Title

Tutorons: Generating Context-Relevant, On-Demand Explanations and Demonstrations of Online Code

Authors

Head, Andrew
Appachu, Codanda
Hearst, Marti A
et al.

Publication Date

2015

Peer reviewed

Tutorons: Generating Context-Relevant, On-Demand Explanations and Demonstrations of Online Code

Andrew Head, Codanda Appachu, Marti A. Hearst, Björn Hartmann
Computer Science Division
UC Berkeley, Berkeley, CA 94720
{andrewhead, appachu, bjoern}@eecs.berkeley.edu, hearst@berkeley.edu

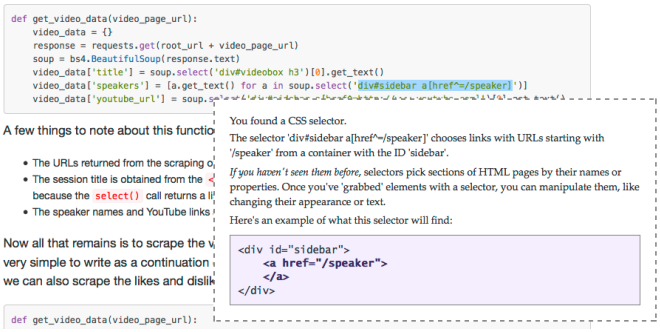
***Abstract*—Programmers frequently turn to the web to solve problems and find example code. For the sake of brevity, the snippets in online instructions often gloss over the syntax of languages like CSS selectors and Unix commands. Programmers must compensate by consulting external documentation. In this paper, we propose language-specific routines called Tutorons that automatically generate *context-relevant, on-demand* micro-explanations of code. A Tutoron detects explainable code in a web page, parses it, and generates in-situ natural language explanations and demonstrations of code. We build Tutorons for CSS selectors, regular expressions, and the Unix command "wget". We demonstrate techniques for generating natural language explanations through template instantiation, synthesizing code demonstrations by parse tree traversal, and building compound explanations of co-occurring options. Through a qualitative study, we show that Tutoron-generated explanations can reduce the need for reference documentation in code modification tasks.**

I. Introduction

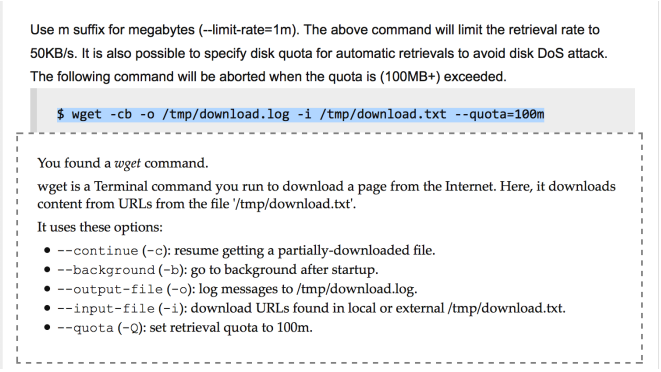
Many programmers develop code by interleaving opportunistic information foraging with writing code [1], [2]. When learning about unfamiliar technologies, programmers often start by searching for tutorial web sites. Instead of examining the prose, they experiment with code examples, using source code from the pages to support learning by doing [1]. However, programmers of all backgrounds struggle to leverage web documentation to solve programming problems [3]–[5].

One source of confusion is that authors write teaching materials with a certain audience in mind. This target audience determines what information is included, and what is omitted. For mixed language tutorials, guidance on the languages 'embedded' within a dominant language may be nonexistent. An author of a web scraping tutorial may describe Python libraries while assuming knowledge of the CSS selectors and regular expressions used as arguments. The author's audience may have different skills. As a result, answers on Q&A sites like StackOverflow can have insufficient explanation [6], and tutorial blogs may lack scaffolding for readers seeking clarification on unfamiliar or complex syntax.

To provide the missing scaffolding for unexplained code in tutorials, Q&As, and other online programming help, we propose routines called *Tutorons* that produce context-relevant, on-demand descriptions of code. Tutorons automatically find pieces of code in web pages and augment these with micro-explanations: such explanations are *context-relevant*, describing only the syntactic elements that are present and important within a snippet, using domain-specific terms. They are also



(a) A micro-explanation of a CSS selector with an automatically generated natural language explanation and demonstration of an HTML element it matches.



(b) A micro-explanation describing the high-level intent and low-level argument values of a wget command.

Fig. 1. Tutoron-generated micro-explanations enable programmers to seek in-situ help when reading and adapting code found in online tutorials and Q&As. Programmers select the code they want clarification about, and view natural language descriptions and demonstrations of what the code does.

on-demand — they can describe confusing code found anywhere to provide just-in-time understanding.

We have built Tutorons to explain single lines of code that frequently occur in command lines and embedded languages like CSS selectors and regular expressions (see Figure 1). In our implementation, each language can be supported in several hundred lines of code. Each one can be built as a standalone server, accessible by an addon in the programmer's browser to provide explanations of code anywhere on the web.

We make three main contributions. First, we introduce a framework for building Tutorons. Second, we describe the processing pipeline, technical considerations, and design

guidelines involved in building Tutorons, providing details about our efforts to implement them for CSS selectors, regular expressions, and Unix commands. Finally, we show through an in-lab study how Tutoron-generated micro-explanations can help programmers modify existing code without referring to external documentation.

II. Definitions

Throughout this paper, we refer to the following terms to describe the system, and its input and output:

A *Tutoron* is a routine on a web server with language-specific rules for detecting, parsing and explaining source code written on a web page.

An *explainable region* is a substring of an online code snippet that a Tutoron can parse and explain. For example, the string `div.klazz` in the JavaScript code `$('div.klazz').text('foo')` is an explainable region for the CSS selector Tutoron.

A *micro-explanation* is the output of a Tutoron for an explainable region of code. This can be a natural language explanation or a domain-specific demonstration of what the code does.

The *Tutorons addon* is a browser extension that queries Tutorons for micro-explanations of code and displays them on-demand for explainable regions. We use the addon to demonstrate the advantages of context-relevant, on-demand in-situ programming help.

III. Related Work

Our work is related to research from two themes. In this paper, we compare Tutorons to past work on the automatic explanation of code and demonstration of code.

A. Automatic Explanation of Code

Most closely related to our work is Webcrystal [7], a tool that assists web authors with low-level programming tasks required to reuse and learn from examples. Webcrystal generates human-readable textual answers to user's "how" questions about how to recreate aspects of selected HTML elements as well as customized code snippets for the users to recreate the design. Similarly to Webcrystal, we generate human-readable representations of code to help programmers reuse and learn from online examples. In contrast to Webcrystal, we focus on describing short, embedded languages like regular expressions and Unix commands instead of HTML, and develop guidelines for generating effective explanations for these languages.

In its technical approach, our work relates to recent efforts to generate natural language explanations of code and software engineering artifacts [8]–[14]. This body of work aims to reduce the cost of searching and understanding code and specifications by automatically summarizing blocks of code [8], class diagrams [9], Java methods [10], unit test cases [11], method context [12], parameters [13], and classes [14]. Similarly to Sridhara et al., we consider how to detect explainable code prior to generating descriptions [8], [10]. However, their code detection process emphasizes summarization of methods and blocks of code rather than locating instances of a language

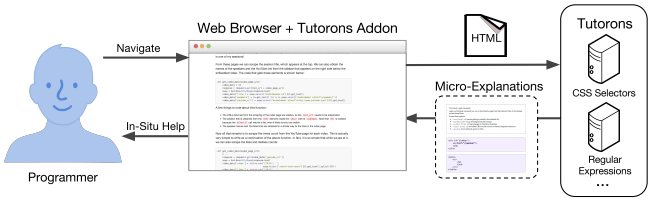


Fig. 2. The Tutorons ecosystem and information flow. A programmer browses for help in a web browser with the Tutorons addon. When they visit a page, the addon queries a bank of Tutoron servers for different programming languages. Each server detects explainable regions of code and produces micro-explanations. The programmer can view these in-situ and on-demand in tooltip overlays by clicking on the explained regions.

in potentially mixed-language code snippets. Our work is also distinguished from this past work by examining several frequently used languages from web-based programming help.

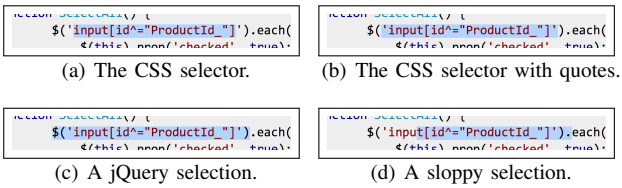
B. Automatic Demonstration of Code

Visual tools have been used to aid in programming instruction; Sorva [15] presents a good review. PythonTutor [16] is a recent programming visualization tool for CS education. The tutor is embeddable within a web page and supports simultaneous viewing of program source code, program execution, visual representations of Python objects, and program output. More recently, Ou et al. produced visualizations of pointer-based data structures in the heap [17]. Beyond visualizations, code can be represented by discrete structures. D'Antoni et al. [18] created counterexamples of system behaviors to provide helpful feedback for computer science students doing automata design problems. We draw inspiration broadly from this past work in visualization and example generation to automatically produce demonstrations of regular expressions and CSS selectors. We believe we are the first to produce guidelines for generating effective micro-explanations of code found in online programming help.

IV. Authors and Readers in the Tutoron Ecosystem

In the current incarnation of Tutorons, an author of programming documentation writes a Tutoron to adaptively describe code that programmers find while they browse the web. A Tutoron is a routine for generating explanations or demonstrations of code for a specific language, accessible as a web API. When queried with the content of a web page, a Tutoron detects explainable regions of the language, parses them, and returns explanations for each region as formatted HTML that can be viewed from a tooltip.

The information flow of the Tutorons ecosystem is shown in Figure 2. By installing an addon for the browser, a programmer receives instant access to micro-explanations, or in-situ descriptions of code, from Tutoron servers. The addon queries existing Tutorons with the page source, receiving micro-explanations for all explainable regions of code. After the addon receives a response, the programmer can click on any explainable string of text to view an explanation in a tooltip (see Figure 1). The addon can query many explanation servers for multi-language support within the same page.



(a) The CSS selector. (b) The CSS selector with quotes.

(c) A jQuery selection. (d) A sloppy selection.

Fig. 3. If a Tutoron does not automatically detect code a programmer wants to have explained, they can request a specific Tutoron to explain it. Tutorons can allow "sloppy" selections by adjusting user-reported regions to ones that can be parsed. For example, our CSS selector Tutoron searches for jQuery strings closest to the user-reported bounds, making the above selections equivalent.

After a page loads in the browser, the addon sends parallel requests to the Tutoron servers to detect all explainable code. Explanations are therefore fetched in one batch when the user first accesses the page. Once the response is received, explanations can be accessed instantaneously by clicking on the explained regions. The original web page is instantly available to the programmer, and micro-explanations for each language become available as each Tutoron processes the document. Computational burden then resides on the server.

The addon colors explainable regions to indicate what code programmers can gain clarification on. If a Tutoron fails to detect code that a programmer wants to have explained, they can select text and pick a Tutoron to generate an explanation from a context menu. Users may select the bounds of an explainable region in a way that the Tutoron doesn't expect, e.g., including the quotation marks on either side of a CSS selector string literal. Tutorons can adjust the boundaries of user-reported regions before parsing, if given sufficient context from the browser addon. As an example, one variant of our CSS selector Tutoron fixes sloppy region bounds for selectors in jQuery strings by searching for the pair of quotation marks nearest to the user's original selection (see Figure 3).

Micro-explanations could also be integrated into individual programming help pages through a Wordpress plugin or Javascript library. We have implemented a Javascript library so any tutorial page can fetch and display explanations from Tutorons regardless of the visitor's browser.

V. How to Build a Tutoron

A Tutoron is a routine that detects, parses, and explains code for a programming language in HTML documents. We describe each processing stage with overarching strategies we have determined for finding relevant code in a page, parsing languages, and generating domain-appropriate explanations. We augment our discussion with implementation details of Tutorons we developed for CSS selectors, regular expressions, and the wget Unix command.

A. Detection

In the *detection* stage, a Tutoron should extract explainable regions from an HTML document using the language's lexicon and / or syntax. This can consist of four steps:

First, the Tutoron extracts blocks of code from HTML pages by selecting code-related elements. Each of our current Tutorons extracts blocks of code from some combination of `<code>`, `<pre>`, `<div>`, and `<p>` elements.

Second, it divides code blocks into candidate explainable regions based on language-dependent rules. CSS selectors and regular expressions can be detected as string literals in parsed Python or JavaScript code, requiring an initial parsing stage to detect these candidate regions. Commands like wget often occupy a line of a script, meaning that candidate regions can be found by splitting code blocks into lines.

Third, it passes candidate regions through a language-specific syntax checker to determine if it follows the grammar. Note that this can be combined with the parsing stage of the Tutoron processing pipeline.

Finally, a Tutoron may reduce false positives by filtering candidates to those containing tokens representative of the language. This is necessary when candidate regions are small and the lexicon and syntax of the language are lenient or broad. While a string like `'my_string'` in a JavaScript program could represent a custom HTML element in a selector, it is more likely a string for some other purpose. Elements in a CSS selector more often than not have tag names defined in the HTML specification (e.g., `p`, `div`, or `img`).

It may happen that the same explainable region is detected by multiple Tutorons. For example, '`div`' could be a CSS selector as well as a regular expression. While in practice the candidate regions of our current Tutorons differ enough that this is not a noticeable problem, we describe approaches to solve this problem in the discussion.

B. Parsing

Detected code snippets are parsed into some data structure in preparation for explanation. We have found two methods of building parsers to be useful. The first method is to introduce hooks into existing parsers to extract the results of parsing. This is useful when it is necessary to recognize a wide range of symbols and their semantics, such as handling the complex interplay of spaces, quotation marks, and redirects involved in parsing Unix commands.

The second method is to develop a custom parser for the language that supports a subset of the language. For CSS selectors, we wrote a 30-line ANTLR[1] parser to gain control over the parse tree structure, allowing us to shape it to how we wanted to generate explanations. The resulting parser integrated nicely into existing code for the selector Tutoron.

C. Explanation

During the final stage, *explanation*, the Tutoron traverses the parse structure to generate explanations and demonstrations of the code. The implementation details of this stage are specific to the parse structure and the desired micro-explanation. In the next section, we describe techniques to provide inspiration for future language-specific micro-explanations.

VI. Designing and Implementing Rich Micro-explanations with Tutorons

We designed and implemented micro-explanations for three languages. For each, we aimed to generate useful explanations for a range of possible code examples without knowing the

[1] www.antlr.org

```
function VISIT_NODE(node)
    if node has id then clause = node.element
        clause += ' with ID node.id'
    else
        clause = pluralize(node.element)
        if node has class then
            clause += ' of class node.class'
        end if
    end if
    if node has child then
        child = visit_node(node.child)
        clause += child + 'from' + clause
    end if
    return clause
end function
```

Fig. 4. A recursive representation of our technique for generating an explanation of a CSS selector from a tree of elements and their attributes.

precise content ahead of time. We discuss techniques for natural language explanations, document generation, template instantiation, and common usage mining to construct micro-explanations for CSS selectors, regular expressions, and wget.

A. CSS Selectors

CSS selectors appear in JavaScript, stylesheets, and web scrapers. Despite their small size, advanced and compound CSS selectors cause errors for programmers working with HTML and CSS [19], making them an appropriate language for developing automatic micro-explanations.

1) Explaining CSS selectors with natural language: Our method of building natural language explanations for CSS selectors involves language generation through parse tree traversal. Our English language explanation technique is implemented as a tree walker that starts at a leaf node and climbs up through its ancestors. The walker first generates a clause at a leaf node that consists of a single noun — a description of the element's tag type in plain English (e.g., 'link' instead of the anchor tag 'a', and 'bolded text' instead of 'strong'). It then appends modifiers to the noun to describe the ID, classes, attributes, and pseudo-selectors for the node which must be present for the node to be selected.

Then the walker ascends. At each ancestor, it forms a phrase that its child selector is picked "from" the parent (see Figure 4). When it reaches the root, the walker completes a sentence describing that the selector shown chooses the child via all of its ancestors. English language is realized using SimpleNLG[2]. We show descriptions we generate in Table I.

2) Generating example documents: Selectors refer to node characteristics and parent-child and sibling relationships manifest in HTML. We synthesize simple documents for which those characteristics and relationships hold.

Our generation technique is implemented as a visitor for the parse tree of a CSS selector. For each node that represents a tag of an HTML element, we use PyQuery[3] to construct an HTML element. The visitor starts at the leaf nodes of the tree. As it ascends, the visitor constructs the document by appending child nodes to their parent. The resulting document is pretty-printed with BeautifulSoup[4] with spaces, newlines, and carets escaped for display in the browser.

[2]https://code.google.com/p/simplenlg/

[3]https://pypi.python.org/pypi/pyquery

[4]http://www.crummy.com/software/BeautifulSoup/

(a) `form.myform`

```
<form class="myform">
</form>
```

(b) `table tr td`

```
<table>
    <tr>
        <td>
        </td>
    </tr>
</table>
```

(c) `#first:hidden`

```
<div id="first">
    <!-- pseudoclass "hidden" -->
</div>
```

Fig. 5. Examples of HTML documents that can be matched by a CSS selector, automatically generated by our CSS selector Tutoron. Note that we bolden the font of the selected node, for example in subplot 5(b), where the selected element is the `td` nested inside a `tr` in a `table`.

To convey that the leaf nodes are the elements the selector chooses (rather than all elements in the generated document), leaf nodes are marked with a `span` to bolden and color the element. If an element has no tag name, we create a generic `div` to represent it. Attributes, IDs, and class names are added to the opening tag of the element. As pseudoclasses like `checked` and `hover` are not visible from the document but are valid components of selectors, we demonstrate their presence as comments within the element that was selected (see Figure 5(c)).

Future iterations of CSS micro-explanations might benefit from rendering the documents as they would appear when viewed in a browser, particularly for graphical elements like tables and form inputs.

B. Regular Expressions

Regular expressions can be difficult for novices and time-consuming for experts to understand. There is a plethora of tools that generate token-by-token explanations of patterns[5], visualize them[6], and help programmers debug them[7]. We discuss techniques to visualize and explain regular expressions by reappropriating third-party tools and generating example strings.

1) Explaining expressions through diagrams via third-party visualizations: In an early prototype, we developed a Tutoron for visualizing regular expressions, leveraging the third-party visualizer *RegExper* (see Figure 6). While the technical implementation details are trivial, we mention it here as an example of how a Tutoron author can leverage demonstrations produced by their peers to create micro-explanations.

2) Generating example strings: Inspired by recent work on building counterexample-guided feedback on automaton design problems [18], we aimed to generate helpful and instructive examples of what a regular expression can match. The explanation stage we have developed shares thematic ties with the CSS selector document generator.

Like the demonstrations for CSS selectors, we choose to generate brief but demonstrative examples, assuming that strings cannot be too long, or else readers will not be able to grasp the intent of the pattern. Given a regular expression, we build a parse tree that consists of branches, repetitions and literals by examining output from Python's built-in regular expression compiler. We generate each sub-pattern marked with a repetition $\star$ or + exactly once to ensure the subpattern is represented in the text without adding too much length to

[5]http://rick.measham.id.au/paste/explain.pl

[6]http://regexper.com/

[7]https://www.debuggex.com/

TABLE I. TEXT GENERATED TO EXPLAIN CSS SELECTORS

Selector	Realized Text
`div.video-summary-data a[hrefˆ=/video]`	The selector 'div.video-summary-data a[hrefˆ=/video]' chooses links with URLs starting with '/video' from containers of class 'video-summary-data'.
`input:focus`	The selector 'input:focus' chooses in-focus inputs.
`p.introduction::text`	The selector 'p.introduction::text' chooses text from paragraphs of class 'introduction'.

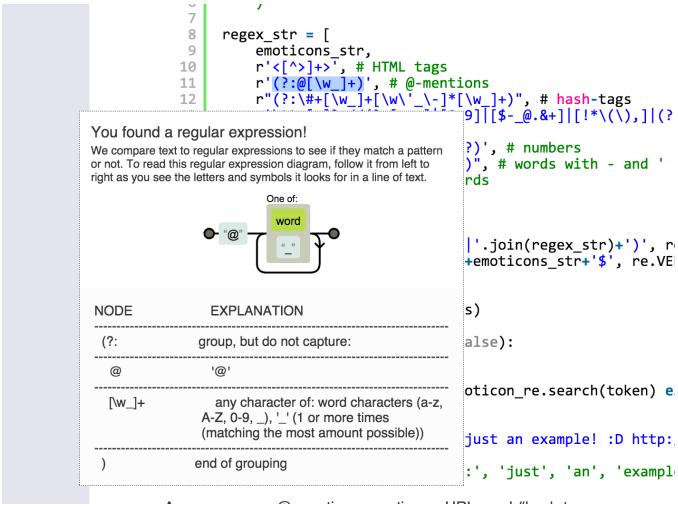


Fig. 6. A micro-explanation of a regular expression that reappropriates content from the third-party tools *RegExper* and *explain.pl* in an in-situ tooltip.

String Type	Regular Expression	Generated Example
MAC addresses	`([0-9A-F]{2}[:]){5}([0-9A-F]{2})`	`D8:E6:0C:ED:0E:79`
US Phone numbers	`ˆ(\d{3})-(\d{3})-(\d{4})$`	`307-039-2703`

Fig. 7. Examples of regular expressions followed by automatically generated example strings our Tutoron builds.

the string. All literals are printed exactly as they appear in the pattern, and we choose a random side of each branch. See Figure 7 for examples of strings we generate to satisfy regular expression patterns.

We take effort to produce readable strings. When we find a repetition of alphanumeric characters, we print an English dictionary word four to six letters in length. If the character class contains symbols or numbers, we insert two random non-alphabetic characters from the class.

C. wget Unix command

As Miller et al. [20] point out, command lines offer efficiency at the price of learning command names and options through up-front effort. We created a Tutoron for the file retrieval Unix command "wget" to show how Tutorons can reduce the effort of recalling the meaning of single-character argument symbols and positional arguments for Unix commands in general. wget has around 100 arguments, making it impractical to remember what each individual one means, particularly with its opaque flags. Utilities like wget already have comprehensive (albeit non-adaptive) documentation accessible through `man` pages and the `-h` flag from the command line. These documents can be used as a starting point for building in-situ micro-explanations.

1) Listing command options and extracting help text from source code: We generate argument-level help by extracting and modifying help phrases for each argument from the wget source code. Two patterns in the help strings allow us to adapt them into more context-relevant descriptions of the arguments: *(a)* Some strings begin with a noun indicating what the value represents, (e.g., "location for temporary files created by the WARC writer.") After detecting a noun at the start of a string using a standard typed dependency parser[8], we add the value to the start of the statement, (e.g., "*my_dir is a* location for temporary…"). *(b)* If the variable name occurs in all capital letters in the help phrase (e.g., "insert STRING into the warcinfo record."), we substitute the value of the argument into the help phrase, replacing the variable's name.

2) Compound explanations of frequent option groups: In many commands including wget, multiple options are frequently used in conjunction with each other to define program behavior. Based on a pilot effort with wget, we propose a way to find common use cases of commands with many options by mining programming help. We scraped 1000 questions from StackOverflow matching the query 'wget'. From these, we extracted 510 lines of code that began with the string 'wget'. Using a custom wget parser from an earlier version of the wget Tutoron, we parsed 346 of these lines without error. For all lines, we then computed the co-occurrence of options, counting how often certain options were used together.

Commonly co-occurring option pairs include `-r` and `-A` (28 counts), `--user` and `--password` (23 counts), and `-r` and `-l` (22 counts). These pairs are indeed semantically meaningful: `-r`, `-l`, and `-A` can be used together to recursively scrape for a certain file type from a site; and `--user` and `--password` are used concurrently to authenticate.

In our wget Tutoron, we create string templates for describing the intent of these frequent groups of options, into which the values of the parameters can be substituted (Table II). To reduce the verbosity of text descriptions, we require that any one option can be described in at most one compound explanation. With this technique, we generate explanations for wget commands like that shown in Figure 8.

D. Design Guidelines for Micro-Explanations

We propose four guidelines to aid in designing micro-explanations for languages in online code examples.

[8]http://nlp.stanford.edu/software/lex-parser.shtml

TABLE II. TEMPLATES FOR DESCRIBING COMBINATIONS OF WGET OPTIONS

Template	Command	Realized Text
Recursively scrape web pages linked from {url} of type '{accept}', following links {level} times.	`wget -l3 -A '*.jpg' site.com`	Recursively scrape web pages linked from http://site.com of type '*.jpg', following links 3 times.
Authenticate at {url} with username '{user}' and password '{password}'.	`wget --user andrew --password mypass site.com`	Authenticate at http://site.com with username 'andrew' and password 'mypass'.

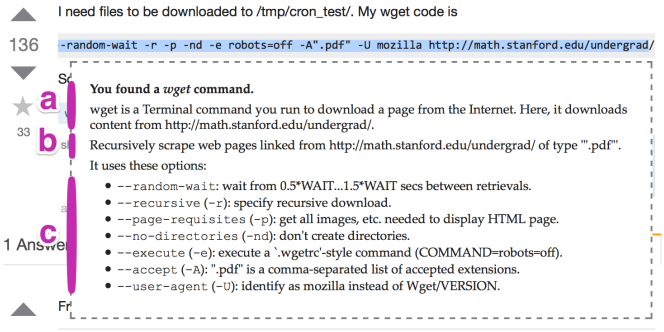


Fig. 8. A multi-level explanation of wget comprises three parts. *(a)* An introduction to the command. *(b)* A meaningful description of what this combination of options does at a high level. *(c)* A review of the meaning and values of each of the options.

1) Leverage multiple representations to illuminate high-level intent and enable low-level understanding of syntax: We explain CSS selectors and regular expressions with text and examples, and wget with high-level and low-level explanations. Such a format supports multiple information needs at once, enabling a viewer to grasp the purpose of the code and engage in fine-grained inspection to prepare for modifying it.

2) Be concise — skip static explanations and focus on dynamically generated content: Our CSS natural language explanations open with one sentence introducing the purpose of CSS selectors. Our usability study revealed that parts of explanations that remain unchanged are likely to be ignored.

3) Reappropriate existing documentation: Explanations for the wget Tutoron were built from help messages extracted from the utility's source code. Programmers working with languages like regular expressions have already built visualization and explanation engines used widely within their communities. The task of the Tutoron author, when good tools and documentation exist, is to locate and build mashups of parsers and explainers built by their peers, and then support in-situ explanations by producing a reliable language detector.

4) Inspect code examples on a large scale to support explanations of common usage: Common usage can be defined in a data-driven way, rather than relying on intuition. By parsing CSS selectors from jQuery code on 50 tutorial pages, we found that 332 / 466 (71.2%) are single tag names, IDs or class selectors. To determine common argument groups that can be described together, detect and parse code examples from online Q&As or web tutorials, as we did for the wget command.

VII. EVALUATION

A. Detection and Parsing Accuracy

While users can explicitly select text that they want to have explained with the Tutorons addon, automatic preprocessing and detection of explainable regions can provide improved information scent. In this section, we assess our Tutorons' accuracy in detecting explainable regions.

We collected a set of online programming help pages that used the languages of each of our Tutorons. To assemble this corpus, we formulated queries for tutorials. Using the API for StackOverflow, a popular programming Q&A site, we extracted the most frequent triplets of tags that included each language's tag. For these triplets, we queried Google with the triplet and the word 'tutorial' (e.g. 'mod-rewrite redirect regex tutorial'), yielding around 200 programming help documents per language. After choosing random sample of 50 documents for each language from these sets, we manually extracted the element and character location of all uses of each language, yielding more than 100 examples per language. We ran the Tutorons on each document and compared the regions the Tutorons found to those we extracted by hand.

With the current Tutorons, we observe 95% precision (129/136) and 64% recall (129/203) for wget, 80% precision (191/238) and 41% recall (191/466) for CSS selectors, and 70% precision (62/88) and 14% recall (62/445) for regular expressions. Our detectors achieve high precision yet low recall. Major reasons for low recall are: *(a)* wget: we fail to reliably detect code examples within sentences of text. *(b)* selectors: we scan only 'pre' and 'code' nodes while skipping 'div' nodes. *(c)* regular expressions: while we wrote rules to reliably detect regular expressions in some contexts (Apache server configs, sed, grep, and Javascript), rules are missing for others (Python, Perl, etc.). We compensate for low recall by extending the browser addon to allow programmers to request explanations for arbitrary text. Recall may be further improved by incorporating a machine learning approach to recognizing commands (e.g., [21]).

We note that we should achieve a 100% success rate parsing valid wget commands and Python flavor regular expressions by reusing existing open source parsers. As we built our own CSS selector parser, we tested its accuracy parsing the ground truth selectors from the detection tests. Of 466 selectors, our Tutoron parsed 440 (94%) without error. Of the selectors that failed to parse, 13 contained invalid characters in the original HTML markup or jQuery pseudo-classes unrecognized in the CSS lexicon. The other 13 illuminated cases our parser cannot yet handle: selectors grouped by comma, double quotes around attribute values, and pseudo-classes that take arguments (e.g., `:nth-child(5)`). Our concise hand-written parser achieves high parsing success in its current form,

and we will revise the selector Tutoron in the near future to properly parse the above failure cases.

B. In-Lab Usability Study

We conducted a usability study to understand how Tutoron-generated micro-explanations affected programmers' ability to perform code modification tasks with online example code. We recruited nine programmers from university listservs for undergraduate and graduate students in computer science and information science. All participants had at least two years of programming experience, with a median of four years of experience in their longest-used language.

Each participant attempted eight code modification tasks (plus two practice tasks) using two different languages: CSS selectors and wget. For each language the four coding tasks increased in difficulty with the fourth being especially tricky.

Each code modification asked a participant to:

1) Read a task. For example: "modify the [wget] command so it does not automatically fetch the images necessary to display the webpage."
2) View a snippet that is unrelated to the specific modification goal, but that contains some clue on how to do it. For example, a StackOverflow post that includes, but doesn't explain, this command:
```
wget --random-wait -r -p -nd -e
    robots=off -A".pdf" -U mozilla
    http://math.stanford.edu/undergrad/
```
3) Write and test code against a condition we provide.

After reading the snippet, participants could use any resources they wanted to, including web search, to complete the task. To make the code examples similar to what a programmer would find when searching for help, we adapted snippets from StackOverflow questions or answers. Tasks asked participants to modify code, as we hoped to find whether the Tutorons provided enough understanding of the code to help programmers perform a practical task.

In one condition, snippets were enhanced with micro-explanations. The explanation did not contain exactly the missing information to complete the task. For example, for wget tasks, participants were shown explanations of all options and asked to discover and modify or remove only those relevant to the task. While these tasks limit the generalizability of our study, we believe that it shows that Tutorons can help in the situations we devised and tested.

Participants viewed micro-explanations for alternating tasks so we could observe differences in how they approached the tasks with and without them; order of exposure to micro-explanations was counter-balanced across participants. When in the micro-explanation condition, participants were asked to find and view all micro-explanations for the source code after reading the snippet.

Participants were told they would have five minutes for each task; if participants had extra time and they had not completed the task, we often asked them to continue so we could observe their problem-solving process. For this study, the browser addon was not equipped with visual indicators of explainable regions as this feature was not yet conceived, although micro-explanations were pre-fetched. We asked participants to "think aloud" as they performed these tasks.

Task	P1	P3	P5	P7	P9
CSS 1					
CSS 2				D	D
CSS 3					
CSS 4		D	D	D	D
wget 1					
wget 2	D	D		D	D
wget 3					
wget 4	D	D	D	D	D

Task	P2	P4	P6	P8
CSS 1				
CSS 2				
CSS 3				D
CSS 4				
wget 1	D	D	D	
wget 2				
wget 3	D	D	D	D
wget 4		D		D

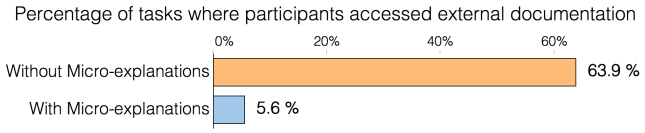


Fig. 9. Tasks for which participants sought additional documentation beyond the snippet. In white rows, participants used micro-explanations; in gray rows, they did not. A cell is marked with the letter 'D' if a participant accessed external documentation during that task. Participants used additional documentation for 63.9% of tasks when micro-explanations were not available, and for 5.6% of tasks when they were available.

C. Results

Our primary goal in observing participants was to determine if Tutoron-generated micro-explanations were helpful during code modification tasks and reduced the need to reference additional documentation. In the discussion below, we refer to our participants as $P1-9$.

1) Micro-explanations help programmers modify code without using other documentation: When using the micro-explanations, 2 out of 36 tasks, or 6% required external documentation. However, for those tasks where micro-explanations were not available, 64% of the time participants did indeed access external resources in order to complete the task. This difference is significant (Fisher's exact test, $p < 0.0001$) The wget tasks especially required external help, with `man` pages and other resources being accessed in 89% of cases. A log of external document accesses is shown in Figure 9.

These preliminary results suggest that the micro-explanations are effective at reducing the need to switch contexts to find relevant programming help for completing some programming tasks. The micro-explanation aided the programmers in several ways:

Reducing the need to access external documentation. Participants were able to identify which flags had to be removed from wget commands without consulting external documentation, despite not having used wget before ($P4$). For some, the micro-explanation affirmed a guess the participant already had about how to construct a CSS selector ($P1$).

Context-relevant pattern matching. One participant noted that the micro-explanations helped to map flags for a wget command to the higher-level intent of the task ($P4$). For the most complex CSS selector task, two participants noted that the example HTML shown in the micro-explanation provided

a pattern of what fields needed to be changed from the selector in the snippet to capture the element and ID required by the task prompt ($P2$, $P4$).

Learning programming concepts. For another participant with little previous experience with CSS selectors, a micro-explanation in the first task provided him with the knowledge needed to write the selector for the next two tasks, one task for which he was not allowed to view micro-explanations ($P5$).

2) Programmers without micro-explanations searched external documentation: Some of the difficulties participants faced in the no-micro-explanation condition highlight the benefits of in-situ help. Some participants had difficulty searching for programming help on a web search engine (Google) and using the search results. One participant could not express the symbols '^=' as a query term when searching for the pattern's meaning for CSS selectors. Her follow-up queries with English language query terms yielded search results that were not relevant ($P3$).

Participants also had difficulty navigating conventional forms of programming help. When looking for the `-r` flag in the `man` page for wget, participants found that a page-internal search for the flag yielded so many matches that it was difficult to find the desired definition ($P2$, $P4$). The description of the `-N` timestamp flag that was relevant to the code modification task was not adjacent to where the flag was introduced, causing one participant to overlook this information ($P3$).

These results underscore the usefulness of context-relevant explanations located within the tutorial text itself.

3) Opportunities for improvement: In those cases where the micro-explanation did not aid programmers, there were a few primary causes:

No visual affordances. For the study, we did not place visible cues showing where explanations were available. Some programmers failed to find micro-explanations since they had to be explicitly invoked by selecting text. One participant ($P2$) had to be reminded that he had to click around the snippet to find micro-explanations for unexplained code when he had not found all explainable regions. Since this study, the browser addon has been extended to provide visual feedback of all regions automatically detected.

Selection region ambiguity. At the time of the study, the algorithm we used to summon an explanation for selected text caused irrelevant explanations to appear when users selected text that was a substring of a detected, explained string. Several participants experienced this problem ($P1$, $P3$, $P5$). With the addition of visual information scent, clickable regions now have exactly one correct explanation to overcome this problem in the default case.

Incomplete explanations. The micro-explanation text may not include enough detail to help programmers develop adequate mental models of unfamiliar material. For instance, even after completing all four CSS selector tasks, $P5$ appeared to believe that CSS selectors were HTML elements themselves, rather than labels that could fetch them, perhaps confused by the example HTML produced in each micro-explanation. In this case, the idea of micro-explanation can be expanded by adding links to fuller tutorial material.

VIII. Discussion

A. Study Caveats

We are encouraged by the results of this preliminary evaluation of the micro-explanation idea of context-sensitive programming explanations. Participants were able to complete the tasks in most cases without requiring other documentation, whereas participants without the explanatory helper text needed to consult external resources. That said, the study had several limitations. It consisted of only a small number of participants, and the tasks were designed such that the Tutorons would function properly.

Furthermore, participants may have been more patient with micro-explanation content than they would have been with documentation they found on their own, especially if they assumed micro-explanations were selected to provide critical insight about the code snippets. In real world scenarios, programmers may not take the same care in reading the micro-explanation text as they did in the study.

Finally, tasks in the study involved edit or deletion tasks with existing code. Tutorons will have to be augmented before they can provide insights for additive tasks and suggest relevant code options that are not in the code itself.

B. Future Work

When Tutorons are available for many languages, disambiguation between different Tutorons will become necessary as multiple detectors can match a given string. We see two potential avenues: First, users could explicitly restrict which Tutorons they wish to activate on a given page. Second, explanations could be ranked by the likeliness that an explainable region belongs to each detected language. We are currently investigating the use of both rule-based and learned weights computed from the frequency of characteristic node types in parse trees in order to generate these likeliness ratings.

We are also interested in conducting a study with novice programmers. We believe that observing the problem-solving strategies of novice programmers will yield better recommendations for how to design in-situ help for this group.

IX. Conclusions

Programmers frequently turn to the web to solve coding problems and to find example code. Tutorons produce explanations for unexplained code that programmers find on the web. In our framework, Tutorons can produce effective micro-explanations when they leverage multiple representations, focus on dynamically generated content, build on existing documentation, and address common usage. We show that Tutorons for a collection of languages achieve around 80% precision detecting code examples, and improve programmers' ability to modify online code examples without referencing external documentation. Tutorons bring context-relevant, on-demand documentation to the code examples programmers encounter in online programming documentation.

Acknowledgments

This work was supported by NSF IIS 1149799.

REFERENCES

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1589–1598.

[2] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 513–522.

[3] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," in *Proceedings of the 2012 International Conference on Software Engineering*, pp. 266–276.

[4] B. Dorn, A. Stankiewicz, and C. Roggi, "Lost while searching: Difficulties in information seeking among end-user programmers," *Proceedings of the American Society for Information Science and Technology*, vol. 50, no. 1, pp. 1–10, 2013.

[5] B. Dorn and M. Guzdial, "Learning on the job: characterizing the programming knowledge and learning strategies of web designers," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 703–712.

[6] S. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming Q&A in StackOverflow," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 25–34.

[7] K. S.-P. Chang and B. A. Myers, "WebCrystal: understanding and reusing examples in web authoring," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3205–3214.

[8] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 101–110.

[9] H. Burden and R. Heldal, "Natural language generation from class diagrams," in *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, pp. 8:1–8:8.

[10] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 43–52.

[11] M. Kamimura and G. Murphy, "Towards generating human-oriented summaries of unit test cases," in *2013 IEEE 21st International Conference on Program Comprehension (ICPC)*, pp. 215–218.

[12] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 279–290.

[13] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *2011 IEEE 19th International Conference on Program Comprehension (ICPC)*, pp. 71–80.

[14] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *2013 IEEE 21st International Conference on Program Comprehension (ICPC)*, pp. 23–32.

[15] J. Sorva, "Visual program simulation in introductory programming education," Ph.D. dissertation, Aalto University, 2012.

[16] P. J. Guo, "Online python tutor: embeddable web-based program visualization for cs education," in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, pp. 579–584.

[17] J. Ou, M. Vechev, and O. Hilliges, "An interactive system for data structure development," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pp. 3053–3062.

[18] L. D'Antoni, D. Kini, R. Alur, S. Gulwani, M. Viswanathan, and B. Hartmann, "How can automatic feedback help students construct automata?" *ACM Trans. Comput.-Hum. Interact.*, vol. 22, no. 2, pp. 9:1–9:24, Mar. 2015.

[19] T. H. Park, A. Saxena, S. Jagannath, S. Wiedenbeck, and A. Forte, "Towards a taxonomy of errors in HTML and CSS," in *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, pp. 75–82.

[20] R. C. Miller, V. H. Chou, M. Bernstein, G. Little, M. Van Kleek, D. Karger, and others, "Inky: a sloppy command line for the web with rich visual feedback," in *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, pp. 131–140.

[21] A. Pavel, F. Berthouzoz, B. Hartmann, and M. Agrawala, "Browsing and analyzing the command-level structure of large collections of image manipulation tutorials," Citeseer, Tech. Rep., 2013.