Reading Hierarchies in Code: Assessment of a Basic Computational Skill

Thomas H. Park, Meen Chul Kim, Sukrit Chhabra, Brian Lee, Andrea Forte
College of Computing and Informatics
Drexel University
Philadelphia, PA, USA
{park, meenchul.kim, sukrit.chhabra, bl389, aforte}@drexel.edu

ABSTRACT
One of the skills that comprise computational thinking is the ability to read code and reason about the hierarchical relationships between different blocks, expressions, elements, or other types of nodes, depending on the language. In this study, we present three new instruments for assessing different aspects of reading hierarchies in code, including vocabulary, reasoning, and fluency. One of these instruments is Nester, an interactive tool we have designed to elicit mental models about the hierarchical structure of code in computing languages ranging from HTML, CSS, and LaTeX to JavaScript and Lisp. We describe a lab study in which we administered these instruments to 24 participants with varying degrees of web development experience. We report findings from this study, including participants' ability to define, reason about, and manipulate hierarchies in code, and the errors and misconceptions that relate to them. Finally, we discuss avenues for future work.

CCS Concepts
Social and professional topics → Computational thinking • Social and professional topics → Assessment

Keywords
computational thinking; web development; assessment; program comprehension.

1. INTRODUCTION
Computation is increasingly recognized as a fundamental literacy alongside reading, writing, and arithmetic [2]. The term computational thinking was coined by Wing to describe "solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science" [30]. In their work with Scratch, Brennan and Resnick expand on this notion by proposing a framework of concepts (*e.g.*, iteration, parallelism), practices (*e.g.*, debugging, remixing), and perspectives (*e.g.*, expressing, connecting) [1]. As their framework demonstrates, computational thinking can be thought of as a rich, multi-layered set of knowledge and skills.

ITiCSE '16, July 09 - 13, 2016, Arequipa, Peru

ACM 978-1-4503-4231-5/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2899415.2899435

Web development is one domain that reflects this rich and multi-layered quality, given the broad set of technologies and practices that it calls upon, as well as its role as the first major exposure to creative computation for many people [3, 22]. Multiple facets of web development have been examined through the lens of computational thinking. For example, Miller et al. [14] analyze the errors students make when using a tree representation of a filesystem to construct relative and absolute paths. Dorn and Guzdial [4] characterize the knowledge gained by web developers about fundamental programming concepts like assignment, scope, and recursion through their experience with languages like JavaScript and PHP, finding that participants often recognize but do not fully understand these concepts.

Previously we proposed that basic markup and style-sheet languages like HTML and CSS can also engage aspects of computational thinking, like notation, nesting, and parameters [15]. Our subsequent work has explored the common errors web developers make when writing HTML and CSS [17], identifying the deep nesting of code as a particularly troublesome area for beginners [16].

In this paper, we build on that work by focusing our attention on the knowledge and skills associated with reading deeply nested hierarchies in code. Navigating hierarchies in code involves multiple perceptual, cognitive, and motor processes, from parsing long strings of text by identifying delimiters and other features of the code, forming mental models that reflect the code's hierarchical structure, reasoning about relationships between different sections of the code based on this model, and editing the code to reflect a newly desired state.

Like many concepts found within computational thinking, hierarchies provide a way of managing complexity. By being able to move adroitly between different levels of nesting, one can view the same code at different levels of abstraction, focusing attention on a particular level or chunk of code while understanding its function in relation to the rest of the program. Moreover, this is a basic concept that can be applied to a wide range of computing languages. However, these abilities presumably must be learned over time and developed through practice.

To explore knowledge and skills related to navigating hierarchies in code, we pose the following research questions:

1. How familiar are web developers with the vocabulary of hierarchies?
2. How well do web developers apply rules and reason about navigating hierarchies?
3. How can we measure skills associated with fluently navigating hierarchies, and how do they transfer from familiar to unfamiliar computing languages?

2. RELATED WORK

The program comprehension literature describes several models of how programmers construct an understanding of code [23]. In early work, two fundamental approaches were proposed. The top-down model suggested that programmers start with the problem domain and program goals, relating them to elements of the code [25]. In the bottom-up model, programmers begin with elements of the code and map them to the program goals [19]. Since then, other models depict a more nuanced picture where programmers switch opportunistically between these approaches [11], depending on their present state of knowledge about the program and the programming task at hand [13].

During program comprehension, code can be read along multiple dimensions, with different features of the program becoming more relevant and providing alternate forms of information [7]. These dimensions facilitate different strategies that professional programmers use to understand code, like tracing the data flow and the control flow [10, 21]. Another dimension is spatial, referring to the code's order in the source, in contrast to its execution order [8]. In the present study, we consider a structural view in which programmers examine the hierarchical structure that is created by the nesting of code blocks in a program. This structural view is tied to both the logic underpinning the code's organization, as well as its spatial arrangement when formatted with indentation.

In expert program comprehension, researchers have found evidence for a reliance on beacons, which Wiedenbeck defines as "lines of code which serve as typical indicators of a particular structure or operation" [29]. Green has likewise found support for the indentation of nested code enabled by structured programming for acting as "redundant perceptual encoding… [that] provides a secondary clue to their logical structure" in improving program comprehension [6]. These empirical findings suggest that just as individual lines can serve as landmarks, reading hierarchies can help reveal the more general terrain of code during program comprehension.

Finally, several studies point to the design and practices of a language, such as its nesting syntax and use of whitespace, as having an effect on program comprehension. Like paired tags used in HTML and XML, the redundant labelling of "begin" and "end" statements used to mark a nested block of code in ALGOL was found to aid program comprehension among novices [24]. Stefik and Seibert conducted several studies that found the syntax of different program languages like Python, Ruby, and Java to have a significant effect on their understandability and ease of use [27]. If significant differences exist among imperative programming languages, then for drastically different languages like LaTeX, CSS, and Lisp, there is the question of the degree to which such differences exist, and how the basic skill of reading hierarchies in code may transfer across languages.

Despite distinct differences from reading code, the extensive research on reading comprehension as it relates to prose is also instructive [19]. Reading fluency—decoding and comprehending text accurately and with the appropriate rate and prosody—calls on a cascade of sub-processes and knowledge such as letter sound fluency and vocabulary [9]. Studies have found that lower-level skills like word decoding are a critical factor in reading ability [5]. We similarly consider reading hierarchies in code as a basic computational skill that can support higher-level programming activities.

3. METHODS

To address our research questions, we developed three instruments for measuring knowledge and skills about hierarchies in code. We then conducted a lab study in which we invited participants into the lab and had them complete tasks based on these instruments. We triangulate the data collected from these instruments for our analysis.

3.1 Data Collection

For our study, we sought participants who had prior experience in web development, ranging from beginner to expert. We posted on-campus flyers and announcements on mailing lists, and offered $25 for participation in the approximately hour-long session.

The participants were invited into a lab, where we provided them with a computer to be used for the study. After giving informed consent, participants were directed to complete a computer-based pre-questionnaire that collected information about their area of study and self-reported expertise with a variety of computing languages. Following this, they completed tasks based on the three instruments described in the next section. The study concluded with a post-questionnaire that asked participants to rate their perceived difficulty with the tasks, and provide demographic information such as age and gender. We delayed collection of demographic information until after the tasks in order to minimize the effect of stereotype threat [26].

24 participants volunteered to participate, 11 females and 13 males. All participants were either undergraduate or graduate students, with an average age of 22 years. Their areas of study emphasized design or technology, including majors in digital media, software engineering, computer science, and information systems. The sole exception was a chemical engineering student.

None of the participants practiced web development professionally, but all had some level of prior experience. Among the computing languages surveyed, participants were most familiar with HTML, followed by CSS and JavaScript. All 24 participants reported experience with HTML, with a mean of 2.1 on a scale 0 (no experience) to 4 (expert), 22 participants with CSS (μ = 1.7), and 19 with JavaScript (μ = 1.3).

3.2 Instruments

In this section, we present three instruments we have developed to assess knowledge and skills related to the concept of hierarchies in code. These include hierarchical vocabulary, hierarchical reasoning, and hierarchical fluency.

3.2.1 Hierarchical Vocabulary

The first instrument assesses basic understanding of vocabulary associated with hierarchies. Seven terms are presented:

- Parent
- Child
- Ancestor
- Descendant
- Sibling
- Root
- Leaf

For each term, participants are first asked to rate whether they know the definition in the context of hierarchies, recognize the term but do not know the definition, or are not familiar with the term. Next, they are prompted to define each of the terms in their own words.

3.2.2 *Hierarchical Reasoning*

In the second instrument, participants are presented with a code sample (Figure 1) and 14 items asking them to identify various nodes based on combinations of the aforementioned terms. For example, one item asks participants to identify the parent of the node present in line 2. This type of reasoning is frequently employed in activities like debugging HTML code, or navigating and selecting DOM nodes using CSS or JavaScript. The instrument provides definitions for the terms in case participants are not already familiar with them, and asks them to identify nodes by providing their element names and line numbers (*e.g.*, html in line 1).

The complete list of items is presented in Table 1. The first seven items apply the hierarchical terms individually, while the rest relate to more complex scenarios involving what we predict to be common misconceptions or pitfalls when reasoning about hierarchies. For example, item 10 assesses whether participants recognize nested nodes when formatted inline; Item 11 assesses whether participants mistake "cousins" for "siblings" when the cousins have no siblings of their own. The code sample is designed to support all of these items.

This instrument takes an approach that is similar to code tracing problems [12] in that participants must reason about a static representation of code. However, rather than predicting the program execution and output, they must determine hierarchical relationships between different parts of code.

```
1  <html>
2    <head>
3      <title>Thingamabobs</title>
4    </head>
5    <body>
6      <header>
7        <h1>Thingamabobs</h1>
8        <h2>For Sale</h2>
9        <p>Starting $99</p>
10     </header>
11
12     <table>
13       <tr>
14         <th>Price</th>
15       </tr>
16       <tr>
17         <td>$99</td>
18       </tr>
19     </table>
20
21     <ul>
22       <li>Easy to use</li>
23       <li>Affordable</li>
24       <li>Compatible with
25         <ul>
26           <li>Gizmos</li>
27           <li>Doohickeys</li>
28         </ul>
29       </li>
30       <li>Lightweight</li>
31     </ul>
32
33     <form>
34       <fieldset>
35         <label>Email address</label>
36         <input type="email">
37         <label>Password</label>
38         <input type="password">
39         <div>
40           <label>
41             <input type="checkbox"> Remember me
42           </label>
43           <button type="submit">Buy</button>
44         </div>
45       </fieldset>
46     </form>
47
48     <footer>
49       <div><strong>©2015 <em>Thimgamabobs</em></strong></div>
50     </footer>
51   </body>
52 </html>
```

Figure 1. The HTML code sample used in the hierarchical reasoning instrument.

Table 1. Items used in the hierarchical reasoning instrument.

Item	Principle	Instructions
1	Parent	List the parent of <head> (line 2)
2	Child	List all children of <header> (line 6)
3	Ancestor	List all ancestors of <h1> (line 7)
4	Descendant	List all descendants of <table> (line 12)
5	Sibling	List all siblings of <h1> (line 7)
6	Root	List the root element
7	Leaf	List all leaf elements contained in <table> (line 12)
8	Proximity	List the parent element of <button> (line 43)
9	Depth	List all leaf elements contained in (line 21)
10	Inline	List the parent element of (line 49)
11	Cousins	List all sibling elements of <td> (line 17)
12	Filter	List all <input> elements that are descendants of any
13	Compound	List all descendants of <body> (line 5) that are also ancestors of <td> (line 17)
14	Common Ancestor	List the closest ancestor that <input> (line 38) and <input> (line 41) share in common

3.2.3 *Hierarchical Fluency*

The third instrument assesses hierarchical rules as applied within an interactive coding environment. We developed a tool called Nester (Figure 2) that presents participants with unformatted snippets of code in various computing languages, with each line of code represented as a movable block. Participants are asked to indent the lines of code to reflect the nesting rules of the language, whether or not whitespace is significant in the given language. The purpose of Nester is to elicit observable representations of mental models that participants hold about the code.

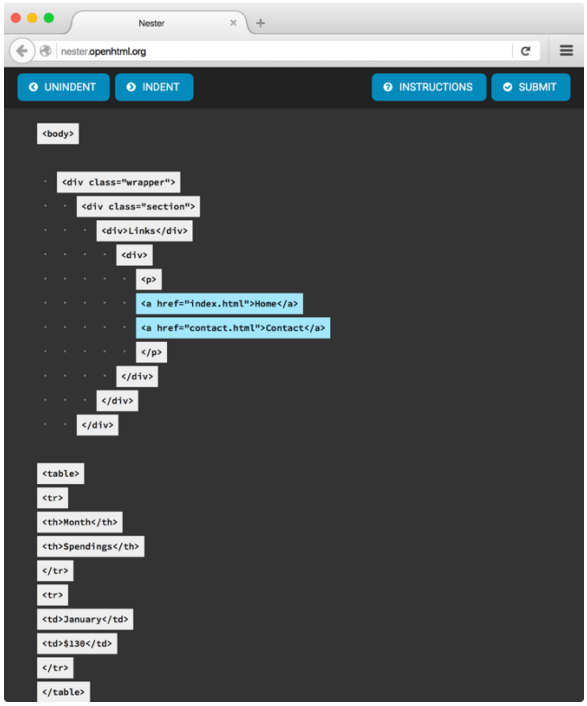

Figure 2. Nester, the interactive tool developed as the hierarchical fluency instrument, loaded with the HTML task.

In our previous work with HTML and CSS [17], we applied the skills-rules-knowledge framework [20] to differentiate rule-based errors that are rooted in misconceptions from skill-based errors that arise even when no misconceptions are held. While the previous two instruments are aimed at uncovering misconceptions or knowledge gaps, this instrument is designed to present more complex and interactive tasks that place a greater load on participants' working memory [28]. In short, it is designed to expose the fluency with which participants handle hierarchies in code.

Compared to a traditional code editor, Nester constrains the possible operations on code. Only the indentation of individual lines of code, not their contents, can be edited. Some programming environments offer auto-indent features that would render Nester's tasks trivial. However, although Nester does not provide an authentic experience that programmers would encounter in practice, like Parson's programming puzzles [18] they can be used to target assessment at a specific aspect of code comprehension.

The common operations for selecting, indenting, and unindenting lines of code are supported via key combinations and the graphical user interface. For example, the up and down arrow keys can be used to select different lines, and a selected line can be indented by using the tab or right arrow keys. Lines can also be selected by mouse by clicking specific blocks, and indent and unindent buttons are displayed prominently at the top of Nester. Multiple lines can also be added to a selection by holding down the shift or control modifier keys while selecting a new line, or by dragging a box with the mouse cursor (*i.e.*, lassoing) around multiple blocks. Before the tasks, participants were given a freeform orientation task to familiarize themselves with the operations of Nester.

Nester is designed to support any number of languages and code samples, but for this study, we used Nester to present a single code sample for each of seven languages: HTML, XML, JSON, LaTeX, SCSS, JavaScript, and Lisp. These languages were selected to represent a broad range of syntaxes for delimiting nested blocks of code. For example, HTML, XML, and LaTeX as markup languages rely on start and end tags, JSON, JavaScript, and SCSS (an extension of CSS) on braces, and Lisp on parentheses. Additionally, we expected the languages to vary in terms of their familiarity among participants. This was confirmed in the pre-questionnaire where all 24 participants had prior exposure to HTML, but only 4 had exposure to Lisp.

The code samples used in the study were comprised of a root node and three sub-trees, each of which was designed to be isomorphic across languages in terms of their nested structure. This was to control for variables like lines of code and complexity in the code samples. For example, if an element in the HTML sample had two child elements, then the equivalent node in JavaScript had two statements in a nested block of code. The order of the sub-trees within a code sample was randomized for each task, and the order of the languages was randomized for each participant.

At the start of each task, instructions and an example are presented within Nester for how code should be formatted in case the participant was not familiar with the language. The instructions can also be invoked later on by clicking a button in the toolbar. After formatting the code, the participant was directed to click the Submit button, where Nester reports whether the task was solved correctly. If errors were present in the code, the opportunity is given to fix the errors and resubmit.

3.3 Data Analysis

For the hierarchical vocabulary instrument, self-reported familiarity with each term was converted to 0 (not familiar), 1 (recognize but do not know the definition), or 2 (can define). The definitions that participants provided were also graded as 0 (incorrect), 1 (partially correct), or 2 (correct). Summing these values resulted in a scale from 0 to 14 for each of the two parts.

For the hierarchical reasoning instrument, responses were rated similarly: 0 (incorrect), 1 (partially correct), and 2 (correct). This resulted in a scale from 0 to 28. Mistakes were examined qualitatively in further detail.

For the hierarchical fluency instrument, data logged by Nester for analysis included the total time spent on each task, the number of attempts per task, as well as the number and locations of errors per attempt.

4. FINDINGS

4.1.1 Hierarchical Vocabulary

In the hierarchical vocabulary instrument, participants' self-rated familiarity with the terms ranged the full scale from 0 to 14., with a mean of 10.7 ($\sigma = 3.8$). Five of the 24 participants had a score of 7 or less, indicating a low level of familiarity, with one participant reporting no familiarity at all. A breakdown of the items is provided in Figure 3, revealing that "leaf", "descendant", "ancestor", and "root" were the least familiar.

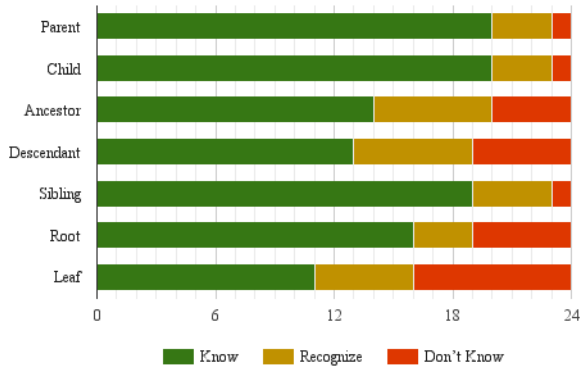

Figure 3. Count of participants' familiarity with terms.

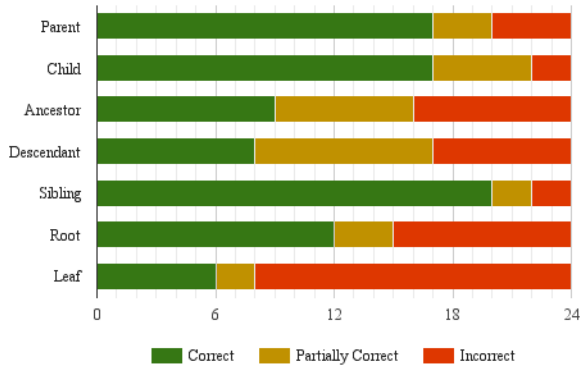

Figure 4. Count of participants' correctness of definitions.

In defining the terms in the participants' own words, scores again ranged from 0 to 14, with a mean of 8.7 (σ = 4.1). Seven participants had a score of 7 or less, with two participants not attempting any definitions. Correctness of each term's definition (Figure 4) was largely in line with the respective self-rating, with a correlation of $r = 0.81$, $p < 0.001$, though participants tended to overestimate their knowledge slightly.

Beyond the implicit metaphor of trees, a number of participants made references to families, object-oriented programming, and file systems in their definitions. Several had fuzzy notions of "leaf", confusing the term with nodes in general. Another common misconception was that "ancestors" excluded "parent" and "descendants" excluded "child" nodes.

4.1.2 Hierarchical Reasoning

For the hierarchical reasoning instrument, scores ranged from 2 to 28, with a mean of 23.1 (σ = 5.9). Performance on the term-centered items in the first half of the reasoning instrument (Table 1) was only moderately correlated with the equivalent items on the vocabulary instrument: $r = 0.72$, $p < 0.001$. However, this performance tended to be higher, suggesting participants were more comfortable applying the terms concretely in the context of code than they were expressing more abstract definitions.

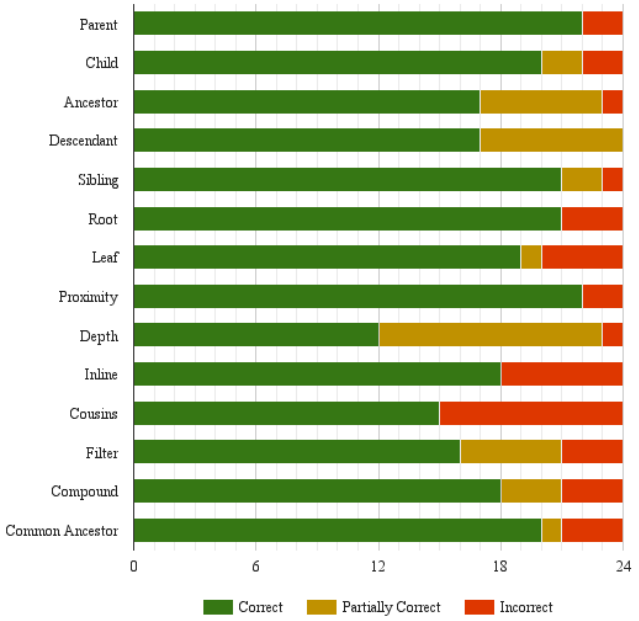

Figure 5. Count of participants' correctness in hierarchical reasoning instrument.

The second half of the instrument revolved around potential pitfalls or misconceptions, and participants did perform more poorly on several items including "depth", "cousins", and "inline". For "depth", a common error was to overlook more shallow leaf nodes when identifying leaf nodes deep within a hierarchy. For "cousins", a cousin node was mistaken for a sibling on several occasions. For "inline", three elements were nested within one another on a single line of code; the correct element, which was located in the middle of the line, was overlooked in favor of the element that was located most prominently at the start of the line.

4.1.3 Hierarchical Fluency

Hierarchical fluency was measured through seven tasks in Nester. Participants varied considerably in time on task, ranging from 6.2 to 64.3 minutes (μ = 16.7, σ = 11.9). The cumulative attempts and errors averaged 12.8 (σ = 4.5) and 4.0 (σ = 10.6) respectively. Strong correlations were found between performance on the reasoning instrument and time ($r = -0.84$, $p < 0.001$) and attempts ($r = -0.70$, $p < 0.001$), but not errors ($r = -0.45$, $p < 0.01$).

The two programming languages Lisp and JavaScript had the greatest variability for time on task, with Lisp taking longest (Figure 6). Attempts (Figure 7) and errors (Figure 8) were highly skewed, with many tasks requiring just one attempt. No correlation was found between the participants' reported expertise with each language and performance on the corresponding task.

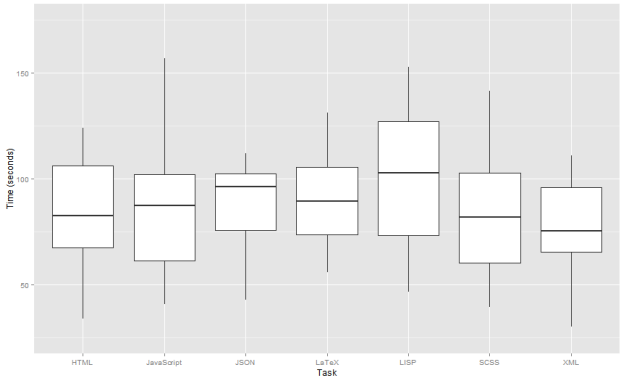

Figure 6. Time on each task.

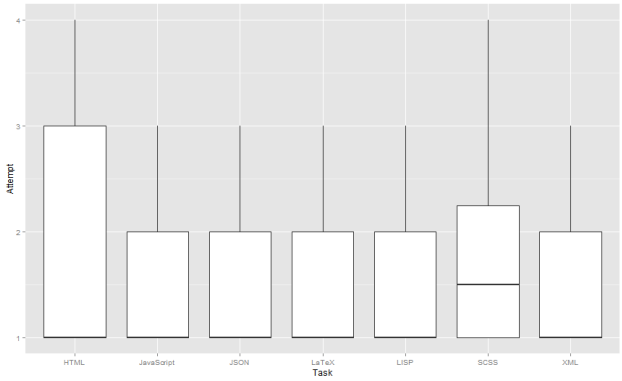

Figure 7. Number of attempts on each task.

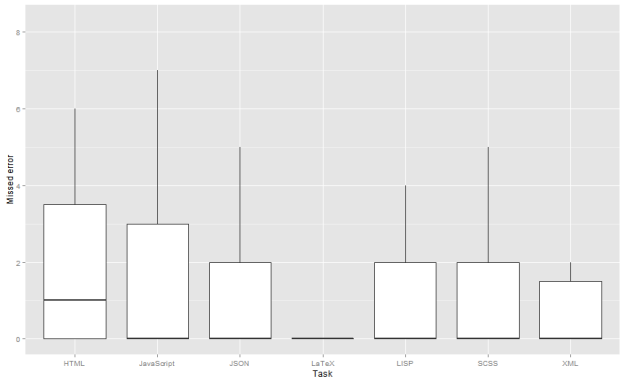

Figure 8. Cumulative number of errors on each task.

5. DISCUSSION

Overall, participants were generally found to be familiar with the vocabulary of hierarchies. However, they were more proficient in reasoning about these terms within the context of code than articulating formal definitions of them.

Contrary to expectations, familiarity with a language did not relate to better performance in the hierarchical fluency instrument. Learning effects with respect to Nester and the tasks themselves may be a factor here. In our analysis, we discovered steep changes in performance during the first three tasks that were administered, independent of language given that task order was randomized, before stabilizing for the rest of the tasks. To address this, a lengthier training session as well as greater quantity and variety in the code samples may be required.

Based on our experiences, we identified several opportunities to refine our instruments. For the vocabulary instrument, we plan to add a prompt for defining the concept of "hierarchy" itself. For the reasoning instrument, identifying nodes by element name and line number was cumbersome; an interactive format that allows nodes to be selected directly would improve usability. Finally, iterations on Nester could add features like syntax highlighting, providing a more familiar and authentic experience.

In future work, we plan to administer these in the context of introductory computing courses, gaining insight into the impact these courses have on student ability to read hierarchies in code.

6. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under CNS grant #1339344.

7. REFERENCES

[1] Brennan, K. and Resnick, M. 2012. New frameworks for studying and assessing the development of computational thinking. *Proc. AERA* (2012), 1–25.

[2] diSessa, A.A. 2001. *Changing Minds: Computers, Learning, and Literacy*. MIT Press.

[3] Dorn, B. and Guzdial, M. 2010. Discovering computing: Perspectives of web designers. *Proc. ICER* (2010), 23–29.

[4] Dorn, B. and Guzdial, M. 2010. Learning on the job: Characterizing the programming knowledge and learning strategies of web designers. *Proc. CHI* (2010), 703–712.

[5] Gough, P.B., Hoover, W.A. and Peterson, C.L. 1996. Some observations on a simple view of reading. *Reading Comprehension Difficulties Processes and Intervention*. C. Cornoldi and J.V. Oakhill, eds. 1–13.

[6] Green, T.R.G. 1977. Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*. 50, 2 (1977), 93–109.

[7] Green, T.R.G. 1990. Programming languages as information structures. *Psychology of Programming*. J.-M. Hoc, T.R.G. Green, R. Samurçay, and D.J. Gilmore, eds. 117–137.

[8] Grigoreanu, V.I., Brundage, J., Bahna, E., Burnett, M.M., ElRif, P. and Snover, J. 2009. Males' and Females' Script Debugging Strategies. *Proc. IS-EUD* (2009), 205–224.

[9] Hudson, R.F., Pullen, P.C., Lane, H.B. and Torgesen, J.K. 2009. The complex nature of reading fluency: A multidimensional view. *Reading & Writing Quarterly*. 25, (2009), 4–32.

[10] Katz, I.R. and Anderson, J.R. 1987. Debugging: An analysis of bug-location strategies. *Human-Computer Interaction*. 3, 4 (1987), 351–399.

[11] Letovsky, S. 1987. Cognitive processes in program comprehension. *Journal of Systems and Software*. 7, 4 (1987), 325–339.

[12] Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. and Thomas, L. 2004. A multi-national study of reading and tracing skills in novice programmers. *ITiCSE-WGR* (2004), 119–150.

[13] Mayrhauser, A.V. and Vans, A.M. 1995. Program comprehension during software maintenance and evolution. *Computer*. 28, 8 (1995), 1–12.

[14] Miller, C.S., Perkovic, L. and Settle, A. 2010. File references, trees, and computational thinking. *Proc. ITiCSE* (2010), 132–136.

[15] Park, T.H. and Wiedenbeck, S. 2011. Learning web development: Challenges at an earlier stage of computing education. *Proc. ICER* (2011), 125–132.

[16] Park, T.H., Dorn, B. and Forte, A. 2015. An analysis of HTML and CSS syntax errors in a web development course. *ACM Transactions on Computing Education*. 15, 1 (Mar. 2015), 1–21.

[17] Park, T.H., Saxena, A., Jagannath, S., Wiedenbeck, S. and Forte, A. 2013. Towards a taxonomy of errors in HTML and CSS. *Proc. ICER* (2013), 75–82.

[18] Parsons, D. and Haden, P. 2006. Parson's Programming Puzzles: A fun and effective learning tool for first programming courses. *Proc. ACE* (2006), 157–163.

[19] Pennington, N. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*. 19, 3 (1987), 295–341.

[20] Rasmussen, J. 1983. Skills, rules, and knowledge; Signals, signs, and symbols, and other distinctions in human performance models. *IEEE Transactions on Systems, Man, and Cybernetics*. 13, 3 (1983), 257–266.

[21] Romero, P., Boulay, du, B., Cox, R., Lutz, R. and Bryant, S. 2007. Debugging strategies and tactics in a multi-representation software environment. *International Journal of Human-Computer Studies*. 65, 12 (Dec. 2007), 992–1009.

[22] Rosson, M.B., Ballin, J.F. and Nash, H. 2004. Everyday programming: Challenges and opportunities for informal web development. *Proc. VL/HCC* (2004), 123–130.

[23] Schulte, C., Clear, T., Taherkhani, A., Busjahn, T. and Paterson, J.H. 2010. An introduction to program comprehension for computer science educators. *ITiCSE-WGR* (2010), 65–86.

[24] Sime, M.E., Green, T.R.G. and Guest, D.J. 1977. Scope marking in computer conditionals: A psychological evaluation. *International Journal of Man-Machine Studies*. (1977), 107–118.

[25] Soloway, E. and Ehrlich, K. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*. 10, 5 (1984), 1–15.

[26] Steele, C.M. 1997. A threat in the air: How stereotypes shape intellectual identity and performance. *American Psychologist*. 52, 6 (1997), 613–629.

[27] Stefik, A. and Siebert, S. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education*. 13, 4 (2013), 1–40.

[28] Sweller, J. 2003. Cognitive load during problem solving: Effects on learning. *Cognitive Science*. 12, (Nov. 2003), 257–285.

[29] Wiedenbeck, S. 1986. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*. 25, 6 (1986), 697–709.

[30] Wing, J.M. 2006. Computational thinking. *Communications of the ACM*. 49, 3 (2006), 33–35.