Many new programmers find it difficult to retrieve relevant results from online documentation due to their current knowledge gaps about the code they are writing. Many online resources do not employ the usage of examples which prevents these programmers from utilizing such resources effectively. Combined with the ambiguous language of many programming languages, this results in a much steeper difficulty curve for many novice programmers trying to enter the field (Dorn, 2013). Thus, this paper's goal is to examine a recent development in CS education, Tutorons, which seeks to bridge the knowledge gap by providing relevant and domain-specific descriptions of a provided code snippet.

Tutorons work through three stages when given an input HTML document to process. The first stage is *detection* where it identifies and scrapes code blocks from the HTML page which are often enclosed with elements such as `<code>`, `<pre>`, and `<div>` (Head, 2015). It then needs to divide the code blocks into recognizable regions. For instance, if a Tutoron wants to determine which part of a code block contains a CSS selector or regular expression, it can do so by detecting string literals in the code. Additionally, the tutoron then performs tokenization where the candidate regions are split into smaller “tokens” or lines of text. The purpose of this is to filter out regions that do not contain tokens representative of the programming language. For instance, the text “my_string” would be discarded as it is too broad to reliably link it to an actual HTML element in a CSS selector (Head, 2015).

Next, the code snippets are then parsed into a modified type of the Abstract syntax tree (AST) data structure. ASTs differ from a regular parse tree in that they will strip away all syntactic details (commas, parentheses, semicolons, etc.) to ensure that only the essential content remains to analyze (Joshi, 2017). Building the AST directly can be difficult though since the parser has to not only find the tokens and represent them correctly, but it also has to decide which tokens are relevant or irrelevant (Joshi, 2017).

Thus, Tutorons uses the ANTLR library which works by taking in a custom grammar file that contains the rules of the desired language to be analyzed. ANTLR then outputs two files: a lexer and parser. The lexer is the tokenizer that splits the input into individual pieces. The parser takes the stream of tokens and produces the subsequent parse tree that can then be traversed (Tomassetti, 2022). Finally, the final stage in Tutorons is the *explanation* step. Since the parser generated an abstract parse tree, the Tutoron traverses through each node while checking the custom rules that were written based on the current value of the node. For instance, if the current node is the `wget` string, then ANTLR will invoke the corresponding function that returns the micro explanation for that UNIX command (Kundel, 2020). For snippets that require the full

context such as CSS selectors, a node's parent will also be referenced as well. To elaborate, the traversal algorithm will first start a child node and gradually climb up to its parent node. The algorithm first generates a template clause at any leaf nodes that consist of a single noun. It then adds modifiers to the noun that refer to the ID, attributes, and classes needed for the node to be selected. Then as the algorithm ascends the tree, it forms a complete phrase with the structure of "the child selector [modifier phrase] from the parent node" (Head, 2015).

When designing explanations for programming languages in online code examples, Tutorons also employs the use of four guiding principles in order to maintain consistency and readability. First is to leverage multiple forms of data such as images in order to support multiple information needs. Most developers prefer example-oriented learning and a tutoron needs to be able to support that sort of system. Secondly is that explanations need to be dynamically generated. By specifically extracting the elements in a CSS selector, the explanations are much more meaningful because they are context-specific. Third is that tutorons should utilize existing documentation in order to broaden its accessibility. A specific example would be the implementation of a built in visualizer for regular expressions where many of these tools can already be found online. Finally, a tutoron for a specific language should be able to automatically expand its training library by scraping online resources. In the current implementation, a tutoron will call the StackOverflow API endpoint, "/tags/related" which returns the tags that are most related to the current tag. So for an input of "python", the endpoint would return the top related tags which would include: pandas, numpy, django, etc. The tutoron would then query Google with these tags while appending the word "tutorial" (EX: "python pandas tutorial"). From there, it takes in a random sample of HTML documents where it extracts any code blocks to add as a recognizable region for future suggestions. This method achieves high precision (95% precision for "wget" command), but has low recall (only 64%) as the authors state that tutorons still currently fail to reliably detect code examples that are enclosed within sentences of text (Head, 2015). The authors propose one way to compensate for low recall by allowing programmers to request explanations for text or leveraging a machine learning solution to recognize commands as well. (Head, 2015)

Overall, Tutorons has potential based on its current success in providing explanations for CSS selectors or regular expressions. With that said, the biggest challenge that is preventing Tutorons from expanding to more programming languages is the problem where one explainable region may be too ambiguous to narrow down which language it corresponds to. For instance, there could be expressions in a code block that can be valid to either a Python or

JavaScript tutoron. For future work, the authors believe that this problem can be addressed through two approaches. One is to allow users to restrict which programming languages should be searched for on a given page. The second is to utilize rule-based and learned weights that are computed from the frequency of node types in the ANTLR parse trees to generate a likelihood ranking that a code region belongs to a detected language (Head, 2015). With more work done for the Tutorons system, there is a significant possibility that it could be integrated into IDEs to provide in-line explanations and further boost developer productivity in the future.

References

Dorn, Brian, et al. “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, 2013, pp. 1–10., https://doi.org/10.1002/meet.14505001059.

Head, Andrew, et al. “Tutorons: Generating Context-Relevant, on-Demand Explanations and Demonstrations of Online Code.” *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2015, https://doi.org/10.1109/vlhcc.2015.7356972.

Joshi, Vaidehi. “Leveling up One's Parsing Game with ASTs.” *Medium*, Basecs, 5 Dec. 2017, https://medium.com/basecs/leveling-up-ones-parsing-game-with-asts-d7a6fc2400ff.

Kundel, Dominik. “Introduction to Abstract Syntax Trees.” *Twilio Blog*, Twilio, 11 June 2020, https://www.twilio.com/blog/abstract-syntax-trees.

Tomassetti, Gabriele. “The Antlr Mega Tutorial.” *Strumenta*, 23 Mar. 2022, https://tomassetti.me/antlr-mega-tutorial/.