols that have a specific use in a specific paper. Key features of ScholarPhi include tooltips for symbols that give information such as definitions, greying out all text in a paper other than what is relevant to a selected term, equation diagrams that show multiple definitions of symbols in an equation at the same time, and a glossary of terms in the paper. The tool was designed following an iterative design process and gives the following seven design principles for "in-situ" definitions for scientific text [8]:

1. Tailor definitions to the location of appearance. Since symbols in scientific text can have different meanings throughout a text, the definitions shown should be about the specific location the reader is in the text.
2. Connect readers to definitions in context. ScholarPhi realizes this by including with definitions links to where in the paper those definitions were given.
3. Consolidate information. This includes showing the subsymbols information of a given symbol together.
4. Provide scent. Give indications to the reader for what symbols they can get more information.
5. Minimize occlusion. Readers do not want important text information to be hidden by the tool, but they also do not want to lose where they are in the text by having to look elsewhere for additional information. ScholarPhi manages these two conflicting desires by using condensed tooltips. Additional information can be viewed in a sidebar.
6. Minimize distractions.
7. Support error recovery. Because the information used in ScholarPhi is automatically generated from the scientific text, there are possible errors from which the reader needs to be able to recover.

ExplainThis tailors definitions to the particular use of a syntactic form in the source program, showing clear links between the code in the editor and explanations, including information about subforms with the currently considered syntactic form, not covering the code that is being investigated, and minimizing distraction by allowing the user some control over display highlighting (see Sec. 3).

Nota [3], a tool similar to ScholarPhi [8], is designed to give a more interactive way to consume programming language research literature than static PDFs. The tool works in the browser and includes functionality such as dynamic code snippets, tooltips, definitions and references, flexible layouts, convenient importing of language grammars, interactive diagrams, a way to show and hide less critical information, and buttons to toggle between more and less formal definitions. Crichton designed Nota based on the following principles [3]:

1. A reader should always be able to access the definition of a symbol.
2. Jumping around is bad — definitions should be visible in context.
3. A static display is preferable to an interactive one, all else being equal.

ExplainThis follows these principles, particularly that the documentation is always available to the programmer because of Hazel's semantic guarantees [18, 19] and that the documentation is presented alongside the context in which the syntactic form is used. We may consider providing an editable sandbox for programmers to interact with code examples in future iterations of our documentation design.

Another tool that gives contextualized explanations is Tutorons [7]. This tool gives automatic and queryable explanations for code that is found in online sources such as web code tutorials. These explanations only give information relevant to the specific use of the code that the user is investigating. Explanations have multiple levels, including high-level descriptions and more detailed information. The system supports multiple languages, where each language support is a Tutoron. The authors give the following guidelines for explanations in their system [7]:

1. Use multiple representations to illuminate high-level intent and enable low-level understanding of syntax.
2. Be concise – skip static explanations and focus on dynamically generated content.
3. Reappropriate existing documentation.
4. Inspect code examples on a large scale to support explanations of common usage.

ExplainThis gives multiple levels of explanations and uses dynamically generated explanation content (see Sec. 3).

7.3 Natural Language

There has been research into making use of natural language knowledge for performing programming tasks. Using fMRI, Siegmund et al. showed that learners of a new programming language leverage the same areas of the brain as those of learners of a natural language [29]. A study of 49 schoolchildren between the ages 9 and 13 showed that reading code aloud improved their ability to remember the syntax, but did not necessarily make a difference in their understanding of the code [32]. Here we will describe two lines of work. One aims at programming languages which have syntax that is more reflective of natural language. Another aims at giving natural language versions of code.

One study on the MOOSE programming language was seeking to understand if there were potential downsides to natural language programming, which has generally been used for application-specific and end-user programming [2]. MOOSE is a programming language that was specifically designed to be used by children to create text-based worlds. The study looked at natural language errors in the language, which were defined as errors where the command the child was trying was more "English-like" than the correct command. The researchers looked at several children's programming logs and found that 10.6% of the programming errors were natural-language errors, and 41.1% of these errors had easy recovery. The researchers concluded that "making use of people's preexisting natural language knowledge is an effective strategy for programming language design for children, end users, and others new to coding."

Other work has investigated the effectiveness of natural language style programming languages. [31] investigated the performance of first time programmers using one of three languages: Quorum, Perl, and Randomo. Quorum is a programming language that has been designed with heavy emphasis on adhering to programming language usability research, including a focus on using intuitive programming language constructs. Perl is a traditional, multi-paradigm programming language that uses constructs such as `for` loops. Randomo is a "placebo-language" where some of the syntax was chosen by generating random characters. Interestingly, when comparing the performance of novices using these different languages, the programmers using Quorum were significantly more accurate at their tasks than the other two languages and the programmers using Perl were not significantly more accurate than the Randomo programmers. An interesting consideration from the study design is that programmers were given reference sheets and code samples for some of their tasks, but they were not specifically taught the meaning of individual lines of code in the examples. Rather than focusing on changing language constructs, ExplainThis seeks to leverage natural language knowledge by explaining syntactic forms in natural language.

Another line of work translates code to natural language. One example of this is automatic code summarization, such as automatically generating method summaries. State-of-the-art research has seen good results using natural language processing techniques, combining information from the code and the abstract syntax trees of the code to generate summaries of methods [12]. However, recent work studied the effectiveness of such automatically generated summaries for programmer comprehension and found that traditional measures of summary quality did not correlate strongly with programmer comprehension, indicating a need to evaluate the effectiveness of such summaries differently [30]. Another line of research focuses on verbalization of code, rather than summarization. This has been done for proofs generated by proof assistants such as Nuprl [9] and Coq [1]. Similarly, work has gone into automatically generating context-aware "lexical simplifications," which are substitutes for technical terms in scientific text to terms that are understood by a wider audience [10]. ExplainThis does not attempt to summarize blocks of text. Rather, it performs a function closer to verbalizing chunks of code, though with additional emphasis on being explanatory and instructive.

Additional work has looked at integrating natural language processing techniques into programming environments. CodeMend [27] is an integrated system that provides suggestions, relevant syntactic forms, and documentation based on the body of a program where the user intends to insert new code. The user has the option to further refine the results by providing a natural language query. CodeMend can provide recommendations based solely on the user's code. However, its strength comes from its ability to allow users to look up a specific API function using a text search. CodeMend's NLP model is trained to look up API functions in the context of the lines of code surrounding where the cursor is placed and return useful recommendations to the developer, who may not be familiar with a specific API. Future iterations of ExplainThis could include similar search and complete functionalities for holes (incomplete parts) in a program, making use of NLP techniques to improve the amount of contextual information available to leverage.

8 Conclusion

As programmers learn different programming languages, adaptive, contextual documentation provides quick access to the level of information that a programmer needs to be productive. We present an early exploration of the design space of contextually adaptive programming language documentation systems, including a survey of current documentation practices, an early design for a concrete system in Hazel, and a formative evaluation of properties of explanations and examples that experienced programmers view as beneficial. Future directions of these documentation tools include: adaptation to a programmers' knowledge of the language they are currently working in; customization based on other known languages, explaining new concepts in terms of those that are already understood; tailoring information based on the specific use cases of the user, such as different forms of documentation for one-off use versus trying to become an expert; and more through evaluation of the tool in various educational and development contexts.

Acknowledgments

We thank Andrew Blinn and David Moon for their help with implementation work. We thank Eunice Jun and the HSD team at the University of Washington for input on human subject studies. We thank Andrew Blinn and Benjamin Kushigian for piloting our study. Also, we thank Steven L. Tanimoto and the anonymous reviewers for their valuable feedback on earlier versions of this paper.

References

[1] Andrew Bedford. 2017. Coqatoo: Generating Natural Language Versions of Coq Proofs. *CoRR* abs/1712.03894 (2017). arXiv:1712.03894 http://arxiv.org/abs/1712.03894

[2] Amy S. Bruckman and Elizabeth Edwards. 1999. Should we Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language. In *Proceeding of the CHI '99 Conference on Human Factors in Computing Systems: The CHI is the Limit, Pittsburgh, PA, USA, May 15-20, 1999*. ACM. https://doi.org/10.1145/302979.303040

[3] Will Crichton. 2021. A New Medium for Communicating Research on Programming Languages. (2021). https://willcrichton.net/nota/ Accessed on 10.19.2022.

[4] Paul E. Dickson, Neil C. C. Brown, and Brett A. Becker. 2020. Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2020, Trondheim, Norway, June 15-19, 2020*. ACM. https://doi.org/10.1145/3341525.3387404

[5] Philip J. Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *The 44th ACM Technical Symposium on Computer Science Education, SIGCSE 2013, Denver, CO, USA, March 6-9, 2013*. ACM. https://doi.org/10.1145/2445196.2445368

[6] Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, and Jan Vahrenhold. 2019. Notional Machines and Programming Language Semantics in Education (Dagstuhl Seminar 19281). *Dagstuhl Reports* 9, 7 (2019). https://doi.org/10.4230/DagRep.9.7.1

[7] Andrew Head, Codanda Appachu, Marti A. Hearst, and Björn Hartmann. 2015. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2015, Atlanta, GA, USA, October 18-22, 2015*. IEEE Computer Society. https://doi.org/10.1109/VLHCC.2015.7356972

[8] Andrew Head, Kyle Lo, Dongyeop Kang, Raymond Fok, Sam Skjonsberg, Daniel S. Weld, and Marti A. Hearst. 2021. Augmenting Scientific Papers with Just-in-Time, Position-Sensitive Definitions of Terms and Symbols. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, Article 413. https://doi.org/10.1145/3411764.3445648

[9] Amanda M. Holland-Minkley, Regina Barzilay, and Robert L. Constable. 1999. Verbalization of High-Level Formal Proofs. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA*. AAAI Press / The MIT Press. http://www.aaai.org/Library/AAAI/1999/aaai99-041.php

[10] Yea-Seul Kim, Jessica Hullman, Matthew Burgess, and Eytan Adar. 2016. SimpleScience: Lexical Simplification of Scientific Terminology. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*. The Association for Computational Linguistics. https://doi.org/10.18653/v1/d16-1114

[11] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2020, Dunedin, New Zealand, August 10-14, 2020*. IEEE. https://doi.org/10.1109/VL/HCC50065.2020.9127201

[12] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM. https://doi.org/10.1109/ICSE.2019.00087

[13] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail N. Giannakos, Amruth N. Kumar, Linda M. Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2018, Larnaca, Cyprus, July 02-04, 2018*. ACM. https://doi.org/10.1145/3293881.3295779

[14] Briana B. Morrison. 2017. Dual Modality Code Explanations for Novices: Unexpected Results. In *Proceedings of the 2017 ACM Conference on International Computing Education Research, ICER 2017, Tacoma, WA, USA, August 18-20, 2017*. ACM. https://doi.org/10.1145/3105726.3106191

[15] OCaml. 2022. OCaml. https://ocaml.org/ Accessed on 10.21.2022.

[16] Cyrus Omar. 2021. Hazel is a live functional programming environment featuring typed holes. https://hazel.org/

[17] Cyrus Omar and Jonathan Aldrich. 2018. Reasonably programmable literal notation. *Proc. ACM Program. Lang.* 2, ICFP (2018), 106:1–106:32. https://doi.org/10.1145/3236801

[18] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proc. ACM Program.*

Lang. 3, POPL (2019). https://doi.org/10.1145/3290327

[19] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM. https://doi.org/10.1145/3009837.3009900

[20] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.SNAPL.2017.11

[21] Hannah Potter and Cyrus Omar. 2020. Hazel Tutor: Guiding Novices Through Type-Driven Development Strategies. https://hazel.org/hazeltutor-hatra2020.pdf

[22] Pyret. 2022. Pyret. https://www.pyret.org/ Accessed on 10.21.2022.

[23] Yizhou Qian and James Lehman. 2017. Students’ Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1 (2017). https://doi.org/10.1145/3077618

[24] Racket. 2022. Racket. https://racket-lang.org/ Accessed on 10.21.2022.

[25] Adrian E. Raftery, Hana Sevcikova, and Bernard W. Silverman. 2021. The vote Package: Single Transferable Vote and Other Electoral Systems in R. *R J.* 13, 2 (2021). https://doi.org/10.32614/rj-2021-086

[26] Reason. 2022. Reason. https://reasonml.github.io/ Accessed on 10.21.2022.

[27] Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. 2016. CodeMend: Assisting Interactive Programming with Bimodal Embedding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology, UIST 2016, Tokyo, Japan, October 16-19, 2016*. ACM. https://doi.org/10.1145/2984511.2984544

[28] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. 2022. Here we go again: why is it difficult for developers to learn another programming language? *Commun. ACM* 65, 3 (2022). https://doi.org/10.1145/3511062

[29] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*. ACM. https://doi.org/10.1145/2568225.2568252

[30] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A Human Study of Comprehension and Code Summarization. In *ICPC ’20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM. https://doi.org/10.1145/3387904.3389258

[31] Andreas Stefik, Susanna Siebert, Melissa Stefik, and Kim Slattery. 2011. An empirical comparison of the accuracy rates of novices using the quorum, perl, and randomo programming languages. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools, PLATEAU 2011, Portland, OR, USA, October 24, 2011*. ACM. https://doi.org/10.1145/2089155.2089159

[32] Alaaeddin Swidan and Felienne Hermans. 2019. The Effect of Reading Code Aloud on Comprehension: An Empirical Study with School Students. In *Proceedings of the ACM Conference on Global Computing Education, CompEd 2019, Chengdu,Sichuan, China, May 17-19, 2019*. ACM. https://doi.org/10.1145/3300115.3309504

[33] Bret Victor. 2012. Learnable Programming: Designing a programming system for understanding programs. http://worrydream.com/#!/LearnableProgramming

Received 2022-07-12; accepted 2022-10-02