openHTML: Assessing Barriers and Designing Tools for Learning Web Development

A Thesis

Submitted to the Faculty

of

Drexel University

by

Thomas H. Park

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

December 2014

Acknowledgements

My deepest gratitude to my advisor, Andrea Forte, who has been a source of inspiration and guidance from our very first lunch at the pizza shop years ago. The greatest lessons I've learned from her go beyond research methods and study designs, on how to cross disciplines, bridge communities, and set the conditions to help others succeed.

I would like to thank my committee for their thoughtful feedback and encouragement. To Brian Dorn, who has always been generous with his time and expertise, and with whom I've been fortunate to collaborate on multiple occasions. I have also benefited greatly from the wisdom offered by Mark Guzdial, Greg Hislop, and Jennifer Rode, and the unique perspectives they bring to programming and computing education.

I am grateful to Susan Wiedenbeck and Margaret Burnett, who mentored me in the ways of research during my early years at Drexel, and to the National Science Foundation for their support of this work. Also, a shout-out to all the wonderful designers, developers, and researchers at the Mozilla Foundation.

I would like to acknowledge my fellow students who have accompanied me on this journey. To Warren Allen, Haozhen Zhao, Rachel Magee, Diana Kusunoki, Ankur Saxena, Swathi Jagannath, Heather Willever-Farr, Naz Andalibi, Jill Cao, Stephen Oney, Bernie Zang, and countless others, thank you for the camaraderie and the delightful discussions.

Finally, thanks to my friends and family — most of all, to my mom and dad for their unwavering love, and my wife Amy for supporting me every step of the way.

Table of Contents

List of Tables

List of Figures

Abstract

openHTML: Assessing Barriers and Designing Tools for

Learning Web Development

Thomas H. Park

Andrea Forte, Ph.D.

In this dissertation, I argue that society increasingly recognizes the value of widespread computational literacy and that one of the most common ways that people are exposed to creative computing today is through web development. Prior research has investigated how beginners learn a wide range of programming languages in a variety of domains, from computer science majors taking introductory programming courses to end-user developers maintaining spreadsheets. Yet, surprisingly little is known about the experiences people have learning web development. What barriers do beginners face when authoring their first web pages? What mistakes do they commonly make when writing HTML and CSS? What are the computational skills and concepts with which they engage? How can tools and practices be designed to support these activities?

Through a series of studies, interleaved with the iterative design of an experimental web editor for novices called openHTML, this dissertation aims to fill this gap in the literature and address these questions. In drawing connections between my findings and the existing computing education literature, my goal is to attain a deeper understanding of the skills and concepts at play when beginners learn web development, and to broaden notions about how people can develop computational literacy.

This dissertation makes the following contributions:

- An account of the barriers students face in an introductory web development course, contextualizing difficulties with learning to read and write code within the broad activity of web development.
- The implementation of a web editor called openHTML, which has been designed to support learners by mitigating non-coding aspects of web development so that they can attend to learning HTML and CSS.
- A detailed taxonomy of errors people make when writing HTML and CSS to construct simple web pages, derived from an intention-based analysis.
- A fine-grained analysis of HTML and CSS syntax errors students make in the initial weeks of a web development course, how they resolve them, and the role validation plays in these outcomes.

- Evidence for basic web development as a rich activity involving numerous skills and concepts that can support foundational computational literacy.

Chapter 1
Introduction

As the role of computing in society grows, so grows the importance of a computationally literate citizenry. Just as traditional literacy—that is the fundamental skills needed to read, write, and think critically about written text—has transformed society, giving individuals access to vast sources of information and modes of communication that empower them "to achieve their goals, to develop their knowledge and potential, and to participate fully in their community and wider society", forming a new "basis for positive social transformation, justice, and personal and collective freedom" [UNESCO 2004], computational literacy has the potential to do the same.

Computational literacy is defined as "a socially widespread patterned deployment of skills and capabilities in a context of material support… to achieve valued intellectual ends" [diSessa 2001], using computation as its material basis. diSessa contrasts the abilities needed to create artifacts through computational media such as a programming language with computer literacy's "casual familiarity" with spreadsheets and word processors. Though end-user applications such as word processors and mobile apps can be used to produce expressive artifacts, he stresses that the goal of computational literacy is "not only to control a computational medium, but to create genuinely new

representations", which will have "a penetration and depth of influence comparable to what we have already experienced in coming to achieve a mass, text-based literacy". The alternative threatens to be a monopoly held by "highly trained computing professionals acting as 'high-tech scribes'" [Fischer 2004].

Educational pioneers like Alan Kay [Kay and Goldberg 1977] and Seymour Papert [Papert 1993] have long been inspired by the vision of a world in which every person wields computation as a tool for personal expression and enrichment, civic action, and creativity, making new things humanly possible [Fischer 2004]. Today, this vision is a feature of national policy. The America COMPETES (Creating Opportunities to Meaningfully Promote Excellence in Technology, Education, and Science) Act identified the promotion of technological and scientific literacies among all Americans as a top priority in establishing a more competitive workforce and stimulating U.S. creativity and innovation [Congress 2007].

Despite its promises, we are a long way from achieving widespread computational literacy. A 2010 study by the Association for Computing Machinery (ACM) and Computer Science Teachers Association (CSTA) finds that "paradoxically, as the role and significance of computing has increased in society and the economy, quality computer science education is being pushed out of the K–12 education system in the U.S." The report concludes that K-12 education in most states is "focused almost exclusively on skill-based

aspects of computing... and have few standards on the conceptual aspects of computer science that lay the foundation for innovation and deeper study in the field" [ACM 2010].

At the university level, the rate of students enrolling in computing-related disciplines has not consistently kept pace with projected job growth in these areas [Denning and McGettrick 2005]. Low participation among women and minorities has been a particular source of concern [Camp 1997; Fisher and Margolis 2002]. Retention rates are equally dismal, with up to 40 percent of CS students choosing another major by the end of their first year [Beaubouef and Mason 2005]. After their first year, a significant number of CS students are still unable to write or trace basic programs [McCracken et al. 2001; Lister et al. 2004]. Incoming CS majors often lack an effective model of computers, presenting "a serious obstacle" when learning to program [Ben-Ari 1998].

1.1. Foundations of Computational Literacy

When it comes to traditional literacy, reading attitudes and skills develop even before children are able to make sense of written texts [Holdaway 1979]. Holdaway explains that frequent positive exposures to storybooks in childhood lay the foundation for continued engagement with written texts and the development of increasingly sophisticated literacy skills. In mathematics too, students experience the concept of quantity prior to receiving formal instruction in arithmetic: "they have had to deal with operations of division, addition, subtraction, and determination of size" [Vygotsky 1978].

Likewise, I argue that the road to computational literacy begins long before students take their first programming course. Through a variety of experiences, students learn about the precision required by computers. They are exposed to the ways data is represented and programs are written so that they can be interpreted by computers. They may even author programs themselves, learning to juggle the exacting syntax of formal languages with higher-level concerns about logic and design.

For the most part, these foundational experiences are informal and serendipitous, occurring outside of formal instruction. Lu and Fletcher draw parallels between mathematics and computing education, analogizing that programming is to computer science as proof construction is to mathematics; while primary and secondary education build a foundation of mathematics that leads up to proof construction, such a foundation is absent for college students taking their first programming course [Lu and Fletcher 2009].

Earlier computing experiences can have a substantial impact on students' subsequent perceptions, attitudes, and habits toward computing. For instance, an analysis of the computing biographies of college students [Schulte and Knobelsdorf 2007] found that CS non-majors tend to view computers as a tool for work and leisure, using them for office applications and web surfing. They associate computing-related problems with negative emotions like embarrassment and helplessness. Conversely, CS majors view computers as a

tool they can reshape, and engage them in playful exploration and problem solving. A follow-up study by Ko [Ko 2009] concludes:

> ...No one positive experience with code was enough to keep a person engaged with coding throughout their lifetime; instead, it required persistent, cumulative positive exposure... This suggests that not only will children need positive first encounters with code at a young age, but they will need additional, and different experiences throughout middle school, high school, and college.

1.2. The Case for Basic Web Development

I propose that basic web development, constructing web pages by authoring code in HTML and CSS, can play a pivotal role in developing elementary computational literacy. Basic web development can serve to broaden the diversity of people who engage in computation and deepen their understanding by relating it to everyday experiences with the web.

Web development is a broad term with many meanings. Loosely defined, it is the creation of software for the web, ranging from a single static web page to a complex web-based application, and any related activities that support this endeavor. Web development can involve many activities including client-side and server-side programming, database management, server administration,

graphic design, and content development. Basic web development is but one facet of this activity, but one that is fundamental to building web pages.

While traditional programming languages are a more expressive form of computation and have been the main focus of computational literacy efforts, they are not the only activity that can fill this role. A report by the National Research Council on computer literacy [NRC 1999] notes that literacy curricula have needlessly focused on conventional programming languages; the report goes on to acknowledge activities like the sophisticated use of spreadsheets [Nardi 1993] and even the troubleshooting of technical problems as programming activities. In a similar vein, the mail merge feature of word processors has been used to introduce students to key computational concepts like conditionals and branching [Popyack and Herrmann 1993], while programming has also been investigated in the context of domestic appliances like ovens and video recorders [Rode et al. 2004]. Computer science concepts have even been taught through activities requiring no technology at all [Taub et al. 2009].

In much the same way, basic web development involves many aspects of programming and can provide a contextualized, "low floor" basis for learning about computation. And as a form of programming, even markup languages possess many of its pitfalls: "As with the use of JavaScript, even the abstractions of HTML provide the opportunity for syntax errors, runtime errors, or bugs in the form of unintended or exceptional behaviors" [Blackwell

2002]. Research has also found that novice and intermediate users have "patchy" models of the web [Sheeran et al. 2002], and that students have trouble with hypertext and link creation [Désilets et al. 2005] and composing absolute and relative tree paths when referencing resources like images and web pages [Miller et al. 2010].

Beyond the content of web development, its social significance offers value as a context for learning about computation. Papert coined the term constructionism when arguing that learning "happens especially felicitously in a context where the learner is consciously engaged in constructing a public entity, whether it's a sand castle on the beach or a theory of the universe" [Papert and Harel 1991]. He outlined three design principles that engage newcomers and applied them to the development of Logo, a programming environment that enables students to instruct a "turtle" cursor to draw graphics:

- Continuity: The mathematics must be continuous with well-established personal knowledge from which it can inherit a sense of warmth and value as well as "cognitive competence."
- Power: It must empower the learner to perform personally meaningful projects that could not be done without it.
- Cultural Resonance: The topic must make sense in terms of a larger social context.

The qualities laid out by Papert are embodied in the construction of web pages. First, the web is exceedingly familiar to students, establishing continuity with their existing knowledge. Students learning web development are likely to have had meaningful experiences with the web – as of 2009, 93 percent of Americans aged 12 to 17 and 74 percent of adults have been online [Lenhart et al. 2010]. Second, web development empowers learners with the ability to create of visual and interactive artifacts. Finally, web development holds cultural resonance. Web pages are inherently social, meant to be published online and linked to one another. Already, many people learn web development in formal and informal contexts, including members of groups that are traditionally underrepresented in computer science [Rosson et al. 2004; Dorn and Guzdial 2010a]. Furthermore, the boundaries between learning and practice frequently blur as they learn in pursuit of practical end-goals like making a personal homepage or a website for a small business. Learning can be most effective when situated within authentic practice in this way [Lave and Wenger 1991].

1.3. Research Questions

Despite the prevalence of basic web development in practice and its potential as a vehicle for computational literacy, little research has examined the difficulties beginners face when learning HTML and CSS, the computational concepts and skills that they engage with, and how these critical early moments can be turned into more sustained engagement with computation.

The overarching goal of this dissertation is to investigate the largely unexplored terrain of difficulties beginners have when learning basic web development. Specifically, the studies presented in this dissertation pose the following research questions:

RQ1. What are the barriers students encounter in an introductory web development course?

RQ2. What types of errors do beginners commonly make when using HTML and CSS?

RQ3. What computational concepts and skills do beginners engage with when learning HTML and CSS?

RQ4. How can a web editor be designed to support beginners in learning HTML and CSS?

1.4. Methodology

I address these research questions through design-based research (DBR), a methodological approach in the learning sciences that acknowledges the essential complexity within which learning occurs [A. L. Brown 1992; Collins 1992]. In DBR, research alternates between the design of sociotechnical interventions, guided by theoretical principles derived from earlier research, and evaluation of their effects on teaching and learning within the "blooming, buzzing confusion" of real-life settings [Barab and Squire 2004]. DBR has two

principal qualities: its embraces the situated nature of learning, and it attempts to transform learning through innovative interventions.

First, DBR recognizes the situated nature of learning [J. S. Brown et al. 1989] and addresses it head-on by studying learners in their natural settings. Learning and the context in which it happens are considered inseparable: from the interplay between teacher, student, curriculum, tools, and the activities, policies, cultures in which they are embedded, emerge interactions that play an instrumental role in how learners learn. This approach contrasts with the tradition of laboratory experiments, where variables are strictly controlled and learning outcomes narrowly measured. Due to resource limitations and the ethical questions that arise, it is rarely possible in educational settings to control confounding factors and carefully select participants. Here, DBR errs on the side of ecological validity by studying learners within these settings, at the cost of precise experimental results.

Second, DBR has a transformative agenda. Concomitant with the goal of advancing theory is the improvement of practice, driven by designing sociotechnical interventions and evaluating their impacts. Collins [Collins et al. 2004] draws connections between such educational interventions and "artificial sciences" like aeronautics engineering and artificial intelligence [Simon 1996]. In contrast to natural sciences such as physics, biology, and anthropology that strive to develop explanatory theories for observed phenomena, DBR investigates how designed systems affect teaching and

learning, and seeks to play an active role in positively influencing these outcomes. Typically, DBR takes an iterative form of progressive refinement, alternating between the design of interventions, deployment in natural settings, and evaluation of outcomes in order to generate new theories and inform the next round of design. In the present research, I report on the design and deployment of openHTML, a web editor that aims to support learning HTML and CSS.

DBR is an overarching approach and does not prescribe specific research methods however. Challenges stemming from the quantity and complexity of the real-world data that is generated by DBR often calls for a blend of ethnographic and quantitative approaches. Additionally, learning is a dynamic process, but cannot easily be measured. One cannot simply peer inside the minds of participants and observe learning as it occurs, but must rather adopt a variety of methods for externalizing it or otherwise finding useful proxies for it. In this dissertation, the methods I rely on for this purpose include thematic analysis [Braun and Clarke 2006] of forum content, field studies [Corbin and Strauss 1998], verbal protocol analysis of think-aloud tasks in a laboratory [Ericsson and Simon 1993; Chi 1997], and log analysis [Guzdial 1993], complemented with surveys and interviews. I provide a detailed discussion of these methods in later chapters.

Despite the diversity of these methods, they are well integrated within the DBR approach. Beginning with the web workshop, the studies deploy

progressive versions of openHTML, an experimental web editor for beginners, and each study informs the next round of design. The methods as carried out reflect the situated nature of learning to as great an extent as the circumstances allowed, culminating with the study of students in a live web development course.

Although a laboratory-based study may not fit squarely with DBR and capture the full complexity and richness of how people practice web development in the real world, such controlled studies can serve a complementary purpose, exploring specific phenomena and informing more complex, higher-stakes interventions [Gilmore 1990; A. L. Brown 1992]. For instance, a researcher may identify interesting behaviors in a lab study, thereby becoming sensitized to look for similar patterns in the noisiness of a live classroom. In my laboratory study however, I nevertheless preserved the online context of web development practice by allowing participants to conduct web searches to help them complete tasks, which previous research has shown to comprise a major component of web development learning and workflow [Rosson et al. 2004; Dorn and Guzdial 2010b].

Typically in DBR, multiple rounds of research are conducted in the same organizational setting, giving rise to an increasingly refined understanding of the context and the co-design of system and environment. The research presented in this dissertation diverges from this convention, shifting focus from graduate students of library science in an online course, to children in an

after-school workshop, to undergraduate students in a face-to-face course. This was due to the evolving nature of the research, as well as limitations in the access to participants. Nonetheless, broadening the populations under study also confers benefits given the sparseness of prior research on how beginners learn HTML and CSS. Each group pushed the bounds of learning web development in different ways, from exploring the barriers faced by non-technical library science students, to evaluating openHTML and workshop activities for young elementary students, to capturing the range of errors made by participants possessing diverse backgrounds, to investigating the coding behavior of undergraduate students in their first substantial engagements with HTML and CSS.

I note a final commonality in the primary data sources used in my studies. While retrospective methods such as interviews are invaluable for capturing the perspectives and sensemaking of participants, especially at the time of the data collection, memory is notoriously fallible and recalling the order of events that occurred weeks or months ago can be problematic. This is perhaps even more the case for novices, such as web development students, who may have a limited ability to introspect or accurately recall details about their code [Adelson 1981; McKeithen et al. 1981]. Furthermore, what the learner failed to notice or does not fully understand is often precisely what is of greatest interest. Therefore, in all of my studies I have attempted to triangulate retrospective data, such as interviews and surveys about past experiences, with

activity data such as forum posts, field notes, video recordings, and activity logs that are generated contemporaneous to the act of learning and practicing web development.

1.5. Structure of the Dissertation

This dissertation reports on three studies that investigate different aspects of learning and practicing web development, as well as the iterative design of openHTML.

In the first study, I analyzed the help forums of a web development course offered to library science students. I identified five broad types of barriers that students sought help for: administration, technology, code, design, and content. Further analysis revealed that the majority of code barriers related to many basic aspects of HTML and CSS, warranting a deeper investigation of the difficulties beginners have with these languages.

Guided by insights from this study, I designed and developed the initial version of openHTML, namely abstracting away technological issues such as installing and configuring software, facilitating aspects of administration such as sharing code, and positioning code as the focal point of the interface. An implementation of openHTML was then pilot-tested in an after-school workshop for elementary students, in order to assess its robustness and usability.

In the second study, I used openHTML to conduct a laboratory-based study that examined the syntactic and semantic errors participants made when

constructing web pages using HTML and CSS. Applying a framework of human behavior [Rasmussen 1983], I classified a wide range of common errors according to their cognitive causes. Additionally, I found that approximately 70 percent of errors involved invalid syntax, supporting the viability of syntax errors as a window into the difficulties that beginners have with HTML and CSS.

In the final study, I turned back to a live web development course, conducting a fine-grained analysis of the syntax errors undergraduate students make with HTML and CSS during the initial weeks of the course. Two terms of this course were preceded by iterations on the design of openHTML to support its deployment in formal learning contexts. Analysis revealed that two computing concepts, nesting and parent-child rules, underlay the majority of these errors, and that validation was an effective practice for resolving them in most instances.

A timeline illustrating the studies, in terms of data collection and analysis, and how each study informed subsequent rounds of research and design, is given in Figure 1-1.

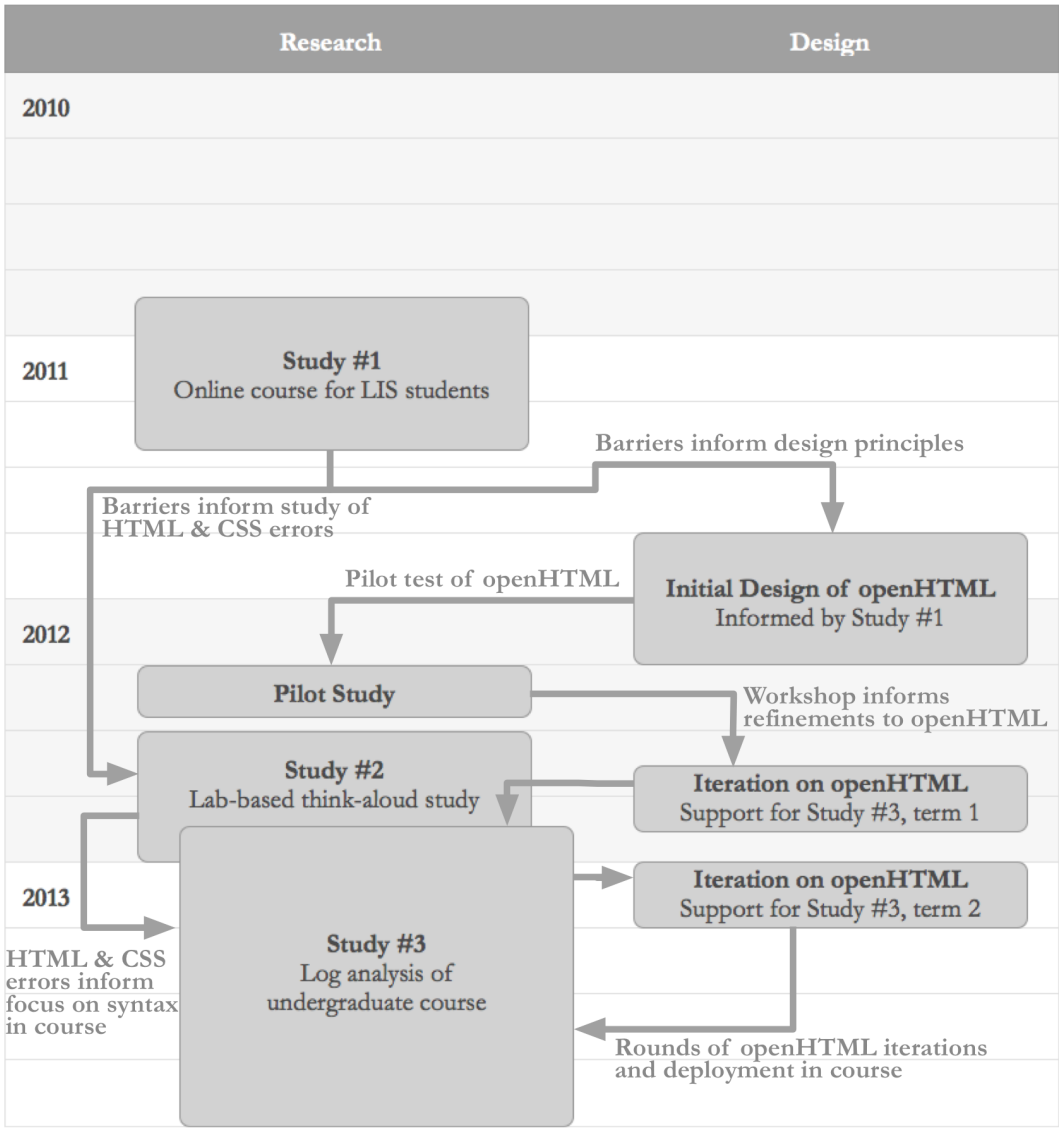

Figure 1-1: A timeline of the research and design described in the dissertation.

The remainder of this document is organized in the following chapters:

- Chapter 2 reviews related literature from the domains of CS education and human-computer interaction.
- Chapter 3 presents the first study, which reports on learning barriers found in an online web development course. This study provides context

for challenges specific to coding in HTML and CSS, and is based on work published in [Park and Wiedenbeck 2011].

- Chapter 4 describes the initial design and implementation of openHTML, as well as a pilot study evaluating it in an after-school web workshop. The design of openHTML was reported in [Park, Saxena, Jagannath, Wiedenbeck and Forte 2013b] and the web workshop in [Park, Magee, et al. 2013a].
- Chapter 5 details the second study, a laboratory-based study of common errors people make when writing HTML and CSS. This study was published in [Park, Saxena, Jagannath, Wiedenbeck and Forte 2013c].
- Chapter 6 presents the third study, where I deployed openHTML in a web development course and investigated the syntax errors that students made in the initial weeks of the course. These findings will be reported in [Park et al. in press].
- Lastly, Chapter 7 summarizes the contributions of this dissertation and discusses future research directions.

Chapter 2
Related Literature

This chapter provides an overview of prior research related to this dissertation. I draw from a rich body of computing education literature on how people learn to program, as well as the relatively sparse research on learning web development. Each section corresponds to one of my four research questions. I start with studies of teaching and learning web development in formal and informal contexts. Then I briefly summarize research on the errors novices make and the misconceptions they have when learning to program. I continue with efforts to define the concepts that are fundamental to computational literacy and the computer science discipline. Finally, I conclude by discussing work on designing programming environments to support web development, particularly with respect to helping beginners overcome barriers and resolve errors.

2.1. Teaching and Learning Web Development

The computing education literature describes numerous examples of courses that have used web development as a context to teach programming and other computational concepts. Many of these studies have focused on the challenges

faced by educators rather than students, and are limited to anecdotal data and informal observations when reporting on student experiences.

The earliest accounts focused on curricular challenges stemming from the broad array of technologies involved in web development and their rapid evolution [Lim 1998; Walker and Browne 1999]. Although course materials required significant overhaul after only a few months due to the "web pace" at which technologies advanced, researchers offered anecdotal support for a breadth-first approach that surveys web development. Based on the personal observations, Lim [Lim 1998] noted that CS students met the intended outcomes of his course and were enthusiastic about their assignments. As evidence for the efficacy of this approach, Walker and Brown [Walker and Browne 1999] reported positive feedback from a student after the course and several cases where students went on to pursue web development professionally. Web development courses aimed at non-computing majors have similarly been evaluated based on teacher observations, indicating high levels of engagement and the potential for difficulties among non-majors transitioning from HTML to JavaScript [Mercuri et al. 1998; Reed 2001].

Klassner [Klassner 2000] describes a web development course that tries to obviate the need for keeping pace with the state of the art by emphasizing functionality rather than the particulars of implementation. He evaluated this approach by surveying students at the midpoint and end of the course, asking questions such as "What elements of the course do you find most useful?"

and "What elements would you want to see changed?" Among his findings were that students were evenly divided between whether a server-side assignment in the first half of the course was at an appropriate level or too ambitious, and that students found the unit on compression techniques too theoretical and overly removed from real-world applicability.

Treu [Treu 2002] adopted a seminar format in which students worked collaboratively to complete a project, selecting topics for themselves as the need arose and teaching them to the rest of the class. In addition to informal observations about the enthusiasm of students in the class, he administered a quantitative survey that asked students to rate how much they learned in the course and the effectiveness of the case study approach. Students rated these numbers highly, although it is difficult to draw strong conclusions given the lack of a comparison point. Sridharan [Sridharan 2004] described a web development course that utilized a strategy of program completion in which students are provided with partial programs and tasked with completing the missing portions, making the switch between multiple forms of technology manageable compared to a program generation strategy in which students are expected to build programs from scratch. Like Treu, he assessed this approach by analyzing course evaluations and found that students also rated nearly all aspects of the course highly. Gurwitz [Gurwitz 1998] provided the most detailed findings based on a post-course survey. Students once again responded positively on the whole, with criticisms centered on acute

administrative problems such as unreliable Internet access and inconvenient computer lab locations.

Studies have also explored the backgrounds and practices of experienced web developers. Interviews with informal web developers who lack formal training or responsibilities but nonetheless find themselves maintaining websites [Rosson et al. 2004] revealed that less expert developers lacked a systematic view of web development and instead developed "pockets of expertise" as they encountered and learned to resolve specific issues. Studies of professional web developers [Dorn and Guzdial 2010a; Dorn and Guzdial 2010b] raised similar issues due to their lack of formal computing education. In the case of both hobbyist and professional web developers, learning was opportunistic in nature and relied heavily on online resources found through web searches, including documentation and code examples.

In an introductory web development course, students can encounter many new aspects of computation, yet there has been little research on their experiences. Given the different circumstances in which experienced developers and students of a structured web development course are operating, the barriers they face are likely to differ considerably. While case studies of web development courses offer some insight, most have assessed their approaches using informal observations and anecdotal data. End-of-course surveys and evaluations have also identified potential barriers to learning web development, but these findings are relatively coarse and focused

on only the most acute problems due to their retrospective nature. An improved understanding of the barriers students face in their first web development course serves as the first step in addressing them.

2.2. Programming Errors and Misconceptions

In computing education research, the errors students make and the misconceptions they hold about programming have long served as a window into their state of understanding [Smith et al. 1993], informing teaching practice and tool design. A brief review of studies representative of this work illustrates potential insights that might be gained from a similar study of HTML and CSS.

The path to programming expertise is a long one [Linn and Dalbey 1985], and the literature makes clear that novices have significant difficulties learning to program on multiples levels [duBoulay 1986]. A series of studies have demonstrated that after a year or more of study, CS students continue to fall short of expected outcomes in their ability to trace [Lister et al. 2004], design [Loftus et al. 2011], and write [Kurland et al. 1986; McCracken et al. 2001] computer programs. Students often enter their first programming course with an impoverished model of the computer [Ben-Ari 1998].

Studies have found that the distribution of errors can be roughly characterized as a power law distribution, where a few types of errors are responsible for the majority of instances. One of the earliest and most extensive classifications of programming errors comes from a study of 73

students learning Cobol [Litecky and Davis 1976]. Litecky and Davis reported that 20 percent of error types were responsible for 80 percent of the errors students made, advocating for teachers to focus on these most common errors when teaching students.

One way that the nature of programming errors has been examined is by classifying them as relating to syntax, semantics, or logic and design. Youngs [Youngs 1974] assigned programming tasks to students and professionals, comparing the errors they made in terms of the statement type (e.g., assignment, input/output, iteration), the specific manifestation of the error (e.g., formatting, omission, illegal operation), and the depth of understanding required to correct it (e.g., syntax, semantic, logic). He found that experts were able to correct syntax and semantic errors quickly, while these lower-level aspects of programming were more troublesome for students. A study by Garner et al. [Garner et al. 2005; Robins et al. 2006] documented the problems students encounter in an introductory programming course using Java, finding over 11,000 problems that students sought help for during lab sessions and classifying them into 27 categories ranging from tools and task understanding to control flow, loops, and hierarchies. The authors expressed surprise at "the persistence, frequency, and uniform distribution of problems relating to basic syntactic details" such as typos and missing semicolons.

Given the difficulties syntax poses for beginners, researchers have gained insights by focusing on the syntax errors students frequently commit. For

example, Jadud [Jadud 2005] instrumented the BlueJ programming environment to log the compilation behavior of 63 students and catalogued that most common types of syntax errors. How well students cope with syntax errors has been found to be one of the most effective predictors of student achievement in a course [Rodrigo et al. 2009].

The literature explains that many of the programming errors that novices and experts make are the result of consistently applying misconceptions that they hold. Even when lacking a sufficient knowledge base, people build mental models of a program, and although these conceptualizations can be incomplete or unworkable, they are nevertheless the result of "systematic applications of the knowledge a student currently does have to the problem at hand" [Pea et al. 1987]. They are often logical conclusions based on current understanding, and for this reason can be extremely resistant to change once set. Much research has been devoted to identifying these misconceptions to aid in the design of courses and curricula [Winslow 1996].

Bayman and Meyer assessed undergraduate students learning BASIC and catalog misconceptions of single-line statements [Bayman and Mayer 1983]. They identified a number of misconceptions related to variables, assignments, and conditionals, and conclude that hands-on experience with programming is not sufficient:

> "Users tend to develop conceptions of the statements that either fail to include the main idea or that include outright

> misconceptions. Explicit training is needed including the introduction of a concrete model..."

Putnam et al.'s study [Putnam et al. 1986] similarly looks at the misconceptions high school students have about programming in BASIC. By administering screening tests and interviews, they found that many misconceptions related to basic programming constructs such as variables, assignments, and loops. Furthermore, misconceptions about these basic concepts "[impede] productive engagement in higher level problem solving skills such as planning and debugging."

Spohrer and Soloway [Spohrer and Soloway 1986b] caution that misconceptions about language constructs may not be the primary source for programming errors. They hypothesize that this "folk wisdom" may stem from experts seeing bugs in terms of what constructs are needed to correct them and incorrectly concluding that the bugs are due to a lack understanding of these constructs. In analyzing syntactically correct programs created by 61 students, they found 284 bugs and classified them into 101 different bug types [Spohrer and Soloway 1986a]. They built "plausible accounts" on the origins of these bug types, and concluded:

> "...misconceptions about language constructs do not seem to be as widespread or as troublesome as is generally believed. Rather, many bugs arise as a result of plan composition problems –

> difficulties in putting the pieces of the program together [...] – and not as a result of construct–based problems, which are misconceptions about language constructs."

In related work, "student-constructed rules" about parameter passing have been identified by conducting interviews where students were asked to predict from a set of programs which work and why [Fleury 1991], analyzing programs written by students in an introductory programming course and develop a checklist of code features that indicate understandings or misconceptions about object-oriented programming [Sanders and Thomas 2007]. Holland et al. [Holland et al. 1997] outline pedagogical strategies for avoiding misconceptions about OOP, describing examples that can be used in class to challenge the most common cases.

Perhaps the single overarching misconception about programming among novices is what Pea calls the "superbug". This occurs when novices act as if the computer has an intelligent mind that can infer intentions from imprecise language in the same way that natural language is used in interpersonal discourse [Bonar and Soloway 1985; Pea 1986]. In a study of misconceptions about programming among high school students, Putnam et al. conclude that many of the misconceptions can be similarly attributed to the "inappropriate imposition of reasoning and knowledge from more informal domains to the formal domain of programming" [Putnam et al. 1986].

The choice of language has also been found to play a large role in the nature of errors students make when learning to program. Stefik and Siebert [Stefik and Siebert 2013] have examined novices using a variety of programming languages such as Java, Python, and Perl, finding significant differences in the accuracy rates depending on language. This is supported by studies like Anderson and Jeffries's [Anderson and Jeffries 1985], which found most errors made by novice programmers using LISP, a programming language that makes heavy use of nested parentheses, involved slip errors with said parentheses.

As with the literature described in this section, examining the errors and misconceptions people have with HTML and CSS can be a fertile approach to understanding how they learn web development, what they learn about computing more generally, and ways of improving support. However, given that relatively similar programming languages lead to significant differences in the types of errors novices make, what might be expected of students learning HTML and CSS, representing entirely different paradigms as markup and stylesheet languages?

2.3. Fundamental Computing Concepts

In order to investigate the computational knowledge students develop through basic web development, I turn to the work of researchers and educators who have taken a variety of approaches to identifying concepts that are fundamental to computer science.

One example is Schwill's framework of fundamental ideas in CS [Schwill 1994], influenced by Jerome Bruner's principle that a scientific discipline should be oriented by fundamental ideas. Schwill outlines four criteria a concept must meet in order to be considered fundamental to a discipline:

- Horizontal criterion: the idea must be widely applicable in the domain.
- Vertical criterion: the idea can and should be taught at all levels of age and education.
- Criterion of time: the idea is observable in the history of the domain.
- Criterion of self: the idea is applicable in everyday life.

By iteratively applying these criteria to evaluate CS ideas, Schwill arrived at algorithmization, structured dissection, and language as candidates for master ideas in CS, decomposing these to other fundamental ideas as shown in Figure 2-1.

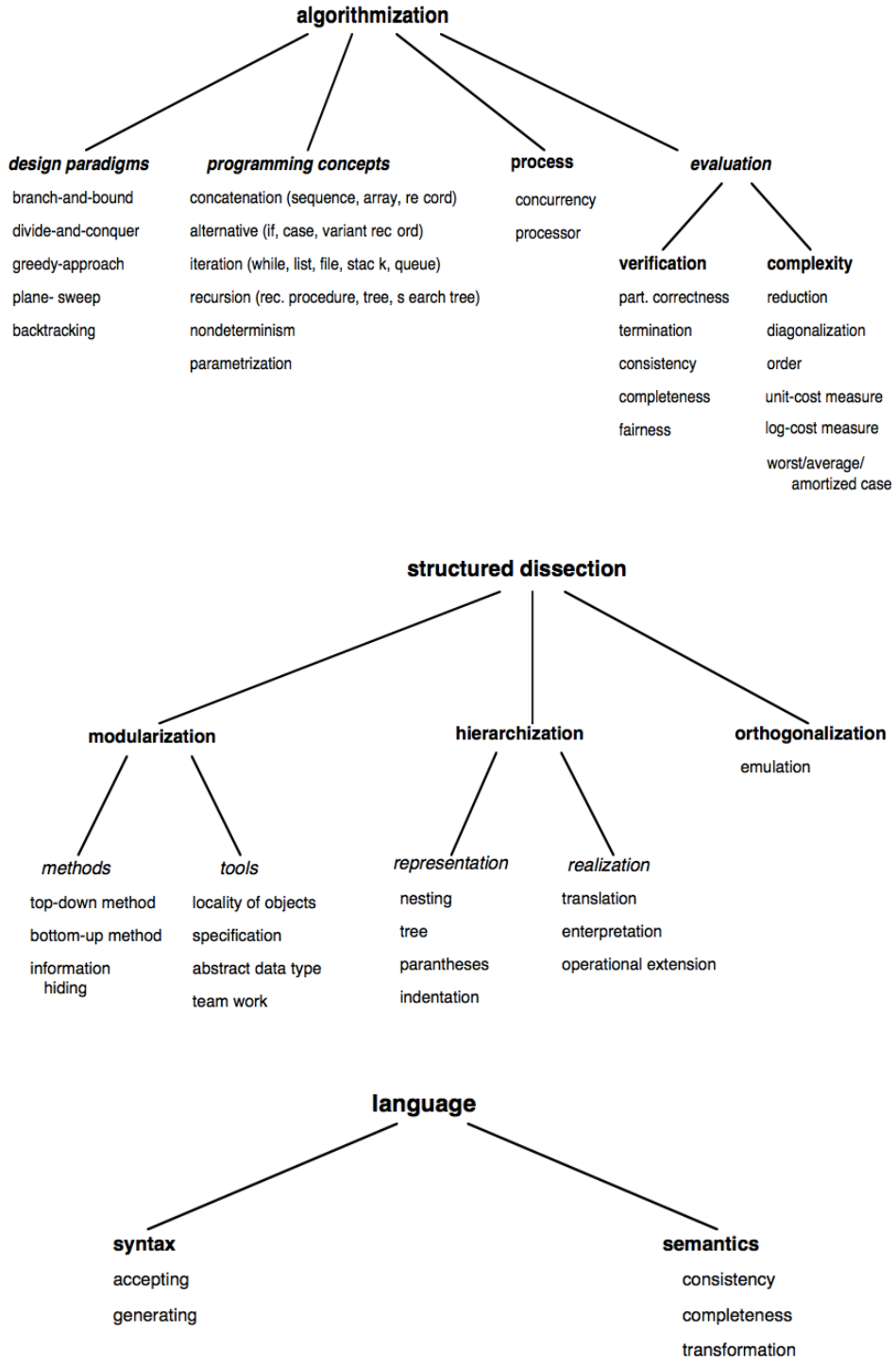

Figure 2-1: Fundamental ideas of CS as proposed by Schwill [1994].

Concept inventories can also characterize the concepts central to a discipline. First used in science education [Peterson et al. 1994; Hestenes et al. 1992], concept inventories are assessments that evaluate critical concepts and identify the precise misconceptions students hold about them. Concept inventories typically take the form of multiple-choice exams. For each question, the correct answer is accompanied by several "distractors" based on common misconceptions that have been identified previously through research. Goldman et al. [Goldman et al. 2008] take the first step in developing a concept inventory for CS by establishing the scope of concepts. Following a Delphi process to achieve consensus among a group of experts, they identify the concepts that are considered most important and difficult in CS. Programming concepts with the greatest consensus included procedure design, scope, inheritance, abstraction, recursion, and debugging (Figure 2-2).

ID	Topic	Importance	Difficulty
	Procedural Programming		
PA1	1. Call by Ref. vs Call by Value	7.0 (2.5)	7.4 (1.2)
PA2	2. Formal vs. Actual Parameters	8.6 (1.2)	5.7 (1.7)
PA3	**3. Parameter scope, use in design**	**9.1 (0.9)**	**7.5 (1.0)**
PROC	**4. Procedure design**	**9.8 (0.4)**	**9.1 (0.8)**
CF	5. Tracing Control Flow thru program execution	9.8 (0.4)	7.0 (0.6)
TYP	6. Choosing/Reasoning about data types	8.2 (1.5)	6.6 (0.5)
BOOL	7. Construct/evaluate Boolean expressions	9.5 (0.7)	7.1 (0.8)
COND	8. Writing expressions for conditionals	9.5 (0.5)	6.7 (0.6)
SVS	9. Syntax vs. Semantics	8.6 (0.7)	7.5 (0.5)
OP	10. Operator Precedence	7.1 (1.5)	4.4 (0.5)
AS	11. Assignment Statements	9.5 (1.2)	4.4 (0.5)
SCO	**12. Issues of Scope, local vs. global**	**9.4 (0.7)**	**8.0 (0.0)**
	Object Oriented Programming		
CO	13. Difference between Classes and Objects	10.0 (0.0)	6.9 (1.4)
SCDE	14. Scope design (e.g., public vs private fields)	9.4 (0.7)	6.6 (0.5)
INH	**15. Inheritance**	**7.6 (1.7)**	**9.5 (0.5)**
POLY	16. Polymorphism	7.1 (1.4)	8.9 (0.8)
STAM	17. Static fields and methods	5.7 (1.3)	7.3 (0.6)
PVR	18. Primitive vs Reference variables	8.5 (2.4)	7.0 (0.8)
	Algorithm Design		
APR	**19. Abstraction/Pattern recognition and use**	**8.8 (0.4)**	**9.0 (0.4)**
IT1	20. Tracing nested loop execution correctly	9.5 (0.5)	6.6 (0.7)
IT2	21. Understanding loop variable scope	8.7 (2.0)	4.3 (0.9)
REC	**22. Recursion, tracing and designing**	**7.8 (2.4)**	**9.2 (0.9)**
AR1	23. Identifying off by one index errors	8.9 (0.8)	5.3 (0.5)
AR2	24. Reference to array vs array element	8.4 (1.4)	5.7 (0.7)
AR3	25. Declaring and manipulating arrays	9.0 (1.4)	5.5 (0.5)
MMR	**26. Memory model, references , pointers**	**7.5 (1.7)**	**8.9 (0.7)**
	Program Design		
DPS1	**27. Functional decomposition, modularization**	**9.3 (0.6)**	**7.9 (0.8)**
DPS2	**28. Conceptualize problems, design solutions**	**9.5 (0.5)**	**8.5 (0.5)**
DEH	**29. Debugging, Exception Handling**	**9.0 (0.0)**	**8.6 (0.5)**
IVI	30. Interface vs Implementation	8.1 (0.8)	7.5 (0.5)
IAC	31. Desiging Interfaces, Abstract Classes	5.0 (1.1)	8.6 (0.7)
DT	**32. Designing Tests**	**9.3 (0.8)**	**8.4 (0.8)**

Figure 2-2: Fundamental programming topics with expert ratings for importance and difficulty as reported by Goldman et al. [2008].

In developing a language-independent assessment for introductory programming, Tew and Guzdial [Tew and Guzdial 2010] analyzed the content of Computer Science volume of the Computing Curricula 2001, popular textbooks, and other documents, distilling over 400 concepts down to ten concepts fundamental to programming.

- Fundamentals (variables, assignment, etc.)
- Logical Operators
- Selection Statement (if/else)

- Definite Loops (for)
- Indefinite Loops (while)
- Arrays
- Function/method parameters
- Function/method return values
- Recursion
- Object-oriented Basics (class definition, method calls)

Card sorting studies have adopted a similar approach to identify a small set of programming concepts [Sanders et al. 2005]. Most relevant to this dissertation, Dorn and Guzdial conducted a card sort to investigate the programming knowledge of professional web developers [Dorn and Guzdial 2010b]. They found that despite only one of 12 participants holding a CS degree, they had high rates of recognition and usage of 26 programming concepts. However, they lacked a systematic view of programming given their lack of formal training in CS. One conclusion is that web developers may benefit from studying CS. An alternative view is that there is an opportunity to make the connections between web development and underlying computing concepts more explicit in the resources currently used to teach and learn web development.

Threshold concepts have alternately been proposed as a way to organize and focus computer science as a discipline [Eckerdal et al. 2006]. Threshold

concepts are defined as concepts that are transformative in the way students view the discipline. Criteria for threshold concepts include that they are irreversible in that they are difficult to unlearn, integrative in tying together concepts in a new way, and potentially troublesome in that they can be difficult and counter-intuitive. Through interviews, Boustedt et al. [Boustedt et al. 2007] suggest object orientation and pointers as potential threshold concepts, although Shinners-Kennedy and Fincher [Shinners-Kennedy and Fincher 2013] temper their enthusiasm for classifying threshold concepts, particularly through retrospective interviews.

Finally, the term "computational thinking" has been used to describe the practices and knowledge central to computer scientists that can benefit all people in dealing with complexity and solving problems [Wing 2006]. Concepts like data representation, modeling, algorithms, abstraction, and decomposition, have been cited as aspects of computational thinking. However, Pea and Kurland [Pea and Kurland 1984] have long cautioned that there is a dearth of evidence supporting the development of higher-order reasoning skills that can transfer to distant domains, particularly at the lower levels of programming skill development, and that much more empirical research is need.

Concepts fundamental to CS have been identified through a variety of perspectives, but the results share many commonalities. Concepts are largely based around the syntax and semantics of language constructs, or relate to

ways of managing complexity in design. The question of whether these concepts are truly fundamental or act as thresholds may never be definitively answered, but their appearance across multiple efforts indicate their importance to computing knowledge.

Many of the identified concepts, such as syntax, parameterization, conditionals, and abstraction, have analogues in HTML and CSS. For instance, HTML elements can be assigned values to various properties in much the same way as objects in object-oriented programming, and CSS media queries define the conditions by which styles take effect. These concepts form connections between basic web development and the broader computing education literature, and lend support to basic web development as a vehicle for engaging with important aspects of computation.

2.4. Tools for Learning HTML and CSS

In order to fully understand how people learn web development, the role that technology play in it, both as mediator of activity and as object of mastery itself, must be considered [Nardi 1995]. Web development tools shape how people engage in and think about web development. After a web development tool has been retired for another, it can leave a lasting impact through the learning that has occurred and the social practices that have evolved through its use.

Web authoring tools generally offer two modes of interaction: the power and efficiency of code editors, or the ease of use of WYSIWYG (what-you-

see-is-what-you-get) editors. Traditionally, web development is practiced by directly editing the source code of a web page in its native, textual language. This is accomplished with the use of a code editor, which can include features like syntax highlighting, bracket matching, and auto-completion (Figure 2-3). Although this textual approach remains popular for the degree of control it affords, researchers have noted its drawbacks. Greene and Petre explain this in terms of the mapping between the code and the real world: "The closer the programming world is to the problem world, the easier the problem-solving ought to be... Conventional textual languages are a long way from that goal" [Green and Petre 1996]. Particularly for novices, the abstract and exacting nature of textual languages poses a significant challenge. Code editors are often designed with power users in mind, providing minimal support for beginners and squandering an opportunity to create a supportive learning environment. Without this support, learners may fail to develop models that adequately equip them to make sense of web development at a deeper conceptual level.

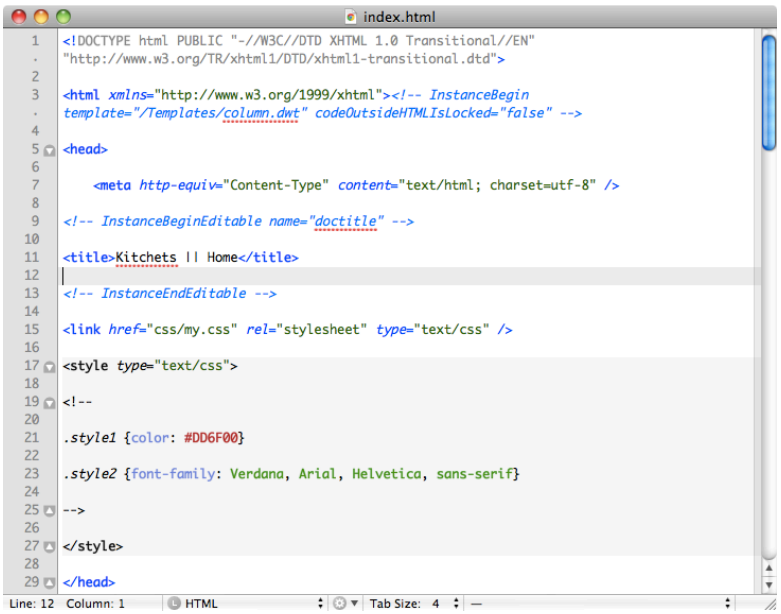

Figure 2-3: TextMate, a code editor with syntax highlighting and bracket matching.

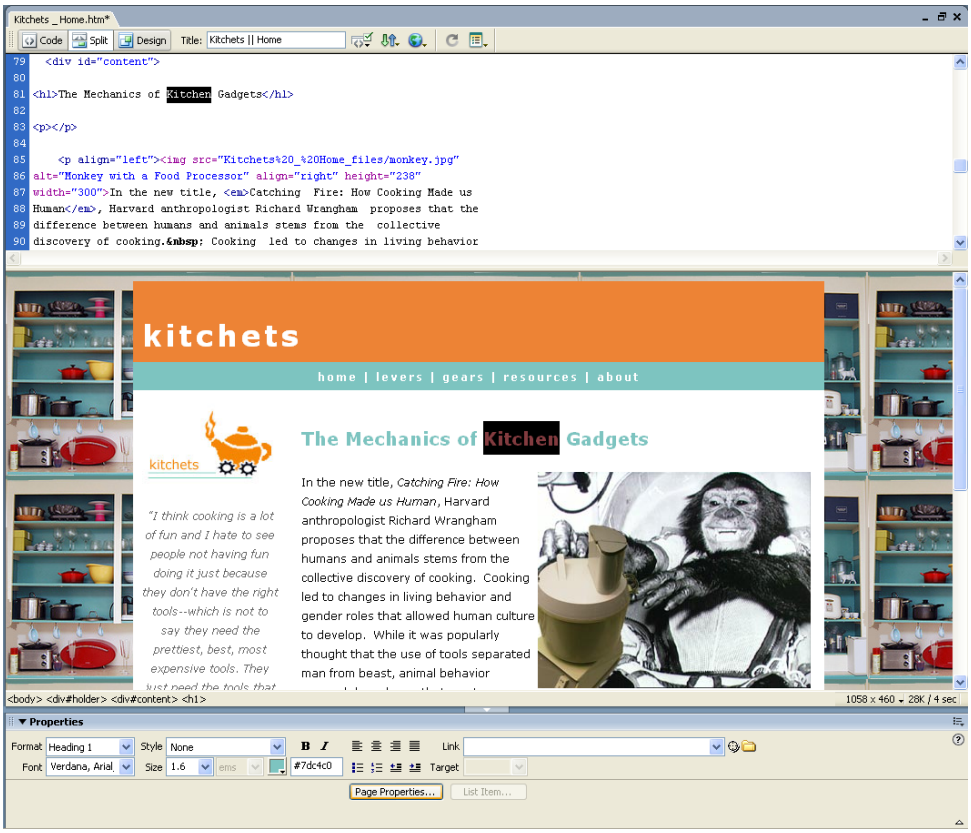

Figure 2-4: Dreamweaver, a web development IDE, with code pane at top and WYSIWYG pane at bottom.

An alternative approach that often appeals to novices is WYSIWYG. A WYSIWYG editor applies the principles of direct manipulation [Shneiderman 1983; Hutchins et al. 1985], enabling users to edit the web page and receive immediate feedback by interacting with its visual output (Figure 2-4).

While WYSIWYG lowers the barrier to entry for coding, it too is not without its shortcomings. WYSIWYG editors have difficulty interpreting the intent of a user's direct manipulations. They often generate inefficient and unreadable source code. WYSIWYG also shields users from the underlying code from which they might otherwise learn to create abstractions and make inferences, such as how code renders in untested conditions. Ben-Ari [Ben-Ari 1998] writes:

> What you see is not what you get: what you get is an internal data structure containing your document and a set of operations for transforming the data structure; what you see is merely a visual representation of the structure... You have to construct a viable model that will enable you to predict the outcome of any operation on the model, and to predict how that outcome will be reflected in the representation you see. The relevance for CSE is that courses, help files and tutorials must explicitly address the construction of a model, and not limit themselves to behaviorist practices of the form 'to do X, following these steps'.

du Boulay warns that "even if no effort is made to present a view of what is going on 'inside' the learners will form their own" [duBoulay 1986]. Lack of appropriate support can create impoverished models that are insufficient for explaining observed behavior. WYSIWYG editors in particular can lead to a misapplication of analogy, where learners intuit more than is warranted from the document metaphor.

In the vocabulary of Sedig, Klawe, and Westrom [Sedig et al. 2001], WYSIWYG editors serve as a form of direct object manipulation, as opposed to direct concept manipulation. The authors explain that "unlike objects whose meaning is at the 'surface' level, conceptual representations can embed knowledge at several levels, making these representations 'highly abstract and with great interiority' of meaning." In the domain of transformation geometry, they have compared direct manipulation interfaces to ones that give explicit representation to concepts like rotation and translation, finding the latter to significantly improve student understanding. In the case of basic web development, such an interface might offer direct manipulation of the CSS box model, which determines the appearance and position of elements, rather than merely the rendered output of the webpage that a visitor would see.

Full-featured integrated development environments (IDEs) such as Dreamweaver juxtapose textual and WYSIWYG modes and can provide an array of additional features aimed at supporting productivity. A drawback here

is that this complexity can overwhelm novices lacking a firm conceptual grasp of web development, though the extent of this effect is not well explored.

Usage of WYSIWYG and code editors is mixed. Vora's 1998 survey found that when comparing editors, web developers rated code editors most highly along a number of measures, including functionality, extensibility, ease of learning, ease of use, and satisfaction [Vora 1998]. A 2005 survey found that while programmers were mixed in their preferences (38% for WYSIWYG editors versus 30% for text editors), non-programmers strongly preferred IDEs providing WYSIWYG interfaces (Dreamweaver and FrontPage combined for 65.9%) over text editors (13%) [Rosson et al. 2005].

While the literature provides numerous examples of learning environments designed to support programming, most notably Logo [Harel and Papert 1990], ALICE [Cooper et al. 2000], Scratch [Resnick et al. 2009], BlueJ [Kölling et al. 2003], and DrScheme [Findler et al. 2002], research on systems that support HTML and CSS is much thinner.

RUMU Editor [Poley 2010] is a web development tool that attempts to reconcile the needs of non-technical developers with some of the shortcomings of WYSIWYG. Users select a layout template, which reveals multiple text fields that correspond to content areas such as header, sidebar, and main body. Users then input their content and tag it semantically using a simplified textual language called Markdown. A predefined stylesheet can be applied to the tagged content, a preview can be invoked, and code can be

generated in XHTML and CSS. Poley conducted an experiment in which participants were provided about twenty minutes to create a two-page website, using either RUMU Editor or iWeb, a commercial WYSIWYG editor. Participants using RUMU Editor showed greater variance in completing the task, with a slightly higher percentage successfully building the website. In a post-study survey, the users of RUMU Editor also reported a slightly higher level of satisfaction.

Virtual Lab is a web-based learning environment that supports HTML coding activities [An 2007]. Users are presented with a problem, submit the code needed to solve it, and receive feedback on how the code renders and the errors that have been committed (Figure 2-5).

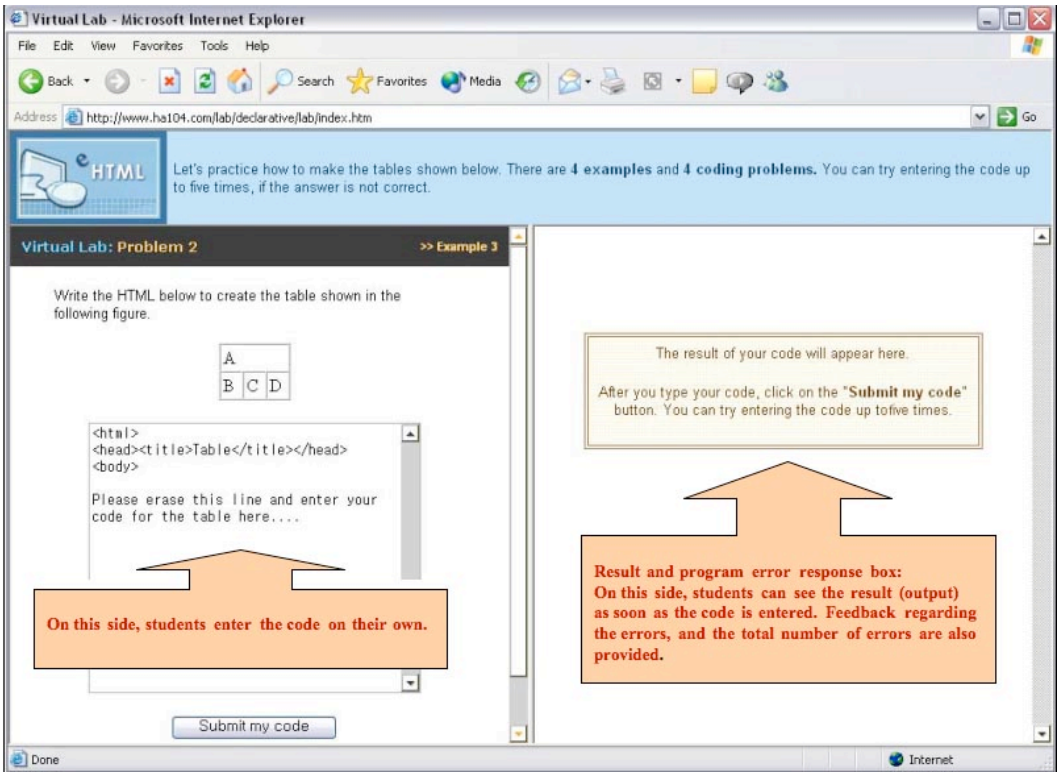

Figure 2-5: Virtual Lab, a web-based environment for learning HTML.

Kaplan and An use Virtual Lab to investigate the effects of different representations of worked examples on students learning HTML [Kaplan and An 2005]. Their study involves three different representations of worked examples: facts, procedures, and visual model. In the facts version, the example code is accompanied with factual information about the syntax and functions of the elements. In the procedures version, step-by-step instructions are given for constructing the example code. Finally, in the visual model, the example code is mapped to a diagram of its structure and to the visual output.

Twenty students were assigned to each of these three conditions and were asked to complete a lesson on HTML tables. After a brief introduction to the topic, students alternated between worked examples that reveal how an expert might solve a problem in their condition's format, and similar problems that they attempted to solve on their own. The lesson concluded with all of the conditions completing the same two questions on factual knowledge, two on output prediction, and two on error detection.

Kaplan and An found that while all three groups demonstrated a similar level of factual knowledge, the visual model group generated significantly more correct code and fewer conceptual errors in the same amount of time as the other groups. They go on to remark that novices often have difficulty taking surface features of the code, such as indentation, whitespace, and other typographical aspects, and abstracting an underlying structure or relationship to output.

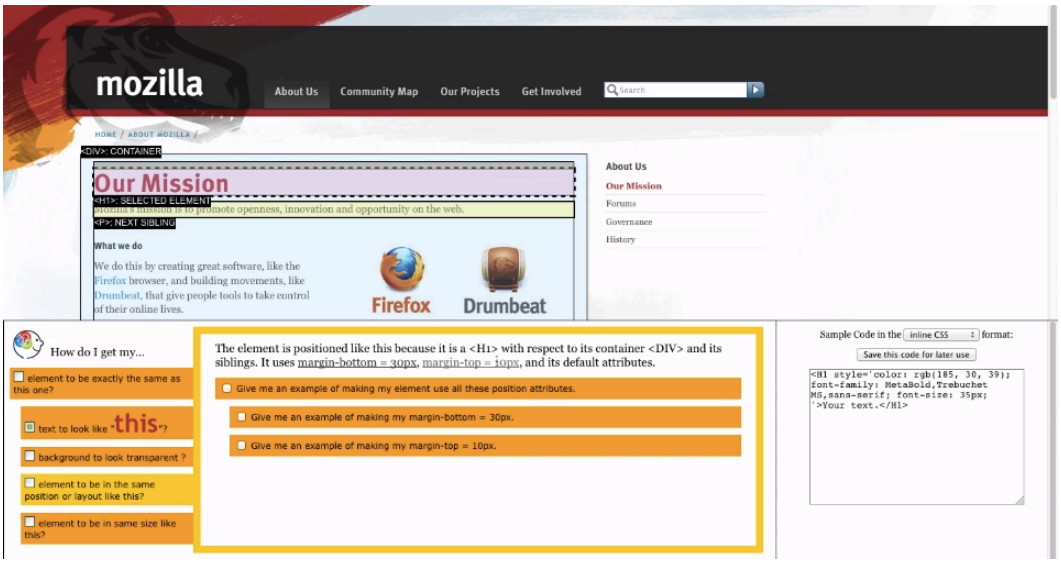

Figure 2-6: WebCrystal, a tool that allows users to learn how to recreate elements on a web page using HTML and CSS.

WebCrystal is a web development tool that allows users to select elements on an existing web page, learn how they are constructed, and extract the relevant HTML and CSS code snippets for reuse [Chang and Myers 2012]. This browser extension prompts users with questions about the aspect of the element they wish to explore, and responds with a textual description and customized code snippet (Figure 2-6). In an evaluation, 6 participants were asked to use WebCrystal and another 6 Firebug (a popular browser extension that facilitates debugging), while completing 10 coding tasks. The participants using WebCrystal completed more of the tasks and in less time. In interviews, participants with novice or intermediate knowledge of HTML and CSS found the textual explanations much more helpful than did the expert users.

This section discusses the major approaches to designing web editors and presents several experimental tools for learning HTML and CSS. Most commercial editors have either been designed using a WYSIWYG interface that lowers the barrier to building web pages but fundamentally changes the nature of the process, or a textual interface that results in greater efficiency for advanced users but raises many barriers for beginners. Experimental systems have either been auxiliary tools for learning assessment (Virtual Lab) and exploration (WebCrystal), or have abstracted the process of building web pages (RUMU Editor). As yet unexplored is how a web editor can be designed to support learners while exposing them to the computational nature of HTML and CSS.

Chapter 3
Identifying Learning Barriers in a Web Development Course

In this chapter, I explore the barriers beginners encounter when learning basic web development, contextualizing the difficulties students have with HTML and CSS within the broader scope of learning web development in a formal learning environment. I accomplish this by analyzing the issues students sought help for in an online web development course. What aspects of the course hindered their progress? What issues turned their enthusiasm into frustration? Were the majority of issues related to writing and reading HTML, CSS, and JavaScript, or were non-coding aspects of web development equally problematic? Were there computational concepts underlying these issues?

The literature offers a wealth of research on identifying and lowering barriers for novice programmers [Robins et al. 2006; Ko et al. 2004; Kelleher and Pausch 2005]. However, few studies have explored such issues in the realm of web development, particularly at the introductory level when students with minimal coding experience are learning HTML and CSS. The struggles and triumphs of non-CS students learning to code can inform not only the design of web development courses and tools, but also of CS courses and tools intending to appeal to broader audiences.

Learning barriers have the potential to be both obstacles and opportunities in the classroom. They can impede progress and induce frustration, anxiety, and attrition among students. Yet, challenge is also an important ingredient of learning. In the right measures, it contributes to student motivation and satisfaction [Ames and Archer 1988]. Barriers can set the stage for "teachable moments" [Hansen 1998], where conceptual conflict leads to a restructuring of beliefs and the assimilation of new ideas [Piaget 1950]. By resolving them with the aid of an instructor or classmate, students practice useful learning strategies [Nelson-LeGall 1985] and develop the ability to resolve similar issues without assistance [Vygotsky 1978]. Help seeking not only can benefit the seeker, but also the helper and other students. Therefore, the goal of identifying barriers is not necessarily to eliminate them, but to inform decisions about whether to deal with them as obstacles to be mitigated or intentional learning opportunities.

The following research questions guided this study:

RQ1. What are the barriers students encounter in an introductory web development course?

RQ3. What computational concepts and skills do beginners engage with when learning HTML and CSS?

By exploring this question, I hoped to identify factors that may cause students to develop negative attitudes toward web development, and uncover opportunities for fostering more productive and enjoyable learning experiences in a web development course.

Section 3.1 provides a description of the course and my data collection and analysis methods. Section 3.2 reports on my findings. Finally, Section 3.3 discusses the implication of these findings in terms of web development education and research.

3.1. Methods

3.1.1. Data Collection

To uncover challenges that students face when learning web development, I examined help-seeking activity in an introductory web development course. In studying cases where students encounter an insurmountable problem and turn to the help forums for assistance, this method shares similarities with critical incident technique [Flanagan 1954] which recognizes the value of examining critical moments in providing insight into the problems participants experience and their potential solutions.

The web development course was offered online to students pursuing Master's degrees in Library and Information Science at a large Mid-Atlantic university. These are students with largely non-technical backgrounds. The course is offered in the curriculum because many librarians go on to work in

small community libraries where the responsibility of maintaining and updating websites falls on them.

The course ran for ten weeks and introduced the topics shown in Table 3-1. During the first eight weeks of the course, each student developed a website incrementally as new topics were introduced, using a barebones text editor such as Notepad. During the final two weeks of the course, students developed a second website using any tool of their choice. Most of the students opted to use Adobe Dreamweaver.

Table 3-1: The weekly schedule of topics for the course.

Week	Topics
1	Internet overview, FTP setup, copyright
2	HTML, XML, CSS, basic formatting, deprecated tags and attributes
3	Tables, lists, links, design concepts, hexadecimal color values
4	Visual design, graphic images, file types and formats, table layouts, web 2.0, navigation
5	Graphic image creation, background tiles and gradients, search engines, CSS
6	Framesets, inline frames, JavaScript
7	JavaScript, rollover buttons, style sheets
8	Image maps, layout with CSS, CGI
9	Forms, CGI, JavaScript form validator, accessibility
10	RSS, meta tags

Help forums were available where students could post questions to classmates and the instructor. Participation in the forums was voluntary and did not impact their grades. In 2010, forum posts and related metadata were collected from two sections taught in the fall terms of 2008 and 2009 by the same instructor. These sections comprised 49 students (39 females, 10 males). From the help forums, I collected each post's title, author, timestamp, and body.

The help forums were chosen as the focus of this study because they offered a way to assess student difficulties that were reported as they were happening, as opposed to retrospective interviews where students are asked to recall details from weeks earlier. It also provides a method of examining the issues quickly and with minimal interference in the course itself, which was appropriate given the broad and exploratory nature of this study.

A total of 747 posts, comprising 213 discussion threads, were collected. On average, students posted 15.24 times (SD = 16.52), with the most active student making 63 posts while three students did not post at all.

3.1.2. Data Analysis

I conducted a content analysis using the data collected from the help forums. Content analysis is a technique for making valid and reliable inferences "from texts (or other meaningful matter) to the contexts of their use" [Krippendorff 2004]. Codes were developed inductively from the data to categorize issues that students sought help for through the forums. These codes are summarized in Table 3-2.

Table 3-2: Codes for categories of challenges.

Category	Description
Administration	Asking questions about curriculum, instructions, and assessment
Content	Collecting, creating, and editing text, images, and multimedia
Design	Planning information architecture and visual design
Coding	Creating and manipulating HTML, CSS, and JavaScript code
Technology	Selecting, installing, and configuring technology
None	Sharing general information and providing help

I selected the thematic unit of analysis, which can flexibly range from a single sentence to multiple paragraphs. Each post was initially classified as a single instance of help seeking, but was examined further to determine if it contained multiple, distinct codes. In such cases, I divided the post into the appropriate number of thematic units.

A second researcher, Susan Wiedenbeck, and I independently coded a random 10 percent sample using this code set, attaining over 90 percent agreement and Cohen's κ of 0.841. A κ value of 0.8 or greater generally indicates the reliable application of a code set [Landis and Koch 1977]. Upon reaching this threshold with the sample, I coded the remainder of the dataset on my own. Posts classified as containing no instances of help seeking were removed from subsequent analysis.

Table 3-3: Codes for types of coding challenges.

Topic	Description
Hyperlinks	Creating links to other resources
Images	Embedding images
Image Maps	Creating image maps
Tables	Creating tables
Lists	Creating lists of items
Forms	Creating forms with input elements and actions
Frames	Creating framesets or inline frames
Backgrounds	Setting background colors, images, and tiling
Fonts	Setting font styles
Layout	Positioning and aligning elements
Functions	Defining functions, attaching as event handlers
Objects	Instantiating objects
Source Files	Managing source code at the file level

I took help-seeking instances pertaining to writing code and divided them into specific topics. This second level of codes is displayed in Table 3-3. A random sample was again coded independently by another researcher and me using this code set, reaching nearly 90 percent agreement and Cohen's κ of 0.869, at which point I coded the rest of the data.

Finally, I took a thematic analysis approach [Braun and Clarke 2006] to the content of the posts, in order to identify patterns among the issues that drove student help-seeking. Thematic analysis is an inductive method for identifying patterns or themes in qualitative data and has commonalities with grounded theory [Corbin and Strauss 1998], including a process of coding data in multiple rounds, but has more flexibility in that the generation of a theory is not necessarily the end goal.

3.2. Findings

In this section, I present the results of the analysis, supplemented with illustrative excerpts from the data.

3.2.1. Types of Barriers

The vast majority of issues students sought help for related to coding, administration, and technology. These three categories combined to make up nearly 90 percent of all help-seeking instances. Over half of all students sought help at least once for each of these categories. Table 3-4 provides a full

breakdown of the help-seeking instances and Figure 3-1 shows how they occurred on a week-to-week basis spanning the ten weeks of the course.

Table 3-4: Help seeking by type.

Category	Help-Seeking Instances		Unique Students	
	Count	Percentage	Count	Percentage
Coding	125	34.3%	25	51.0%
Administration	109	29.9%	29	59.2%
Technology	89	24.5%	29	59.2%
Content	24	6.6%	16	32.7%
Design	17	4.7%	9	18.4%

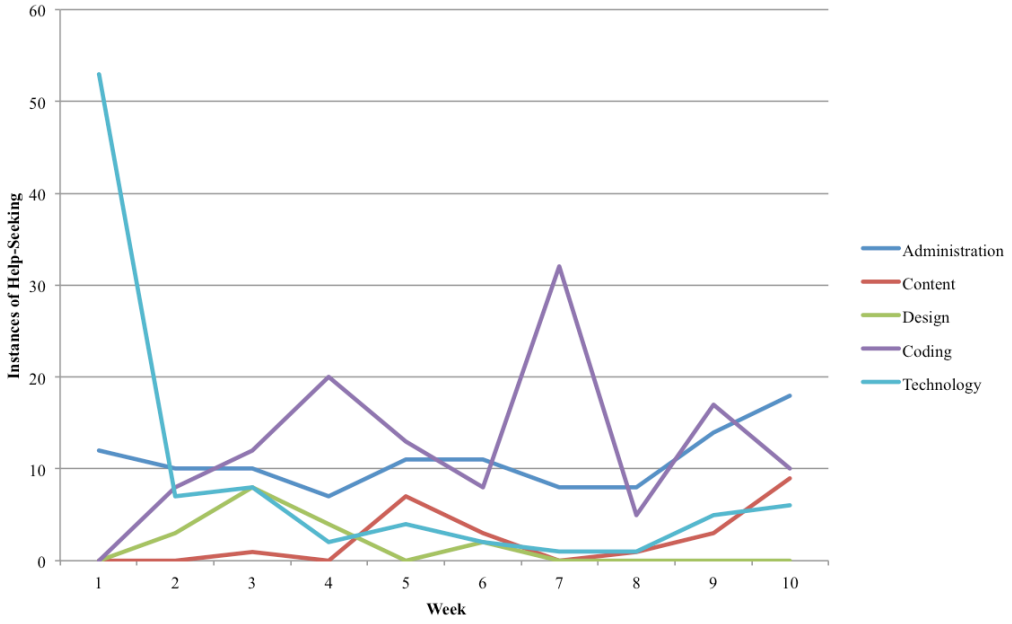

Figure 3-1: A week-by-week profile of help seeking for each category.

3.2.1.1. Coding

About one-third (34.3 percent) of help-seeking instances related to developing HTML, CSS, and JavaScript code. As shown in Figure 3-1, students began seeking help of this type in week 2, coinciding with their first exposure to

basic HTML and CSS. This activity peaked in week 7 with the introduction of JavaScript, which is not surprising given that it was the first programming language most students had ever encountered and the third distinct computing language introduced in the course. Difficulties related to coding remained substantial for the duration of the course and were precipitated by a range of topics. I provide a detailed rundown of these in Section 3.2.2.

3.2.1.2. Administration

After coding, administrational issues (29.9 percent) were most prevalent among help-seeking instances. They remained consistent from week to week and were primarily requests to clarify an assignment's requirements or instructions. I expect similar issues to arise in other courses, independent of the subject matter. Nevertheless, two instructional challenges were particularly relevant to web development.

First, one student expressed ongoing distress about the topics covered in the course, explaining that they did not follow modern web development conventions.

> "Why aren't we learning web standards? We shouldn't be using tables for website layout, or the font tag. This is no longer done. The only thing we should be using tables for is general information (small data stuff). I am ready to cry. I feel like to get an A in the class I have to do everything the wrong way." (P9)

Another student offered a counterpoint, stating that though established practitioners might not use these techniques, they were valuable for pedagogical reasons.

> "I know as a high school teacher if I started instruction where I wanted my students to end up, I would lose most of them. I often assign writing assignments that I am not going to correct for surface accuracy, in order to develop fluency. And what about reading? The point of reading instruction is not phonics or reading out loud, but one takes the students through those steps in order to develop silent reading comprehension. You are like the advanced student who needs enrichment activities... I know it must be frustrating, but I hope you can hang in there until the rest of us reach your level." (P1)

The first student responded by explaining that even as a beginner, she preferred to learn techniques that adhered to web standards from the start:

> "I am not super advanced or anything. I just know some of the web standards rules... I hope the professor gets into CSS soon. I really do not want to design my website in tables, and this global table layout makes me sick to my stomach." (P9)

This exchange underscores the difficulties that course designers and teachers face in keeping pace with the rapid changes that characterize web development practice.

A second instructional challenge faced by many students pertained to the online sharing of code. Various forms of media were used to communicate about code during the course, including videos, text documents, and forum posts. Students discovered that these media were often not well suited for this purpose. A number of students reported difficulty reading code in videos due to their low resolution. In several other instances, students reused example code from Word documents. Unfortunately, in these examples straight quotation marks (i.e., " ") had been inadvertently converted smart quotation marks (i.e., “ ”), causing syntax errors that were difficult to diagnose. Finally, students often included snippets of their code in their help forums posts. On occasion, this code was modified as a security measure by the forum software, which created confusion among the students.

3.2.1.3. Technology

Technological issues were at the root of about one quarter of help-seeking instances (24.5 percent), creating a significant hurdle at the outset of the course. Web development depends on a wide range of technological concerns beyond code, including activating shell accounts, configuring FTP programs, and managing web servers. Troubleshooting problems related to these tasks was complicated by the online nature of the course and the diversity of system

configurations used by the students. These issues in particular sapped student motivation. For instance, while attempting to connect to an FTP server and grappling with authentication errors, one student remarked:

> "I followed the same exact path you did to try to solve this problem. I still cannot connect... This kind of stuff makes me want to just drop this class. Unfortunately, I need it to graduate this quarter." (P20)

Students considered these tasks as distractions, diverting their attention from what they perceived as the main purpose of the course. In the first week of the course, a student reported:

> "I've dropped the class for now. There seems to be too many problems unrelated to what we are supposed to be learning." (P24)

After the initial technological challenges were resolved, new issues emerged on occasion in later weeks and created new impasses. For example, after using a dedicated FTP client successfully for several months, multiple students had difficulty when they attempted to configure the FTP feature built into Dreamweaver.

3.2.1.4. Content

At 6.6 percent, a small share of help seeking related to content, revolving around questions about intellectual property. Students asked how copyright and fair use applied when appropriating logos, stock photography, and streaming video from other sources.

3.2.1.5. Design

Design issues constituted 4.7 percent of help-seeking instances. These occurred mainly in the early stages of the course when design topics were introduced. Students sought advice on the visual design and information architecture of their sites, for instance figuring out which pages should be included in the main menu.

3.2.2. Coding Barriers

Next, I took a more granular look at help-seeking instances pertaining to the development of HTML, CSS, and JavaScript code. Table 3-5 shows the different topics that motivated development help seeking. For each of these topics, students faced a variety of barriers, such as selecting the correct coding elements, coordinate multiple elements together, and understanding their outputs [Ko et al. 2004]. I discuss the most common topics in turn.

3.2.2.1. Source File Management

Among development-related issues, students posted most often about organizing and accessing source code at the file level (20.5 percent). Examples of this included questions about declaring correct document types, assigning

different applications to handle source files, and affixing appropriate file extensions. A number of students operated under the misconception that because source files were assigned to default applications based on their file extensions, they could not be accessed using other code editors or web browsers. In one case, a student weighed in on the appropriateness of using the .html file extension for XHTML code.

> "I too saved them as html. I believe from the reading that it is fine for xhtml to be saved as html. It doesn't have its own extension." (P19)

The frequency of these issues provides evidence that beyond the manipulation of code at the textual level, the management of source code at the file level raises a number of new challenges for novices.

Table 3-5: Coding challenges by topic.

Category	Help-Seeking Instances		Unique Students	
	Count	Percentage	Count	Percentage
Source Files	26	20.5%	12	24.5%
Images	17	13.4%	11	22.4%
Layout	13	10.2%	10	20.4%
Functions	12	9.4%	8	16.3%
Links	11	8.7%	7	14.3%
Background	11	8.7%	7	14.3%
Tables	8	6.3%	7	14.3%
Objects	7	5.5%	3	6.1%
Lists	6	4.7%	3	6.1%
Forms	5	3.9%	3	6.1%
Frames	5	3.9%	3	6.1%
Image Maps	4	3.1%	3	6.1%
Fonts	2	1.6%	2	4.1%

3.2.2.2. Embedding Images

A significant proportion of development help seeking involved embedding images into web pages (13.4 percent). Students had encountered broken images as end users in the past, but were now in a position where they needed to diagnose and correct them.

> "I am not sure what I am doing wrong here. After I put the code in for my image upload the only thing I see on my web page is a white box with a red x in it. I have seen this so many times before on other web sites but never knew what it meant other than there should be a picture in its place." (P20)

Troubleshooting such a problem, one student remarked:

> "Try renaming homeUp.jpg to homeUP.jpg. I [think] this will fix your problem, darn case sensitive browsers! ;-) At least that is my theory at the moment." (P32)

Usually, broken images were a result of an incomplete or incorrect path to the image file. Though they were introduced in week 4, students reported difficulties as late as week 8 of the course.

3.2.2.3. Layout

Layout was another challenge faced by students, motivating 10.2 percent of development help-seeking instances. Students often had difficulty implementing the layouts they envisioned.

> "I just created the table containing my thumbnails (not clickable yet), but my tables seem to land wherever they feel like on the page... I can't figure out the rhyme or reason behind it..." (P22)

Students discovered that they could not specify an element's position by simply applying a property to that element. Instead, layout involved a great deal more complexity, determined by an interaction of rules, neighboring elements, and the context in which the page was rendered.

3.2.2.4. Functions

The use of JavaScript functions to create rollover buttons caused substantial difficulties for students (9.4 percent). In one assignment, students were provided with an example for defining rollover functions and attaching them to images as event handlers, and were required to adapt it to their websites. Assisting a fellow student who was working on this assignment, someone commented:

> "I'm not sure if this is all of what's not working, but there are some places in the script definition where you need to replace text with the actual information about your buttons..." (P45)

Students seeking help with JavaScript functions demonstrated a shallow understanding of the code, unclear on which parts to leave untouched to preserve behavior and which to modify to work with their own sites.

3.2.2.5. Hyperlinks

Closely paralleling difficulties with images were ones creating links (8.7 percent). The most frequent case was a broken link that did not point to its intended destination. Just as with broken images, students would specify an incorrect path, most often when using relative paths. Another common error was forgetting to pair an opening anchor tag with a closing tag. One student confessed:

> "I try and create both my opening and closing tags at the same time and then add the content because I have a tendency to forget closing tags. Lets not talk about the time it took me 2 days to figure out why half my page had a link (forgot a to close the link tag)." (P32)

Even after learning the common culprits for broken images and links, students at times had trouble identifying these errors within their own code. Though

relatively basic topics, students sought help for images and links into the later weeks of the course. When students were struggling to learn about more advanced topics, these difficulties added more fuel to the fire.

3.2.3. Computational Concepts

3.2.3.1. Notation

Students grappled with the formal nature of HTML, CSS, and JavaScript notation when translating their intentions to instructions. For instance, when creating links and images, students made minute errors with case sensitivity, white space, and missing delimiters. An earlier study of problem types in an introductory programming course [Robins et al. 2006] similarly found these problems of "little mechanical details" to occur most frequently.

A student sums up this challenge of formal notation:

> "If there is one thing to learn in this course, it's that the details matter when it comes to writing code... One error – one tiny typo, and sometimes your whole code ends up broken! So be careful when writing your code." (P14)

This formality contrasts with the flexibility not only of natural language, but also of popular computing systems like word processors and search engines. Students had to acclimate themselves to this inflexibility when writing and debugging code. Difficulties with notation were exacerbated by inconsistencies

in how different browsers handled faulty syntax and the issues with communicating code online discussed earlier.

3.2.3.2. File References and Paths

A second concept underlying many of the development issues were file paths, which are used to specify the location of a target within a hierarchy. Students used both absolute and relative paths when creating links, embedding images, and referencing external files.

Here, a student diagnosed a problem that another student was having while attempting to reference a file:

> "Make sure you have a directory in your class folder on the server that is called "javascript_form folder". If you don't have that directory, your code is not finding your validation script. If you put the .js file in your root class directory with all of your other .html files, then just remove the part of your code that includes "javascript_form folder" in the path." (P27)

During the course, students also interacted extensively with hierarchies and paths when managing files on their local machines and using SSH and FTP programs to navigate a server.

3.2.3.3. Nesting HTML Elements

Nesting – embedding constructs within instances of themselves – is a central feature of markup languages like HTML. Content is enclosed in pairs of tags,

and one set of tags is often contained within another. Students were prone to making errors due to the nested nature of markup, forgetting to close tags and instead treating them as sequential commands to be invoked one after another.

> "<font size="4" >Here is an example of short refrigerator story using the word My<font size="3" >" (P46)

Nesting was most prominent when constructing tables and lists. One student attempted to build a list within a list and described her difficulty escaping the sub-list.

> "Okay, my nest is a mess. ha. i see numbers everywhere that i didn't even put in. :0(Also, my nest keeps stretching to the right, and I am not sure how I managed to do this!" (P5)

Difficulties with nesting are likely to prevent substantial progress when learning web development, given that in practice, most web pages require requires writing and navigating many levels of nested HTML code.

3.2.3.4. Decomposition and Abstraction

Students encountered several cases of decomposition and abstraction while learning web development. Decomposition, breaking a program down into subprograms in order to simplify development and maintenance, was practiced when students moved CSS code that was in-line with HTML code to

external style sheets. Students were initially unclear on the purpose and process of these changes.

> "I don't understand step 7 and 8... It says to open the document from which you cut the styles. So from what I understand, after you cut and paste everything into a new document and name it whatever.css. You then re-open the old index.html document and delete the opening and closing style tags. What style tags are they referring to? Everything that we just added in chapters 7-10 we now have to delete?" (P46)

After an exchange with classmates, the student began to realize the benefits of decomposition.

> "...so we are creating only one css page that will work for ALL of our .html pages?" (P46)

Abstraction, hiding the details irrelevant to the current task, was also an aspect of the course. External style sheets and JavaScript files allowed students to readily reuse CSS and JavaScript functionality in their websites without regard for implementation details. The use of CSS selectors such as IDs and classes was also a common case of abstraction, allowing students to apply a style to a set of elements with a single command. While these cases confer benefits in

terms of organization and efficiency, they introduce new constructs and concepts that proved problematic for some beginners.

3.3. Discussion

3.3.1. Authenticity versus Complexity

Prior research has shown that contextualizing a computing course can foster motivation and engagement among students [Forte and Guzdial 2005]. Compromising that authenticity [Shaffer and Resnick 1999] can have a negative effect, as exemplified by the protests of the student who wished to follow web standards as practiced by professionals.

However, this study illustrates how a contextualized approach can sometimes result in increased complexity and must be approached with careful consideration. The breadth of barriers students sought help for underscore this point. Students learned many aspects of web development, including system administration, graphic design, and frontend programming. This broad coverage limited opportunities to dive deeply into a particular topic and spread thin the mental resources that students could apply to learning any one.

Course designers therefore must be selective in deciding which aspects of web development should strive for authenticity to increase motivation and which can be simplified to manage complexity. In this study, technological issues such as configuring software in particular started the course on the

wrong foot and the frustration induced by them seemed to outweigh the motivational effects of authentic practice for some students.

3.3.2. The Role of JavaScript

Many web development courses [Lim 1998; Mercuri et al. 1998; Reed 2001; Treu 2002; Sridharan 2004] include a programming component, and the course in this study was no exception. In ten weeks, students were introduced to a wide array of topics including three distinct computing languages: HTML, CSS, and JavaScript.

Students experienced substantial difficulties with HTML, such as creating links and lists, even in the later weeks of the course. Furthermore, students demonstrated a shallow understanding of JavaScript. Taken together, these findings suggest that instead of a web development course that sprints toward programming, a more elementary version that delves deeply into HTML and CSS may better serve some learners. While the interrelated roles that HTML, CSS, and JavaScript play to construct web pages should be discussed, reserving even the basics of JavaScript for a later course is a viable option. Especially for students without prior programming experience, a few weeks of instruction may not be a sufficient introduction to JavaScript, and to the contrary may cause confusion and instill a negative disposition toward learning to program.

3.3.3. Connecting the Web to Computing Education

In my analysis, I have identified a set of computing concepts that underlie the barriers students encountered when learning web development. They manifest primarily in HTML, which has been the subject of little research in the computing education domain, and give support for using HTML and CSS as rich contexts for exploring computing concepts.

For example, notation puts students into the mind-set of instructing computers using precisely specified language. Hierarchies and paths offer ways of thinking about familiar systems such as file systems and the web, while setting the stage for later topics like traversing the JavaScript Document Object Model (DOM). Nesting makes frequent appearances in HTML, giving students practice with navigating multiple levels of nested code. By separating content (HTML) from presentation (CSS) and behavior (JavaScript), students apply decomposition and abstraction in order to manage complexity.

For many web development courses, including the one in this study, the primary goal is not to teach computer science per se, but to arm students with practical skills for creating and maintaining websites. Nevertheless, by explicitly addressing such concepts in a web development course, educators can help students to go past the surface features and form viable mental models. The goal in these courses too is to attain generative knowledge that can be applied to web development beyond any particular technology.

3.3.4. Limitations

The content analysis of help seeking activity in an online course has two main limitations. First, by using help forums as a data source, the study is biased towards students who were willing to publicly seek help for their difficulties. Cases where students sought help through other resources, or struggled with their difficulties in solitude, are not captured. Furthermore, the study focuses on *insurmountable* barriers. Students are likely to have successfully overcome many other issues without the aid of the help forums, which may nevertheless have contributed to their frustration.

Second, this study relies on the students' interpretation of their own difficulties. Given that the students are novices, the accounts they provided in the forums had the possibility of being highly inaccurate or incomplete. In other cases, they may not even have been aware of problems they were experiencing.

Despite these limitations, my methods provided a useful first pass given the exploratory nature of the study. I was able to investigate the breadth of barriers students face, contextualizing subsequent studies that focus on HTML and CSS, and identify directions for further work.

3.4. Summary

Through a content analysis of help forums, I identified the diverse issues that acted as barriers to learning in an introductory web development course. These included issues related to coding, technology, administration, design,

and content, with coding, administrative, and technological issues combining for the bulk of them at 34.3 percent, 29.9, and 24.5 percent respectively. There was evidence that some students perceived building web pages with code as the primary focus of the course and were more accepting of difficulties related to it, while technological issues such as configuring shell accounts and FTP programs were considered secondary topics that induced frustration, even causing one student to drop the course.

Second, I explored how the barriers related to writing code and underlying computational concepts. Coding barriers made up over one-third of the issues for which students sought help. Of these, most related to writing HTML. Many of these difficulties had at their root concepts that relate to computation more generally, including notation, hierarchies and paths, nesting, and decomposition and abstraction. These findings give support to the idea that an introductory web development course, particularly aimed at non-technical students, does not necessarily need to make programming with JavaScript the focal point in order to be a subject rich with computational concepts. Even HTML and CSS provide many opportunities to develop computational literacy, justifying further study of how people learn these languages and how tools can be designed to better support them.

The severity of the technological and administrative barriers, which students perceived as secondary to building webpages and found frustrating,

and the computational richness of HTML and CSS, motivate the initial design of openHTML, a web editor for beginners.

Chapter 4
Designing the openHTML Editor

In this chapter, I introduce openHTML[1], an experimental web editor that I have designed to support learning. It constitutes the technological intervention of my design-based research approach as described in Chapter 1. In other words, openHTML's design is guided by multiple rounds of research, each of which reveal something new about how students use it and that lead to new research objectives. It also serves as a test-bed for exploring the following research question:

RQ4. How can a web editor be designed to support beginners in learning HTML and CSS?

The field of human-computer interaction has traditionally prioritized efficiency and usability as the criteria to evaluate systems. However, I have adopted a learner-centered approach [Soloway et al. 1994] for the design of openHTML, which emphasizes understanding and growth as the primary goals. I outline the design principles that motivate it, its initial implementation, and a pilot study to evaluate it.

[1] http://openhtml.org

4.1. Design Principles

The design of openHTML is guided by three overarching design principles. These principles are derived from findings described in the previous chapter and are infrastructural, transitional, and instructional in nature:

- Principle #1: Abstract away the infrastructure.
- Principle #2: Focus learning on the code.
- Principle #3: Facilitate code sharing.

In the following sections, I discuss each of these principles in detail.

4.1.1. Principle #1: Abstract Away the Infrastructure

The first principle of openHTML is to abstract away much of the infrastructural issues related to web development, including installation, configuring, and hosting. In the previous study, I found that technological issues posed significant learning barriers for students in an introductory web development course. In the early weeks of the course, students experienced difficulties with installing development software, configuring shell accounts and web hosts, and managing files locally and remotely. Students expressed frustration, viewing these issues as delaying them from coding, which they viewed to be the primary purpose of the course and where they were more willing to accept challenges.

A simplified interface for the editor itself is also a reflection of this principle, given learning enough of a complex development environment to be

productive often requires investing a significant amount of time that detracts from time spent learning other aspects of web development.

4.1.2. Principle #2: Focus Learning on the Code

The second principle is to design software that helps students focus on learning the code. In terms of the cognitive dimensions of notation framework [Green and Petre 1996], openHTML strives to enhance multiple dimensions including progressive evaluation, visibility, viscosity, but does not attempt to reduce the closeness of mapping between notation and the problem domain of building web pages. Instead, HTML and CSS are recognized as essential languages of contemporary web development practice. The previous study showed that students had significant trouble with writing code in HTML and CSS, and that these difficulties relate to various computational concepts, practices, and skills. This provides some justification for HTML and CSS as the subject of deeper study, and for exploring ways to provide greater support for learning them.

Focusing on code as a primary learning goal helps to clarify which aspects of web development can be minimized in openHTML's design and which should be emphasized for beginners. For instance, aims of openHTML include reducing the steps needed before users can start writing and evaluating code, and deemphasizing other aspects of web development such as server configuration and management.

The goal of openHTML is not to replace more full-featured and powerful code editors, but to support quick and productive experiences with coding early on before graduating to more sophisticated tools. One way to conceptualize this is to think of openHTML as scaffolding early experiences with code. Scaffolding can be described as having two goals: enabling students to achieve a goal which would not be possible without external support and (2) eventually learning to achieve that goal without support [Guzdial 1994]. The second goal suggests that scaffolding must fade away or be discontinued and allow the learner to eventually complete activities on her own. Different approaches to designing scaffolding with technologies include intelligent tools that track students' activities and intervene with help when needed [Anderson et al. 1995], tools that structure processes and elicit articulation [Owensby and Kolodner 2002], and tools that structure discourse [Scardamalia and Bereiter 1994]. These are all examples of "within tool" scaffolding – support for activities that is carefully designed into a tool.

But Puntambekar and Kolodner note that scaffolding is not necessarily a feature of a single tool; rather, it can be distributed throughout a socio-technical system [Puntambekar and Kolodner 2005]. openHTML is positioned as one part of a larger system of tools and practice that includes not only the immediate learning context (teacher and peers in the course), but also the tools and practices that learners may eventually adopt as their web-building skills become more developed. In other words, by serving as a simplified

but nevertheless fully functional web editor, openHTML itself can be thought of one element that is consciously designed to contribute to "between tools" scaffolding. Ideally, learners will eventually be able to retire openHTML and transition to more expert tools.

4.1.3. Principle #3: Facilitate Code Sharing

The final design principle was to facilitate communication related to the code. One of the findings of the previous study was that a number of barriers related to sharing and communicating about code. Within the help forums, students would request assistance from their classmates by pasting a snippet of their code. In the forums, this code was formatted as natural text making it difficult to read and to view rendered in the browser. Communication was also hindered on some occasions when the code was unwittingly modified, whether by the forums software to mitigate security concerns or by a text editor that had converted straight quotes to curly quotes.

Therefore, openHTML strives to ease accessing, publishing, and communicating about code that has been written in openHTML. I approach design as a sociotechnical problem with both technical and social components, and accordingly place great importance on the social context in which the openHTML Editor will be used and effect it can have on social communication and collaboration.

4.2. Implementation

openHTML is developed from a fork of JSBin[2], an open-source tool designed for collaborative JavaScript debugging. The decision to develop an editor instead of using an off-the-shelf solution was motivated by two factors. First, it allows me to retain control over data collection. By designing and managing the tool myself, I am able to use openHTML to gain access to the specific data of research interest while ensuring the privacy of participants. Second, it enables experimentations that explore how the design of a web editor can improve the learning experiences of beginners. By developing the tool, I have the freedom to add novel features and resolve usability issues as they are identified. In the following sections, I describe the various aspects of openHTML.

4.2.1. Web-Based

openHTML is used within any modern web browser. Users navigate to the openHTML website, where they are presented with the option to log in or sign up for an account. Once logged in, they have access to the openHTML editor, which accepts HTML and CSS code as input and renders the code in the same browser window. Saved web pages are stored on a central database.

A web-based option is beneficial for several reasons. First, it reduces the need to install and update software, which is can be heavily restricted in classroom environments, instead relying only on a web browser that is likely

[2] http://jsbin.com

already available. Second, it makes it possible to abstract away many of the infrastructural concerns, including file management, web server, and FTP. By eliminating local file management, students can easily access their data from any machine, which may be an issue in a computer lab-based class. Publishing web pages online is not a discrete action that students need to take, but can happen nearly instantaneously. Finally, openHTML can be instrumented for data collection with relative ease.

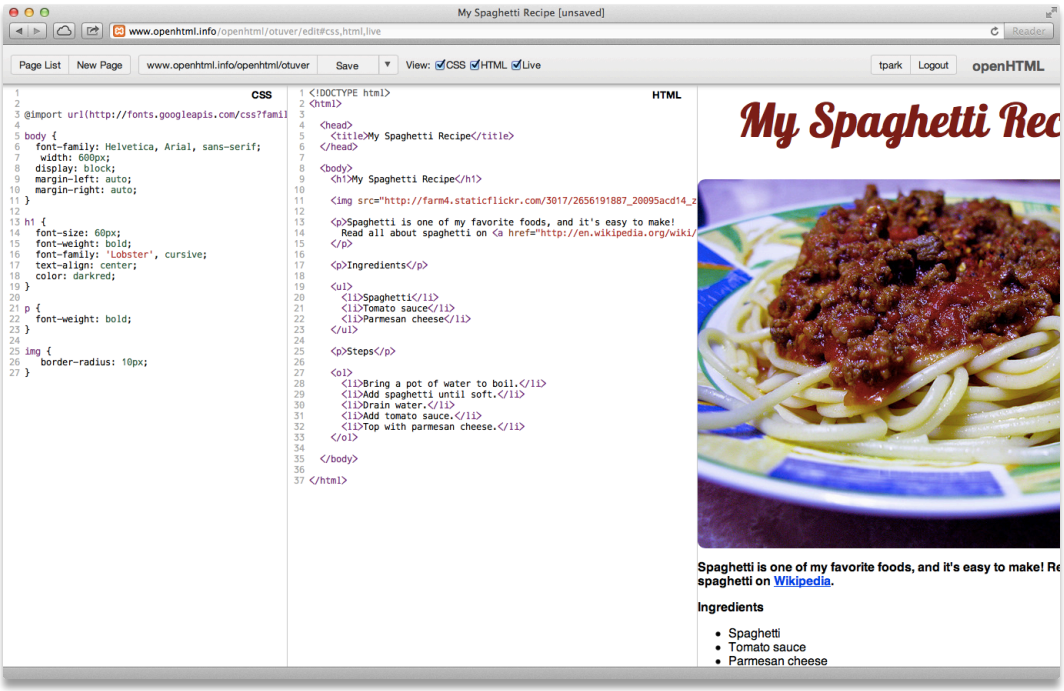

Figure 4-1: The edit mode of openHTML, with a CSS pane, HTML pane, and live preview from left to right. Several other options are provided in the toolbar at top.

4.2.2. Minimal Interface

openHTML is comprised of two primary modes: edit and page list. In the edit mode (Figure 4-1), the user is presented with three panes for CSS input,

HTML input, and a preview of the rendered webpage. The preview provides immediate feedback based on any changes in the HTML or CSS panes.

The web page can be opened in its own window by clicking the button labeled with the custom URL. These panes can be toggled on and off with checkboxes in the toolbar. Changes to the code are saved by clicking the "Save" button, and the drop-down menu beside it reveals addition options for copying and downloading the page. When viewing another user's page, the saving option is disabled, replaced with an option to copy the web page. Compared to most code editors and development environments, openHTML presents a simplified interface and a minimal number of options.

Figure 4-2: The page list mode of openHTML. A list of web pages is shown on the left, and a preview of the selected web page on the right. The same web page has been expanded to show all previous revisions.

4.2.3. Saving Revisions

By clicking the "Page List" button, a user can access the page list mode, which lists all of web pages they have created in openHTML (Figure 4-2). Users can also hover over a page name that displays a preview of the web page in the right pane, and click the "Rename" link to give a custom name to a web page. Web pages created in openHTML are not discoverable, but can be shared and accessed by other users.

openHTML implements a versioning system in order to encourage experimentation with the code. Earlier revisions of a web page are kept in the database and can be accessed by expanding a web page in the Page List. This was motivated by interviews with practicing web developers [Park and Wiedenbeck 2010] where I found that they often devised rudimentary version control systems by incrementing numbers in file names or duplicating their source files to reduce the risk of breaking their websites during development.

4.2.4. Limitations

The design decisions made during the development openHTML are accompanied by a number of tradeoffs. Limitations include support for only individual web pages rather than whole websites. While a website can be constructed from multiple web pages in openHTML, CSS stylesheets cannot easily be shared between them.

Also, the lack of a file system means that HTML documents, CSS stylesheets, images, and other assets can only be linked through absolute

paths. This runs counter to how web projects are typically organized to rely on relative paths.

By focusing on the code, there is less of an opportunity to learn many other aspects of web development, including source code and server management. Nevertheless, as the first web development tool in a broader system of between-tool scaffolds, it provides sufficient functionality to help users start learning web development, and meets the needs of the studies described in the following chapters.

4.3. Pilot Study

In the spring of 2012, I organized an after-school workshop that introduced basic web development topics through activities using the openHTML Editor. Among the goals of the workshop were to test the performance of openHTML in a multi-user session, identify usability issues with first-time users, and get a general sense for how openHTML would be used. In particular, we were interested in the issues raised when teaching younger students to build web pages.

To organize the workshop, I partnered with a local community center that provides an array of social services to disadvantaged families. One of their offerings is a 10-week after-school program where local elementary school students meet twice a week for two hours in a computer lab and learn the basics of office applications and web browsing. Compared to the typical

subjects that are covered, building pages with HTML was a new and relatively advanced topic for the program.

I led the workshop, while a fellow researcher was dedicated to making observations and recording field notes, and another split time between facilitation and note taking. In addition, I administered a pre-workshop survey to collect basic demographic information and prior experiences, and a post-workshop survey about impressions of the activities. The surveys were verbally administered to students one-on-one due to their variable proficiency with reading and writing.

4.3.1. Demographics

A total of 9 children completed the pre-workshop survey, and 7 of them, all fifth-graders, went on to participate in the workshop. Table 4-1 provides details about the students who took part. All names have been changed to protect their identity.

Table 4-1: Demographics of the workshop participants.

Participant	Age	Gender	Favorite Class
Sydney	10	Female	Science
Nathan	10	Male	Art
Alyssa	10	Female	Computers
Kiara	11	Female	Math and Computers
Gabrielle	10	Female	Computers
Alisa	11	Female	Math and Science
Kaliya	11	Female	Reading and Math

4.3.2. Activities

The goal of the workshop was to expose the participants to the idea of creating webpages with code, and have them add original content in the form of paragraphs, lists, links, and images. Activities are organized around remixing web pages. We prepared a detailed plan of activities modeled on the Scratch Curriculum Guide [Brennan et al. 2011]. Table 4-2 gives an overview.

Table 4-2: The workshop agenda.

Part (minutes)	Topic	Activities
Introduction (10)	The Web	Create name cards Discuss the web
Orientation (10)	openHTML	Create an account Explore the features
Part 1 (10)	Lists	Copy favorites page Add own favorites
Part 2 (15)	Links	Copy links Add own links
Part 3 (20)	Images	Copy image gallery Pick a theme for gallery Add images from the web
Part 4 (30)	Put it all together	Copy recipe page Add own recipes using lists, links, and images
Wrap Up (15)	Questionnaire	Administer one-on-one Others continue Part 4

4.3.3. Findings

Overall, openHTML was successful in fulfilling its role in the workshop. With minimal orientation, students were able to start using openHTML write HTML and create remixes of the webpages provided to them. Starting with templates that the children could modify instead of a blank document, and incorporating the students' personal interests were important aspects of this success.

Nevertheless, I identified two major opportunities to improve the design of openHTML based on the workshop, which resulted in the implementation of new features before the start of the next study. First, we observed that naming conventions had a great deal of power. In particular, the random hash strings used to generate web page names and URLs were perceived as "noise" that diminished ownership of the page. To address this, I instead set the default name of new web pages to "Untitled Webpage" with an option to provide them with a custom name.

Second, although students were urged to save their code early and often, in at least one instance a participant accidentally navigated away from the openHTML editor and lost changes. I consider this a catastrophic event in terms of the effect it has on student progress and motivation. Following the workshop, I implemented changes to address this, including a more prominent visual indicator when code is unsaved, and a warning message if the user attempts to navigate away from the editor with unsaved changes.

4.4. Summary

This chapter reports on the initial design and development of the openHTML editor. I outlined three principles, derived from the findings presented in Chapter 3, which guided the design of openHTML, and reported on an after-school web workshop that I conducted in part to pilot test openHTML.

Among the insights from field observations of the workshop were that custom names were an important motivating feature for participants, and that

users were susceptible to navigating away from openHTML, losing unsaved changes to their code. These issues were addressed through minor iterations on openHTML in preparation for the study presented in the next chapter.

Chapter 5
Intention-Based Analysis of Errors in HTML and CSS

In chapter 3, I explored the learning barriers students encounter in an introductory web development course. By analyzing the help forums used in the course, I was able to characterize the broad issues that students grapple with, including coding, technological, and administrative concerns. In chapter 4, I described the design of openHTML, which aims to abstract away the technological and administrative barriers to help students devote their attention to the code.

This chapter describes a study that builds on this work through a detailed investigation of the errors people make when writing HTML and CSS code. Despite the wealth of literature on programming errors in a variety of languages [Eisenberg and Peelle 1983; Anderson and Jeffries 1985; Spohrer and Soloway 1986b; Pea 1986; Pea et al. 1987; Hristova et al. 2003; Robins et al. 2006], few have applied a similar lens to HTML and CSS. Such a study informs how social and technological systems can be designed to help beginners overcome difficulties when learning the fundamentals of web development. While the study presented in chapter 3 explored this to some degree, it focused on insurmountable barriers; I did not have access to the

activity of students before they turned to the help forums, preventing me from capturing the full scope and fine detail of errors including ones they were able to resolve on their own. To address this gap, I conducted a lab study where I observed participants directly as they completed basic web development tasks. This study was guided by the following research questions.

RQ2a. What types of errors do beginners commonly make when writing code in HTML and CSS?

RQ2b. How do beginners recover from these errors?

The rest of the chapter is organized as follows. Section 5.1 describes the study protocol and participants. Section 5.2 provides a comprehensive account of the errors I observed in the tasks. Finally, Section 5.3 discusses the implication of these findings for further research and the design of openHTML.

5.1. Methods

In order to make the detailed observations necessary to understand the errors people make while constructing web pages, I conducted a laboratory-based study where I observed and recorded 20 participants as they completed a set of HTML and CSS coding tasks. A think-aloud protocol was combined with follow-up interviews, allowing me to probe the participants' intentions and understanding as they completed the tasks. Such elicitation methods are used to examine the understanding of a learner and how they reason about and solve problems [Ericsson and Simon 1993; Chi 1997]. I then used open and

axial coding processes to analyze video and screen capture data and classify the errors.

5.1.1. Participants

To capture as broad a sample of errors as possible, I recruited participants with a wide range of expertise in HTML and CSS, requiring only that they had enough prior experience with HTML to follow the task instructions. I used a variety of recruitment tactics including announcements in web development classes, flyers posted on university campuses, and a classified ad in the web design section of Craigslist. Participants were offered $20 for their time.

A total of 20 people, 7 female and 13 male, took part in the study. Their ages ranged from 18 to 47 (M=24.4) and their backgrounds included digital media, environmental science, business, and art. Two participants indicated web design as their profession; however, interviews revealed that they primarily used content management systems like WordPress to build websites, and did not practice much coding. In addition to HTML and CSS, 17 of the 20 participants reported some experience in JavaScript and other programming languages. The participants are described in Table 5-1.

Table 5-1: Participants gender, age, profession, and prior experience with HTML, CSS, and programming languages. Prior experience is self-reported on a scale of 0 (none) to 3 (expert).

P	Gender	Age	Profession	HTML	CSS	Prog
1	Female	19	Student (Digital Media)	••	••	•
2	Female	20	Student (Digital Media)	••	••	••
3	Male	20	Student (Computer Science)	••	••	•••
4	Male	20	Student (Business)	•••	•••	•
5	Male	19	Student (Information Systems)	••	•	•
6	Male	25	Student (Information Science)	••	••	•
7	Female	22	Student (Digital Media)	••	••	•
8	Male	23	Visual Effects Art	•••	•••	••
9	Male	23	Student (Digital Media)	•••	•••	•••
10	Male	20	Student (Computer Science)	•••	•••	•••
11	Female	29	Student (Environmental Science)	•	•	•
12	Male	20	Student (Information Systems)	•••	•••	•••
13	Male	36	Law	•		•
14	Male	22	Student (Information Technology)	•	•	•
15	Male	41	Web Design	•	••	••
16	Female	19	Student (Art)	•	•	
17	Female	47	Web Design	•	•	
18	Male	21	Student (Business)	•••	•••	••
19	Female	24	Student (Education)	•	•	
20	Male	18	Student (Business)	•	•	•

5.1.2. Protocol

In order to provide a consistent experience for all participants and to record the sessions, participants were invited to a usability lab and asked to complete a set of five coding tasks involving HTML and CSS. The tasks were preceded with a questionnaire and brief interview that collected information on demographics and prior experience. Participants were asked to rate their own expertise with HTML, CSS, and any programming languages as no experience (0), beginner (1), intermediate (2), or advanced (3).

The first iteration of openHTML was used to complete the tasks. My design approach, which was to start with a barebones environment and follow an iterative process to extend its functionality, made the first version of

openHTML an ideal environment for the study since it lacked the bells and whistles of more complex editors that were irrelevant to the tasks. Moreover, all participants were equally unfamiliar with the tool. Participants were given an orientation with openHTML before the study began.

For each coding task, I gave participants printed instructions containing multiple sub-goals and an image depicting the expected output of the rendered web page. I asked them to complete tasks to the best of their ability using whatever resources they would normally use including web searches. I explained the think-aloud protocol and encouraged participants to vocalize their thought processes as they completed the tasks. A maximum of 30 minutes was provided for each task, and participants were allowed to end a task at any time. After each task, I asked follow-up questions to clarify their understanding and intent. Sessions were video recorded. Participants averaged approximately 38 minutes of coding activity (ranging from 13 to 57), totaling over 12 hours of video data combined.

5.1.3. Tasks

Participants completed five tasks that involved writing or modifying HTML and CSS. I piloted the tasks to ensure that they could be reasonably completed in 10 to 15 minutes. The tasks were also designed to provide broad coverage of HTML and CSS constructs, setting a low floor and steadily increasing in sophistication. For all of the tasks, the HTML pane was seeded with boilerplate code for the HTML5 document type declaration and html, head,

title, charset, and body tags; additional code was seeded for Task 3 requiring the code to be extended, and Task 4 requiring three bugs to be fixed. The tasks are summarized in Table 5-2.

Table 5-2: The coding tasks.

Task	Requirements
1	Create a heading Create a paragraph Create an ordered list Created an ordered sub-list
2	Embed a hyperlink Embed an image Hyperlink the image
3	Center the text alignment in the provided table Set the background of the pro rows to green and the con rows to red Set the hyperlink text color to green Set the hyperlink text color to red on hover
4	Find and fix bug 1: broken image Find and fix bug 2: unclosed tag Find and fix bug 3: unmatched CSS selector
5	Create a container div Center the container Create a sidebar div Position the sidebar on the right side of the container

5.1.4. Data Analysis

I worked with another researcher, Ankur Saxena, to code the video data in three iterative rounds using the usability testing software Morae. I did not apply a pre-determined codebook; rather, the goal was to use the coding exercise as a way of inductively developing an inventory of errors.

In the first round of coding, every occurrence of a syntax or semantic error was marked. In line with Youngs' definition of programming errors [Youngs 1974], I defined errors as code written by the participant with invalid syntax, or that resulted in actual or potential output (web page rendering) that was not

desirable according to the task or the participant's interpretation of that task. This definition of an error required not only the interpretation of code but of the participant's intent, in order to identify syntax and semantic errors. A total of 791 errors were identified in this initial round.

Table 5-3: The coding scheme for errors.

Code	Values
Level	skill, rule, knowledge
Type	typo, obsolete construct, css selector, etc.
Resolution	resolved, unresolved, bypassed

In the next round of coding, I classified the identified errors based on the emergent coding scheme (Table 5-3). To produce a robust classification of errors, I examined not only the errors themselves, but also the context and response to the errors in a process similar to axial coding from grounded theory [Corbin and Strauss 1998] and informed by an understanding of errors as driven by skills, rules, or knowledge deficits.

This scheme was informed by the skills-rules-knowledge framework, a hierarchical model of human behavior organized in terms of cognitive effort [Rasmussen 1983]. A thorough treatment of the skills-rules-knowledge framework is provided in [Reason 1990], which informed the analysis of cognitive breakdowns at the root of each error:

- Skill-based behaviors, such as typing, are "sensory-motor performance[s] tak[ing] place without conscious control as smooth,

automated, and highly integrated patterns of behavior." Errors at this level are the result of unintended actions from physical slips, inattention, or mode confusion. Norman [1981] offers an extended account of errors that occur at this level.

- Rule-based behaviors are comprised of "a sequence of subroutines in a familiar work situation... typically controlled by a stored rule or procedure." Rule-based behavior is guided by conscious and goal-oriented planning. Errors here result from intentional actions driven by the application of bad rules or the misapplication of previously good rules to exceptional circumstances.
- Knowledge-based behaviors occur at a higher conceptual level when a person faces an unfamiliar situation that necessitates ad-hoc experimentation and problem solving. Errors at this level, or more aptly "breakdowns," result from an incomplete or inaccurate understanding of the situation. Typically, multiple errors are made in succession, entwined with experimentation and information searches.

In order to determine the appropriate level, I relied not only on observed coding behavior but other cues, including the participants' verbalizations while coding, their reactions when errors were detected and resolved, and, importantly, their strategies for resolving them. For instance, a web search could be used to remember complicated syntax, suggesting rule-based

behavior, or for just-in-time learning of a broader topic [Brandt et al. 2009], typical for trying to address a knowledge-based breakdown. Table 5-4 outlines the heuristics that were applied during this part of coding.

Table 5-4: Heuristics used to classify errors as occurring at the skill, rule, or knowledge-based levels of performance.

	Skill	Rule	Knowledge
Types of Activity	Quick routine actions	Simple if-then rules	Slow information seeking
Control Mode	Mainly by automatic processes	Mainly by automatic processes	Conscious processes
Perception	Feedforward	Feedforward	Feedback
Intention	Unintended actions	Intended actions	Intended actions
Solution	Indicator of existence	Brief explanation	Extensive learning

I developed a detailed taxonomy of error types at each of the three levels through an inductive, data-driven process. At the skill-based level, errors tended to be simple and arose from a mismatch between intention and action, such as forgetting to type a semicolon. At the rule-based level, errors became more complex, for example using an attribute that has been deprecated. Knowledge-based level errors proved to be the most complex, for instance a lack of understanding of the positioning model, a central aspect of web development that determines how elements are laid out in relation to each other on the web page. I also coded whether errors were ultimately resolved, unresolved, or bypassed in favor of an alternative approach. The other researcher and I reconciled disagreement through further discussion. In the second and third rounds of analysis, I reviewed the codes with the second researcher and made refinements where necessary.

5.2. Findings

In this section, I present the results of the analysis. I start with an overview of the observed errors including how they related to the skills-rules-knowledge framework. I then discuss errors at each level of the framework in more detail, starting with an illustrative vignette and finishing with a detailed catalog of errors by type, frequency, and resolution.

5.2.1. Overview of Errors

A total of 791 errors were identified in the analysis. Participants averaged 39.6 errors per session (including all tasks) (SD=15.0), ranging from 15 (P14) to 63 (P9). The percentage of errors they left unresolved ranged from 1.7 percent (P3) up to 38.6 percent (P6). Breaking down the activity by task (Table 5-5) shows that task duration and errors made was generally higher for tasks 3 and 5.

Table 5-5: Task completion time in minutes and error count for each task.

Task	1	2	3	4	5
Time (SD)	5.42 (4.61)	5.94 (3.96)	9.40 (5.56)	6.51 (4.62)	10.95 (5.69)
Errors (SD)	7.55 (4.75)	6.70 (4.26)	7.85 (5.44)	4.20 (4.12)	13.25 (8.16)

Based on the analysis, 70.9 percent of errors occurred at the skill-based, 16.9 percent at the rule-based, and 12.1 percent at the knowledge-based levels. The overall percentage of errors that produced invalid syntax was 69.2 percent, and this was remarkably consistent across skill-based (67.3 percent), rule-based (70.1 percent), and knowledge-based errors (69.8 percent).

Overall, 83.9 percent of errors were resolved, although this is heavily skewed by the number of skill-based errors that were made. A scant 4.3 percent of skill-based errors were ultimately unresolved, while 39.6 percent of rule-based and 52.1 percent of knowledge-based remained so. This is depicted in Figure 5-1.

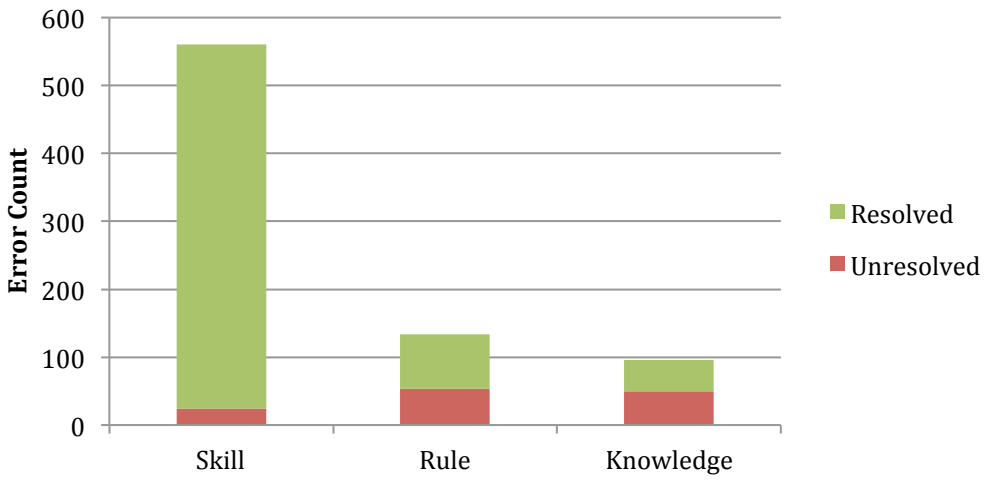

Figure 5-1: Error count and resolution for skill-based, rule-based, and knowledge-based errors.

The number of skill-based errors far exceeded rule- and knowledge-based errors, but participants resolved the vast majority of them by the end of the task. In comparison, knowledge-based errors were infrequent, but were accompanied with substantial episodes of problem solving and often unresolved. These findings align with what has been observed in other domains [Reason 1990] and reflect qualitative differences in the nature of the errors. I dive into these differences in the following sections.

5.2.2. Skill-Based Errors

5.2.2.1. A Vignette

Participant 15, a 41-year-old web designer, is working on embedding an image in Task 2, which instructs that he include an alt attribute that specifies alternate text when the image cannot be found. The correct code should resemble the following:

```
<img src="http://constitutioncenter.org/images/ui/logo-
ncc.gif" alt="My Image" />
```

However, Participant 15 forgets the opening quote in the alt attribute's value.

```
<img src="http://constitutioncenter.org/images/ui/logo-
ncc.gif" alt="My Image />
```

He realizes something is amiss when the image does not render as expected in the preview pane. After carefully examining the code for a minute, scanning it repeatedly, he finally spots the source of the error, exclaiming, "Oh! That's it." Despite successfully enclosing values with quotes numerous times before and after this instance, he makes a skill-based error here, whether due to cognitive overload, inattention, or the slip of a finger. In this case, merely becoming aware of the missing quotation error was sufficient information to fix it.

5.2.2.2. Classification

At the skill-based level, errors were caused by unintentional actions, such as a mental or physical slip, during highly routine activities. Six types of error were observed at this level. They include typographical errors, forgetting to close

paired constructs, forgetting a delimiter, accidentally mixing HTML and CSS syntax due to mode switches, confusing semantically similar constructs such as titles and headers, and misplacing code in a location other than intended.

Skill-based errors could be observed when there was a mismatch between a participant's intentions and their actions. Participants demonstrated that they understood how to complete a task, either by successfully completing similar tasks previously or vocalizing their intent. However, they did not carry out the action as intended due to their attention being pulled in other directions.

Participants were generally capable of resolving these errors if they recognized they had been made. Furthermore, the majority of them not only resulted in syntax errors but had a detrimental impact on the rendered webpage, making their presence highly salient. The combination of these two factors resulted in a high rate of resolution for skill-based errors.

The lone exception was unclosed pair errors, more than half of which were left unresolved. This can be explained by the fact that unlike other skill-based errors, they had little impact on the rendered output of a webpage despite producing invalid syntax. Nevertheless, openHTML's syntax highlighting and automatic nesting, which would behave unexpectedly in the presence of unclosed pair errors, tipped off some observant participants that an error had been made.

Table 5-6: Skill-based error types.

Error Types	Description	Examples	Total	Unresolved

Typographical Errors	Physical slips in the typing process, as with tags, properties, and values	`</blcokquote>`
		`bacground-color`
		`width: 100ps;`
			495	7

Unclosed Pairs	Forgetting to close paired constructs or characters, such as tags, quotes,or braces	`<h1>Note`
		`<img src="foo.png>`
		`a { color: red;`
			27	15

Missing Delimiter	Forgetting other symbols that delimit data, such as semicolons in CSS rules and the hash symbol in hex values	`h1 {`
		`  font-size: 20px`
		`  color: 0000FF;`
		`}`
			6	1

Mixed Languages	Accidentally applying HTML syntax to CSS, or vice versa	`div { color=blue; }` `<div color=red;>`	12	1

Confused Similar Constructs	Mixing up semantically similar constructs	`title & h1`
		`color & background-color`
		`class & ID`
			17	0

Misplaced Code	Accidentally pasting code or typing in the wrong location	`<a` `href=""http://foo.com></a>`	4	0
			561	24

5.2.3. Rule-Based Errors

5.2.3.1. A Vignette

Participant 5, a 19-year-old college student, is progressing with Task 5, which requires him to create multiple div elements in HTML and style them using CSS. To this end, he assigns the elements classes in HTML and selects those classes in CSS. These are skills that he successfully used earlier to complete Task 4. He sets the class of one div to "2" and assigns the class a blue background color. To his surprise, the div does not change color. Though he does not realize it, the cause of this error is that class names cannot begin with a number.

This episode is illustrative of rule-based errors. Participant 5 is familiar with the general rule for how to set classes in HTML, and how to select them in CSS. But he comes up against an unfamiliar exception in how classes can be named. Although he is able to overcome this, he expends significant time and effort to do so, and in the end may still not fully comprehend the source of the error. In this case, the simple elaboration of a known rule is likely sufficient for resolving the error.

5.2.3.2. Classification

At the rule-based level, errors occurred during relatively routine activities as with skill-based errors. However, they were caused by the intentional and consistent, but faulty, use of familiar rules. Rule-based errors were most diverse in their types, which makes sense given they occur when encountering all manner of edge cases where more general rules start to break down. Particularly at this level, the error types are not comprehensive, but simply representative of the errors I observed in our study. I expect that countless others can be added to this list, and that the list is likely to change as standards evolve.

Common causes for rule-based errors included using outdated elements from earlier versions of HTML, extending the general markup syntax to void elements, and failing to recognize constraints in how certain elements like inline, list, or style elements can be placed in the code.

Compared to skill-based errors, rule-based errors had a greater tendency to be unresolved. They often resulted in invalid syntax and were an opportunity for participants to refine their understanding of the edge cases and produce more robust code. Despite this, errors rendered properly in the output, likely leading to an assumption that all was well with their code.

Table 5-7: Rule-based error types.

Error Types	Description	Examples	Total	Unresolved
Obsolete Construct	Using elements, attributes, and properties that once were valid but are no longer support	`<center></center>` `<font color="red"></font>`	12	9
Invalid Construct	Using elements, attributes, or properties that do not exist	`<sidebar></sidebar>`	12	3
Valid But Unsuitable Construct	Using a familiar but cumbersome element, instead of a simpler, more suitable one	`<p>1. First item</p>` `<p>2. Second item</p>`	3	1
Misidentified Construct	Using the wrong name to reference a construct	`font-color instead of color` `align instead of text-align`	24	6
Hyperlink Concepts	Confusing the hyperlink content and destination	`<a href="Google">` `http://google.com</a>`	7	0
Resource Paths	Errors in constructing the path to a resource such as an image or web page	`http:icer-conference.org` `absolute vs. relative paths`	1	0

Lists and List Items	Giving a list element a child other than a list item	`<ol>`
		`  <p>Item one</p>`
		`</ol>`
			13	11

Ordered List Numbering	Manually numbering ordered list items, which are automatically numbered	`<ol>`
		`  <li>1. Item one</li>`
		`  <li>2. Item two</li>`
		`</ol>`
			9	3

Void Element Syntax	Errors with empty elements, which are solitary instead of paired like typical elements	`<img src="image.png"></img>` `</ br> instead of  `	11	9

Style Element Placement	Using style elements outside of head without the scoped attribute	`<body>`
		`  <style>`
		`  h1 {font-color: red;}`
		`  </style>`
			3	2

Inline Style Syntax	Syntax errors while writing CSS code inline with HTML	`<h1 color: red;>Header</h1>`	6	1
Color Hex Values	Misformatting hexadecimal values, which require a hash and 3 or 6 digits	`color: 0000FF;`	2	0
Missing Units	Missing required units on CSS values	`margin: 40;`	3	2
Naming Identifiers	Starting a class or ID name with a numeral or other invalid character	`<div class="1"></div>`	3	1

Mistargeted Style	Applying style to wrong element due to a logic error	`table {`
		`  text-align: center;`
		`}`
			4	0

Overriding Rules	Inadvertently overriding rules due to the CSS cascade	`a:hover {`
		`  color: red;`
		`}`
		
		`a:link {`
		`  color: blue;`
		`}`
			1	0

Invisible Elements	Missing content, height, border, or background, causing an element to not be visible as expected	`<div style="width: 500px;"></div>`	8	2

Centering Block Elements	Inability to center block elements, which requires setting a width, and left and right margins to auto	`<div align="center">Not</div>`
		
		`div {`
		`  text-align: center;`
		`}`
			4	1

Collapsing Margins	Undesired collapsing of vertical margins in adjacent or nested elements	`<div style="margin: 10px;">`
		`</div>`
		
		`<div style="margin: 20px;">`
		`</div>`
			3	2

Non-unique IDs	Using an ID multiple times in a document	`<div id="section1">`
		`  <h1 id="section1">1</h1>`
		`</div>`
			1	0

Comment Syntax	Syntax errors for comments in HTML and CSS	`// HTML comment` `/ CSS comment`	4	0
			134	53

5.2.4. Knowledge-Based Errors

5.2.4.1. A Vignette

In Task 3, Participant 20 is asked to style the text in each cell of the provided table by aligning it to the right. He begins by opening up a website he used in an earlier task to reference the syntax of common tags. On the website is a section called "Alignment tags," which includes the following deprecated code for aligning text to the right.

```
<P ALIGN=Right>your text
```

He copies the code, pastes it into his own, and modifies it to create the following:

```
<table><ALIGN=Right> <tr><td>Pro: Low Unemployment</td></tr>
```

Observing that this code doesn't have the desired effect, he tinkers with the placement of the align code, moving it inside the td element without any success. He moves it again, this time between tr and td tags. It still doesn't work.

Participant 5 searches the web with a query for "align right table". The top result is a question and answer site, where he spots code using the align attribute:

```
<tr><td>..</td><td align='right'>10.00</td></tr>
```

He copies and pastes part of this HTML snippet into the CSS pane, resulting in the following code.

```
table { align='right'
}
```

The style is still not taking effect, so Participant 20 spends the next minute carefully inspecting his code. He adds dummy text between the tr and td tags, confirming that it has some effect on the live preview before quickly deleting it. Next, he conducts another query for "css align right table" and scans three different pages. He comments to the researcher, as he points to the code he had added to the CSS pane, "It said to put this in here. Almost exactly like that." He continues with several more web searches, using general queries like "using css" and "apply css attribute". After much tinkering with the code, Participant 20 gives up six minutes after he started moves on to the next part of the task.

Participant 20's struggles with Task 3 involved the fundamentals of HTML and CSS, and are representative of errors at the knowledge-based level. He has significant knowledge gaps in the structure of an HTML tag, demonstrates persistent confusion between HTML and CSS code, and engages in lengthy web searches. At this level, resolution requires substantial learning.

5.2.4.2. Classification

At the knowledge-based level, breakdowns were caused by a severe deficit of knowledge relevant to completing a task. During these breakdowns, participants consciously engaged in just-in-time learning, characterized by extended cycles of conducting web searches and tinkering with the code. In

more than half of the cases, participants were not able to resolve these breakdowns due to their scope and the time limits of the study.

16 of the 20 participants made at least one knowledge-based error, but they tended to be concentrated in certain participants. Among the four participants who had 10 or more knowledge-based errors, three (P11, P13, P20) reported minimal prior experience that was reflected in their performance. However, one of these participants (P15) reported intermediate experience with CSS and programming languages, which may indicate the difficulty of beginners in assessing their own ability as well as the notion that expertise does not always follow experience. Interestingly, there was no correlation between the number of knowledge-based errors made and either skill or rule-based errors, and these four participants were in the middle of the pack for the other types of errors.

Knowledge-based errors made up only a few types, but related to central models governing HTML and CSS that broadly integrated many topics. HTML fundamentals and CSS fundamentals, which relate to the basic syntax and semantics of the two languages, were most common, reflecting the expertise of participants and the nature of the tasks. These breakdowns were usually represented by basic syntax errors. On the other hand, during the other knowledge-based breakdowns, semantic errors tended to prevail.

Table 5-8: Knowledge-based error types.

Error Types	Description	Examples	Total	Unresolved
HTML Foundations	The basic syntax and semantics of HTML elements, including tags, attributes, and values	`<align="right">Sidebar</align>`	39	17
CSS Foundations	The basic syntax and semantics of CSS rule sets, including basic selectors, properties, and values	`div: color: red;`	26	12
CSS Selectors	Advanced or compound CSS selectors	`.div > #element`	23	15
Box Model	Styling the dimensions of elements using properties of the box model	`width, height, padding, border, margin`	2	1
Positioning Model	Styling the position of an element within the document's flow	`position, float, top, right, bottom, left, display`	6	5
			96	50

5.3. Discussion

In the following sections, I discuss the implications of my findings in terms of web development education and designing tools for beginners.

5.3.1. Triaging Errors

This study maps the landscape of errors people commonly make in HTML and CSS. In addition to observing how the errors manifest in the code, I was able to analyze the cognitive sources of the errors by applying the skills-rules-knowledge framework. Considering the participants' understanding and intent in this way suggests the different types of support needed to help overcome

them. Earlier studies have similarly accounted for intention when diagnosing novice programmer errors [Johnson and Soloway 1984] and developing plausible accounts for the origin of bugs [Spohrer and Soloway 1986b].

At the knowledge-based level, I identified several broad areas fundamental to web development with which participants struggled. Participants were conscious of the breakdowns they were experiencing and engaged in extensive episodes of web searches, tinkering, and other deliberate actions to resolve them. The topics that knowledge-based errors related to suggest different conceptual plateaus on which people are operating. Prior to HTML and CSS foundations, people have only acquired bits of meaning about unconnected code. Upon learning these foundations, they are able to construct the atomic building blocks of web pages: HTML elements and CSS rule sets. Through CSS selectors, they learn how CSS styles can target HTML elements. Finally, through the box and positioning models, they learn how elements and styles can be combined to construct sophisticated web pages.

At the rule-based level, errors give insight into the misconceptions people have about HTML and CSS (Table 5-7). In many cases, the participants were not aware that they were making rule-based errors. At this level, participants applied rules with intention that, while producing errors, accorded with their state of knowledge. These are rules that have served them effectively in the past, but were not workable in the exceptional circumstances or changing contexts. Within the CS education domain, novices' misconceptions have

been studied in a variety of contexts for their role in generating programming errors [Bayman and Mayer 1983; Bonar and Soloway 1985; Sanders and Thomas 2007; Kaczmarczyk et al. 2010].

Finally, skill-based errors were caused by physical or mental slips. Though seemingly minor, skill-based errors sometimes cascaded into other errors and resisted correction because participants overlooked them and directed their debugging efforts at aspects of the code with which they were less familiar.

Skill-based errors are unintentional, requiring information about their existence and location. Rule-based errors require relatively simple explanations of the errors. At the knowledge-based level, substantial learning involving multiple topics is needed. Errors at each level are best addressed by different approaches, due to differences in intentionality and knowledge at their root.

5.3.2. Feedback that Harms and Helps Understanding

This study gives insight into how web development tools can be designed to provide better support for detecting and fixing errors. At all levels, feedback provided by the web editor's live preview panel was observed as instrumental in detecting and resolving errors, complemented with subtle cues from the syntax highlighting and automatic indentation in the code panes. As participants typed their code, they were able to immediately test it as the page rendered in real time.

However, as the primary mode of feedback, the live preview could also be detrimental. Browsers are tolerant of errors, and do their best to render

HTML and CSS code even when it is riddled with bugs. When a beginner writes code that has many errors but still renders as desired, they receive positive feedback. The errors remain latent and unresolved, reinforcing faulty understandings that can become difficult to later unlearn. In several cases, code rendered as intended in the preview pane despite the presence of an error. In other cases, similar errors caused a problem with the rendering, leading to inconsistent feedback.

The analysis revealed that approximately 70 percent of errors that participants made at all three levels of behavior produced syntax errors. This suggests that current HTML and CSS validators are capable of detecting that an error has been made in the majority of cases. However, the degree to which an error message reflects the source of misunderstanding and highlights a path forward can vary considerably. For example, in the event that someone has forgotten to close an HTML element, the validator might appropriately alert that the element is unclosed. In other cases, syntax errors may be symptomatic of a more distant or deeper difficulty. Nevertheless, these cases may also present an opportunity to make inferences about the source of difficulties based on the pattern of errors over time.

Beyond syntax errors, linters apply heuristics that identify common semantic errors that a validator might not catch. For instance, the uniqueness heuristic [Ko and Wobbrock 2010] states that an identifier, such as an HTML ID or class, that occurs only once in the code is likely symptomatic of an

error. The taxonomy of HTML and CSS errors suggests a number of additional warning signs for semantic errors that can be detected in the code. Examples include an element that is assigned visual styles but that is not visible due to having a height of zero, or a border that has been assigned a width and color but does not display due to the style type being unspecified.

Many editors are already adept at helping with skill-based errors. When knowledge-based breakdowns occur however, a validator or linter will often reply with a flood of error messages. This feedback may be counter-productive, overwhelming, intimidating, or otherwise discouraging beginners. Instead, they may be best served by being directed to substantive learning resources. Where validators may have the greatest impact is in providing support for rule-based errors. With these errors, the learner already has a significant base of knowledge, and if properly designed, can learn to overcome them with relatively little guidance.

5.3.3. Interpreting Errors in Natural Settings

In this study, I directly observed the coding behavior of participants, gaining a richer view of coding activity than would have been possible through code inspection or retrospective interviews. Changes in the code were accompanied with verbal articulations, facial expressions, gaze changes, web searches, and even different postures, all of which contributed to interpreting and classifying the errors that were made.

However, there were significant tradeoffs with this approach. The data collection and analysis was time consuming, limiting the number of participants and the diversity of the coding activities could be observed. There is also the question of ecological validity—that is, how closely the coding activity of a diverse set of participants in one-hour sessions correlates to what may be observed among students during a course or other authentic learning experience. Additionally, my presence in observing these tasks and facilitating the think-aloud protocol is likely to influence findings; in computing tasks completed in the presence of another person, gender has been identified as a significant factor for the level of reported stress and performance [Huff 2002].

One avenue for overcoming these limitations is by remotely tracking the coding activity of students as they progress through a web development course. In addition to the coding activity itself, this study lends support for syntax errors as a window into many of the difficulties that students face when learning HTML and CSS. Novel heuristics could also be devised for detecting semantic errors in the code. Lastly, as demonstrated in an earlier study [Brandt et al. 2010], help-seeking activity such as web queries can also be remotely logged and provide data that reflects the mindset of the learner.

5.4. Summary

In this chapter, I have reported on a lab study of errors that people make when writing HTML and CSS code. Over 12 hours of video data was

recorded as the participants completed five coding tasks and analyzed to identify the errors they made.

First, this study contributes a catalogue of errors (RQ1). A total of 791 errors were observed and classified into 32 categories, providing an empirical basis for common HTML and CSS errors. This is one of the first and most substantial investigations of errors in basic web development to date. I also examined the cognitive source of these errors. By using a think-aloud protocol and applying the skills-rules-knowledge framework, I was able to probe the intent of the participants' actions. From this analysis, I found that skill-based errors, characterized by unintentional actions such as typographical errors or physical slips, occurred with greatest frequency (70.9 percent of all errors). Rule-based errors, which stemmed from the intentional application of misconceptions, were less common (16.9 percent). Knowledge-based errors (12.1 percent), which related to severe knowledge gaps, were least common of all.

Finally, I analyzed the resolution of these errors (RQ2). I found that the vast majority of skill-based errors were resolved (95.7 percent). On the other hand, participants had less success in fixing rule-based (60.4 percent) and knowledge-based (47.9 percent) errors. Although knowledge-based breakdowns were most severe in their scope, participants were conscious of the difficulties they were having and engaged in deliberate information

gathering, experimentation, and problem solving to address them. In contrast, participants were often unaware of the rule-based errors they had committed.

Chapter 6
Analysis of Syntax Errors in a Web Development Course

In this chapter, I turn my attention to active students in an introductory web development course. These students are at a critical stage in their development of computational literacy, often possessing at most a rudimentary understanding of computation and the web through their experience as end users, but with an opportunity to engage in deeper learning.

One aspect of computational literacy that many students in a web development course encounter for the first time is learning to read and write code. Novices often have considerable difficulty with the exacting nature of formal computing languages, and juggling the precise syntax of a new language with higher-level concerns about semantics and design. Yet few studies have examined such difficulties with the HTML and CSS, and fewer yet have done so in the context of students encountering these languages for the first time. A better understanding of the errors students make using HTML and CSS during a course and how they resolve them can inform educators and tool designers, particularly in formal learning contexts. This study explores this with the following research questions:

RQ2a. What types of HTML and CSS syntax errors do students commonly make as they progress in an introductory web development course?

RQ2b. How well do students resolve these HTML and CSS syntax errors?

RQ2c. What role does validation play in resolving HTML and CSS syntax errors?

RQ3. What computational concepts and skills do beginners engage with when learning HTML and CSS?

This study builds on the work described in the previous chapter in several ways. By examining the initial weeks of an introductory web development course, I was able to observe beginners during their first sustained experiences with HTML and CSS, rather than people with mixed levels of experience in a single session. In the learning sciences, microgenetic studies on the order of weeks have proven useful for providing an in-depth view of the dynamic process of learning [Siegler 2006]. This study focuses primarily on syntax errors, which can be readily detected by existing validators and which I found in the previous study to be present in the majority of coding difficulties.

Methodologically speaking, I made use of remote log analysis by instrumenting openHTML and deploying it in a course. Compared to direct observation, some of the rich context is lost. The data is often at a much

lower level (e.g., keystrokes) and voluminous, requiring analysis techniques to interpret higher-level patterns within them [Guzdial 1993]. However, this approach scales more effectively, making it possible to analyze the coding activity of more students, inside and outside of the classroom, for greater lengths of time. Findings from this approach can also reduce observer bias and hold greater ecological validity. Finally, a remote approach has direct applications for the design of data-driven web development tools and online learning systems.

The chapter is organized as follows. Section 6.1 describes the methods used in this study, including a description of the course, demographics, iterations on the design of openHTML, and the study design. Section 6.2 reports on my findings. Finally, Section 6.3 discusses the implication of these findings for further research and design.

6.1. Methods

6.1.1. Course Description

This study was conducted during the Fall 2012 and Spring 2013 semesters of a web development course for undergraduate college students. The course is offered by a mid-sized private New England university and introduces the fundamentals of frontend web development. The course was chosen for the study due to the teacher's commitment to adopt openHTML for a significant part of it. Both general education students and CS majors can take the course,

either as a standalone or toward an undergraduate major or minor in web design and development. The course draws many non-CS students majoring in graphic design, audio engineering, or the humanities. Demographic details for the participants follow in Section 6.1.4.

By the end of the course, students were expected to be able to design and implement basic websites using HTML, CSS, and a small amount of JavaScript. Students were also taught to follow a systematic, user-centered design process and author code that complies with web standards. Although minor adjustments were made between the two semesters, Table 6-1 shows a representative schedule with topics and assessments organized by week.

Table 6-1: The weekly schedule of topics and assessments for the course.

Week	Topics	Assessments
1	Internet and web basics	
2	Structural basics	Lab 0
3	Structural basics, links	Lab 1, Lab 2,
4	Introduction to CSS, visual elements, and graphics	Lab 3, Assignment 1
5	Wireframing, mockups, web design best practices	Assignment 2
6	More CSS, page layouts	Assignment 3
7	Page layouts, uploading to servers	Assignment 4 Begin Project 1
8	Midterm Exam	
9	User-centered design, Usability testing	Project 1 Due
10	HTML5 structural elements, tables	Begin Project 2
11	Designing navigation, sitemaps, forms	
12	Media, interactivity, and advanced selectors	Assignment 5
13	JavaScript basics, jQuery	
14	Accessibility evaluation	
15	Publishing, hosting, and search engine optimization	
16	Final Exam, Presentations	Project 2 Due

The teacher had a large hand in designing the course, which includes activities of varying scope. Labs were small coding tasks to be completed primarily in class. Assignments were mid-sized homework based on end-of chapter

projects in the textbook [Felke-Morris 2012] (e.g., creating web pages with fully fleshed style and content). Two projects, a personal web portfolio and site redesign, consisted of multiple components and spanned several weeks. Lastly, midterm and final exams were administered.

Relevant to this study, Assignments 1 and 2 required students to test their code and ensure it passed HTML and CSS validation. The World Wide Web Consortium (W3C), the governing standards organization of the web, outlines the benefits of validation[3], including that it:

- Teaches good practices for beginners and students by helping them spot mistakes and introducing broader quality concepts such as accessibility.
- Guards against errors that may not be handled consistently or gracefully across current and future platforms.
- Signals quality and whether the code is clean and well formed, or quickly hacked together.

Failure to pass validation resulted in a maximum 10 percent penalty for the assignments.

In previous semesters, students completed all activities using Aptana Studio, a full-featured integrated development environment based on Eclipse. For this study, activities from the first five weeks of the course were selected and adapted to use openHTML by the teacher. These activities were chosen because they mostly involved the creation of individual web pages rather than

[3] http://validator.w3.org/docs/why.html

multi-page sites and did not require the use of JavaScript. They were modified to work around some of the limitations of openHTML. The remaining assessments were completed with Aptana Studio as in previous terms. Table 6-2 provides descriptions of the activities that were used in this study.

Table 6-2: A description of the activities used in this study.

Assessment	Description
Lab 0	Create a web page with information about what you did during the last break, using basic HTML tags such as title, header, paragraph, and blockquote.
Lab 1	Create a to-do list using an ordered or unordered list as well as other basic HTML tags.
Lab 2	Copy the web page created in Lab 1 and add a section with your favorite links.
Lab 3	Copy a previously created web page and add an image. Also use CSS to position the image and add a background image to the web page.
Assignment 1	Create a resort homepage with a site header, navigation menu, content area, and footer using HTML.
Assignment 2	Copy the resort homepage from Assignment 1 to create an index and a subpage. Use a definition list in the subpage and use CSS to style the text and background colors. Link the two pages together.

6.1.2. Iterating on openHTML

As previously mentioned, I have taken a design-based research approach [Barab and Squire 2004] to designing openHTML by iteratively developing new features, deploying them in classes, workshops and other settings, and evaluating their impact. The two semesters in this study coincide with two rounds of this iterative design process. In the first semester, the design of openHTML was largely unchanged from the study presented in the previous chapter, allowing me to evaluate its design in a formal course setting. In the second semester, several features were added to provide greater utility in a

formal learning context. These features were designed to give additional support multiple stakeholders, including teachers, students, and researchers.

One of the features necessitated by the shift from a laboratory to an instructional setting was a way for teachers to access their student web pages for evaluation. While previous iterations of openHTML allowed students to share a direct link to a web page with the instructor, I was motivated to design a more efficient way for teachers to access these assignments based on informal feedback from the teacher. For Spring 2013, I created a course view that presents the teacher with a list of accounts. Upon selecting an account, the teacher is given direct access to that account's webpages. The teacher is also provided with a download option, which allows them to archive all revisions of a webpage for record keeping.

I also instrumented openHTML with a replayer that records changes to the HTML and CSS panes at the keystroke level, logs user actions such as saving and validating web pages, and plays back coding sessions (Figure 6-1). This feature is preceded by a number of tools that gather snapshots of programming activity and visualize them, as reviewed in [Heinonen et al. 2014]. The openHTML replayer provided a means to access student activity, given that direct observation was not possible: I was geographically remote from the class, and much of the activity occurred outside of classroom anyway.

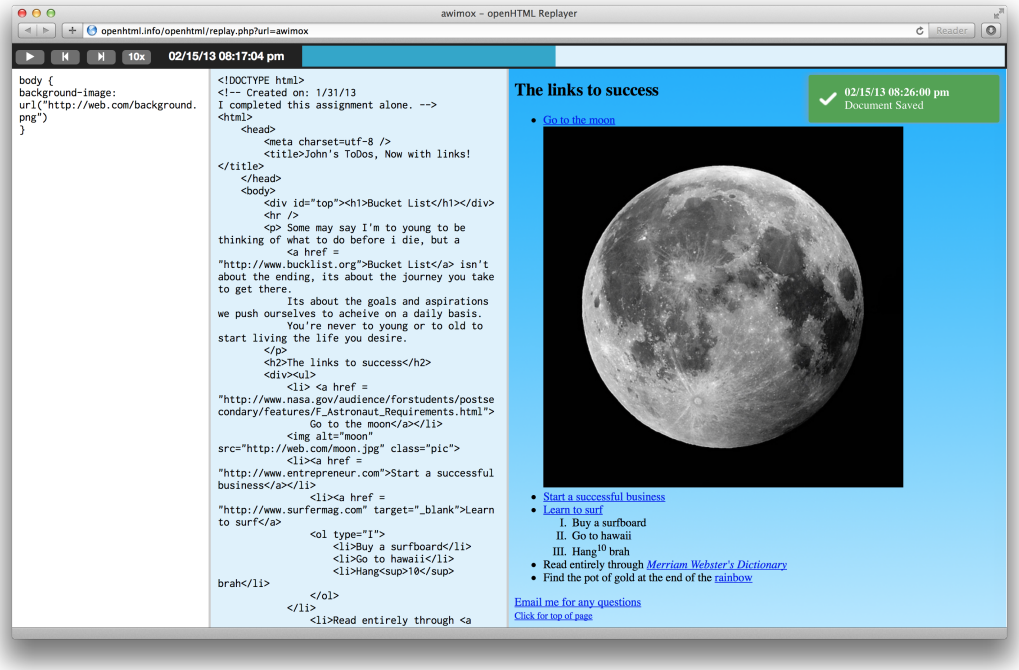

Figure 6-1: The openHTML replayer playing back a previously logged coding session.

In week 3 of the second term, HTML and CSS validators were integrated with openHTML for use with Assignments 1 and 2 (Figure 6-2.) By selecting these options in openHTML, students were able to validate their current webpage's HTML or CSS code, opening a new browser tab that listed the syntax errors that were detected in the code. The validators make use of the markup validation service APIs provided by the W3C.

While students in the previous term were also required to have their assignments pass validation, this involved visiting an external site, copying the code from openHTML, and pasting it into the external validator. The teacher reported that this was a cumbersome process for students. It also prevented

me from tracking their validation attempts, which I recognized as valuable data for interpreting coding activity that demarcated writing and testing code. Integrating a validation feature addressed both of these issues.

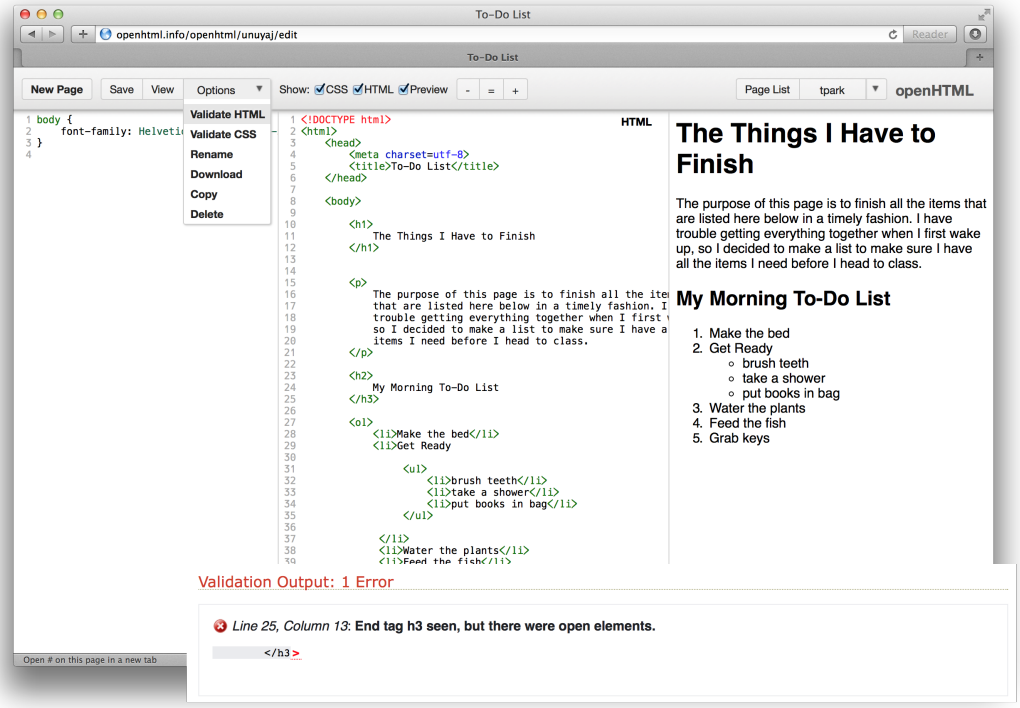

Figure 6-2: The openHTML validator feature, with an example error message.

6.1.3. Study Design

In this study, I performed a log analysis of the HTML and CSS coding activity of all students in the course. In most web development courses, only the students' end products—that is, the submitted version—is available to be assessed by instructors; similarly, much of the action that occurs when students learn web development is "researcher distant" — not amenable to

direct observation [Fincher et al. 2011]. However, I instrumented openHTML to log each saved revision of each project, providing snapshots of the assignments at varying levels of completion. Because openHTML was used for both in-class labs and homework assignments, this gave me a view into development process of students beyond the classroom. For the Spring 2013 study, even finer-grained logging was implemented, granting a keystroke-level view of students' coding activity so that I could inspect how HTML and CSS validation was used to detect and resolve errors. The analysis was informed by an earlier study of compilation behavior in introductory programming courses [Jadud 2006], which described the editing and compiling behavior of students learning Java and catalogued the most common compilation errors.

In the first part of the log analysis, which was conducted for both terms of the course, I examined the unresolved syntax errors present in the final version of the students' code. They represent errors that students did not detect, or even after validation, were unable to resolve, and likely indicate substantial difficulties given the stakes and the available resources. First, passing validation was an explicit requirement of these activities, and any errors remaining in their submissions negatively impacted their grades. Second, students were given several days to complete each activity, including the labs, and had access to the web and other resources to assist them.

To identify and analyze syntax errors, I passed student code through the HTML and CSS validators and cataloged the error messages that were

generated. Mentions of specific elements, attributes, or values were then replaced with placeholders, allowing similar error messages to be combined as of the same error type. For example, two error messages reported by the HTML validator were "Element h1 not allowed as child of element span in this context" and "Element dt not allowed as child of element body in this context". Both were grouped into the "Element Y not allowed as child of element X in this context" error type.

Following this process, I examined the error types along two dimensions:

- **Frequency:** Operationalized as the total count of each error type, this measures the most common unresolved errors in the course.
- **Prevalence:** Operationalized as the proportion of students making a type of error at least once, this measures the unresolved errors that affected the most students in the course.

I also analyzed the error messages by tallying mentions of language constructs in the error messages in order to uncover the elements, properties, etc. that were most problematic for students. This provides insight into the circumstances in which the errors were made.

In the second part of the log analysis, I examined snapshots of the students' code each time they validated their code. The assumption is that during validation attempts, students became aware of bugs present in their code and were making an effort to resolve them. This analysis reveals all of the errors that students encountered rather than only the ones that were

unresolved. It also identifies the errors that recurred in multiple validations and were particularly intractable, whether students were having difficulty resolving the same errors or were repeating similar ones. Along with frequency and prevalence, I analyzed the errors along two additional dimensions:

- **Recurrence:** Operationalized as the number of consecutive validations for which an error persisted, this measures how deeply errors affected students.
- **Resolution:** Operationalized by comparing the errors found during validation with the ones that were still present in the final submissions, this measures how successful students were in correcting errors.

The second part of the analysis was limited to Assignments 1 and 2 in the Spring 2013 term when the validator feature was added to openHTML.

6.1.4. Participants

The log analysis included the work of 23 students (9 in Fall 2013 and 14 in Spring 2013). 12 of these students (4 female, 8 male) agreed to interviews about their experiences with web development and programming prior to the course, which has been shown to predict the success of non-majors in learning to program [Wiedenbeck 2005]. Interviews were conducted in the first week of the course for the Spring 2013 term and near the end of the course for the Fall 2012 term due to scheduling constraints.

The interview participants averaged 21 years of age and ranged from the first to fourth year of their university program. Students were pursuing a

variety of majors, including web design and development, computer science, audio engineering, communications, and business. The course was a requirement for some and elective for others. Two of the participants, P9 and P11, withdrew from the course partway through. Data they submitted before they dropped the course were included in the analysis. Demographic data is provided in Table 6-3.

Table 6-3: Demographic data for the interview participants.

Code	Age	Gender	Major	Term	HTML	CSS	JS
P1	20	Female	Graphic Design	Fall 2012	•	•	
P2	20	Female	Web Design & Development	Fall 2012	•	•	
P3	19	Female	Web Design & Development	Fall 2012	••	••	
P4	19	Male	Computer Science	Spring 2013	•	•	
P5	19	Male	Audio Engineering	Spring 2013	••	•	
P6	21	Male	Audio Engineering	Spring 2013	•	•	•
P7	21	Female	Biology	Spring 2013	••	•	••
P8	24	Male	Computer Science	Spring 2013	•	•	•
P9	19	Male	Audio Engineering	Spring 2013	••	•	••
P10	21	Male	Communications	Spring 2013	••	••	•
P11	28	Male	Communications	Spring 2013	•	•	•
P12	19	Male	Business	Spring 2013	••	•	•

I was surprised to find that all of the participants interviewed had experience with HTML before the course. Students reported their level of prior experience with HTML, CSS, and JavaScript as either none (0), beginner (1), intermediate (2), or expert (3). All participants indicated that they had at least beginner experience in HTML, with an average rating of 1.50 (SD = 0.52). CSS and JavaScript were less familiar, with a mean of 1.17 (SD = 0.39) and 0.75 (SD = 0.75) respectively.

These earlier experiences tended to be limited, and students rarely recalled more than a few basic HTML elements from them. Nevertheless, they

expressed that these experiences were beneficial, allowing them to relate new information to knowledge.

> "...I was dealing with code. I didn't realize that's what I was doing then, but now, like when we learned a couple of HTML things, I was like oh, I knew that... It came in handy. I guess just like giving me confidence with things, like we would learn something and it wasn't totally foreign. I'd just kind of know what he [the teacher] was talking about." (P1)

Only two students reported taking another web development course before this course. Instead, students were primarily exposed to HTML, CSS, and JavaScript through informal activities on popular web services.

> "I think definitely the first time I ever used HTML was just for like having MySpace and I wanted a little bit more creative control, so I just kind of like learned. I knew a basic set of elements." (P6)

Several students reported learning basic HTML tags and CSS styles in order to customize profiles on social networking sites like MySpace, modify templates on blogging services Tumblr and WordPress, and create attention-grabbing posts on the classified advertising site Craigslist.

In their prior experiences, students generally took a more opportunistic approach to web development [Brandt et al. 2009], engaging in just-in-time learning to tweak existing code and personalize their content. They relied on web searches to find relevant information and snippets of code, which they often reused through trial-and- error without a full understanding, for example:

> "I looked up things on different web browsers and it was kind of very much like copy and paste code work. I didn't really understand what I was doing, but I understood that I could use those things and just change the values to whatever variables I needed." (P6)

Students indicated that they were often able to accomplish their immediate goals, but due to this reactive approach to learning, did not learn more fundamental concepts that would have enabled them to connect their experiences and develop a deeper understanding of the web and coding. Instead, they were on the path to developing "pockets of expertise" similar to many informal and even professional web developers [Rosson et al. 2004; Dorn and Guzdial 2010b].

6.2. Findings

6.2.1. Unresolved Errors

In this section, I report on the HTML and CSS syntax errors students were unable to resolve in their assessments. I start with an overview of the errors, and then discuss their relation to two concepts—nesting and parent-child rules, and how that changed as students progressed in the course.

6.2.1.1. Overview

A total of 382 unresolved syntax errors were found in the lab and assignment submissions. The average number of unresolved errors each student had was 16.6 (SD=18.4), ranging from 4 students who had no unresolved errors in their submissions to one student who had the maximum of 63. Figure 6-3 shows the distribution of unresolved errors among students.

Most of the errors related to HTML (97.4%). This is likely a product of the topics covered in the early part of the course when the log data was collected, rather than a generalizable proportion of HTML and CSS errors in an entire introductory course.

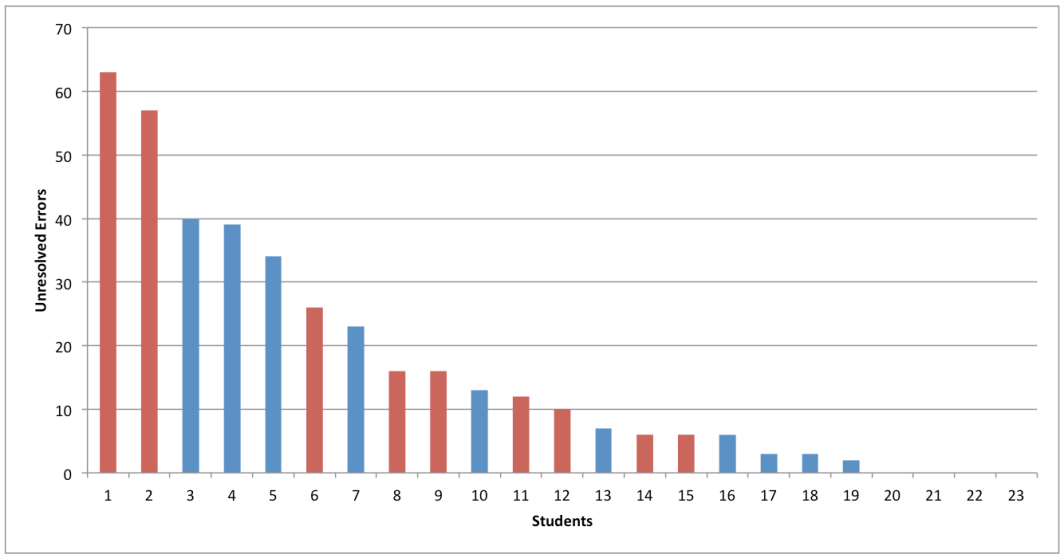

Figure 6-3: The number of unresolved errors per student. Students from Fall 2012 are in red and students from Spring 2013 in blue. All four students without any unresolved errors were from Spring 2013.

The syntax errors were distilled into 38 error types. Error types are shown in Table 6-4, along with their frequency (overall count) and prevalence (proportion of students who made this type of error at least once).

Table 6-4: Error types comprising unresolved errors by frequency and prevalence.

Error Categories	Frequency	Prevalence
Element Y not allowed as child of element X in this context.	77	15 (65%)
Unclosed element X.	47	10 (43%)
End tag X.	48	5 (22%)
Named character reference was not terminated by a semicolon. (Or & should have been escaped as &.)	32	6 (26%)
No X element in scope but a X end tag seen.	25	5 (22%)
End tag X seen, but there were open elements.	24	6 (26%)
Consecutive hyphens did not terminate a comment. -- is not permitted inside a comment, but e.g. - - is.	24	1 (4%)
End tag for X seen, but there were unclosed elements.	19	9 (39%)
Stray end tag X.	11	8 (35%)
Attribute Y not allowed on element X at this point.	9	3 (13%)
Bad value Z for attribute Y on element X.	8	7 (30%)
A X start tag seen but an element of the same type was already open.	8	6 (26%)
Value Error.	5	3 (13%)
End tag X violates nesting rules.	5	3 (13%)
Saw < when expecting an attribute name. Probable cause: Missing	4	4 (17%)

> immediately before.		
Element X is missing a required instance of child element Y.	4	3 (13%)
Quote " in attribute name. Probable cause: Matching quote missing somewhere earlier.	3	2 (9%)
Parse Error.	3	2 (9%)
A slash was not immediately followed by >.	3	3 (13%)
The Y attribute on the X element is obsolete.	2	1 (4%)
Stray start tag X.	2	1 (4%)
Element X is missing one or more of the following attributes: content, itemprop, property.	2	1 (4%)
Document type does not allow element X here.	2	1 (4%)
Y is not a X value.	1	1 (4%)
There is no attribute X.	1	1 (4%)
The X element is obsolete.	1	1 (4%)
Text not allowed in element X in this context.	1	1 (4%)
Self-closing syntax (/>) used on a non-void HTML element. Ignoring the slash and treating as a start tag.	1	1 (4%)
Required attribute X not specified.	1	1 (4%)
Property X doesn't exist.	1	1 (4%)
No space between attributes.	1	1 (4%)
No document type declaration.	1	1 (4%)
End tag for element X which is not open.	1	1 (4%)
Element X must not be empty.	1	1 (4%)
Duplicate attribute X.	1	1 (4%)
Bad character - after <. Probable cause: Unescaped <. Try escaping it as <.	1	1 (4%)
A X element must have a Y attribute, except under certain conditions.	1	1 (4%)
--! found in comment.	1	1 (4%)

The top ten error types accounted for 81% of the instances, with a long tail of errors made by only a small proportion of students. In the following sections, I organize these error types around two concepts, nesting and parent-child rules.

6.2.1.2. Nesting

Nesting is the organization of elements into multiple levels hierarchy and is a central aspect of HTML. Not too surprisingly, eight of the most common errors related directly to managing HTML start and end tags at multiple levels of nesting, comprising 35.1% of the total errors found in the students' final submissions. These included unclosed elements (i.e., missing end tags):

- Stray start tag X.
- End tag for X seen, but there were unclosed elements.
- End tag X seen, but there were open elements.
- Unclosed element X.

Extraneous end tags:

- Stray end tag X.
- End tag for element X which is not open.
- No X element in scope but a X end tag seen.

And errors caused by overlapped nesting (i.e., closing the outer element before the inner element is closed):

- End tag X violates nesting rules.

"End tag X" errors were not counted because I found on closer inspection that they resulted from void elements such as line breaks (br) with malformed syntax rather mistakes related to nesting.

Table 3-1 gives the number and proportion of errors related to nesting. The proportion of errors is a useful point of comparison given differences in the scope of each assignment, and shows that nesting errors remained relatively consistent from one assignment to next, with a slight downward turn.

The HTML constructs reported in the original error messages (Table 6-6) shed light on when and why beginners are likely to make nesting errors.

Table 6-5: The number of nesting errors by assignment. The proportion of overall errors is given in parentheses.

Assignment	Nesting Errors
Lab 0	11 (33.3%)
Lab 1	8 (24.2%)
Lab 2	16 (30.8%)
Lab 3	14 (21.5%)
Assignment 1	8 (12.5%)
Assignment 2	30 (22.2%)

Table 6-6: A count of the HTML elements mentioned in error messages related to nesting.

Construct	Count
div	18
body	17
strong	15
p	14
li	10
small	9
head	9
sub	7
sup	4
em	4
dt	4
ol	3
i	3
cite	3
ul	2
span	2
nav	2
html	2
title	1
hr	1
h1	1
a	1

Nesting error messages most occurred most often when dealing with div elements. There are several reasons this might be the case. First, div elements are simply a frequently used element. Second, they are commonly used at multiple levels of nesting as a generic element to organize content, nested within other divs to define page layouts. Using identical elements multiple

times makes tracking different levels of nesting difficult and increases the likelihood of making errors with them, although this can be mitigated through coding practices like indentation and comments.

Inline elements such as strong, i, small, and em, were also frequently involved in nesting errors. Beginners who are not yet comfortable with CSS tend to rely heavily on these HTML elements to bold, italicize, or change the size of text. In this usage, beginners wrap text with several of these tags at once to apply multiple styles, making them prone to nesting errors. An aggravating factor is that when multiple inline elements are used to wrap a single word or sentence, their tags are often written on a single line of code where no indentation is available to help track nesting.

Finally, upon closer inspection of the responsible code, errors involving head ("Stray end tag head"), body ("End tag for body seen, but there were unclosed elements"), and dt ("No dt element in scope but a dt end tag seen.") were typically not the result of improper nesting. In HTML, when rules requiring elements to have specific parent or child elements were broken (e.g., placing content elements outside of the head or body), they are implicitly closed or new ones created by the validator, resulting in unmatched start or end tags. Thus, web pages with invalid syntax can still often be rendered, making HTML a more forgiving language, but often leading to unexpected behaviors and baffling error messages that hinder debugging. In the next

section, I will discuss additional error messages that directly related to these parent-child rules.

6.2.1.3. Parent-Child Rules

Nesting HTML elements naturally gives rise to a hierarchical structure in the code, where elements are contained by parent elements and themselves contain child elements. These elements have rules that constrain how elements can be nested within others. One example of this is the HTML element, which must be the root-level element and can only contain one head followed by one body element.

Most other elements have more freedom in how they can be nested within one another but are nonetheless governed by parent-child rules. But these rules were frequently unfamiliar or not well understood by the students. One plausible account is that beginners often make the simplifying assumption that aside from a small set of special cases like html, head, and body, elements can be freely nested within one another. In many cases this simplifying assumption is workable, producing in a web page that renders as desired, but resulting in syntactically invalid code. Three of the error types related to rules that dictate how elements can be nested, accounting for 21.5% of all unresolved syntax errors.

- Element Y not allowed as child of element X in this context.
- Text not allowed in element X in this context.
- Element X is missing a required instance of child element Y.

The proportion of errors related to parent-child rules had more variation than did nesting errors from one assignment to the next. As seen in Table 6-7, they were most common in Lab 1, Lab 2, and Assignment 2, which involved the creation of list elements.

Table 6-7: The number of parent-child errors by assignment. The proportion of overall errors is given in parentheses.

Assignment	Parent-Child Errors
Lab 0	3 (9.1%)
Lab 1	10 (30.3%)
Lab 2	12 (23.1%)
Lab 3	13 (20.0%)
Assignment 1	3 (4.7%)
Assignment 2	41 (30.4%)

Once again, I examined the constructs that were mentioned in the error messages to get a better sense of when students encountered these errors. In Table 6-8, parent-child combinations that occurred more than once are shown. The parent elements (i.e., X) are given in the top row and the child elements (i.e., Y) are given in the leftmost column.

Table 6-8: The most common HTML elements mentioned in error messages related to parent-child rules. Parent elements are listed horizontally and child elements vertically.

	body	head	ol	ul	dl	div	strong	small
dt	10					6		
dd	10					6		
title	2	6						
ul			7	4				
blockquote								
hr							3	2
br				4	6			
a				3				

Most of these errors related to description list (dl) elements and their required child elements (dt and dd). The prevalence of description lists is expected as they were a requirement for Assignment 2 and one of the first elements introduced in the course that must be used in coordination with child elements. Ordered (ol) and unordered lists (ul) were similarly problematic for students, particularly when nesting lists and sublists. In these cases, a common error was placing the opening sublist tag outside of its parent's list items.

Most of the remaining parent-child errors occurred when students nested block elements within inline elements. In HTML, there are two basic content models: block elements (e.g., div, table, p) expand to take up the available width, while inline elements (e.g., span, strong, em) contract around a text string. It is valid for block elements to have either block or inline elements as children, but with few exceptions, inline elements can only contain text or other inline elements. Although the instructor taught students about this distinction, it is an open question to what degree it was a matter of student understanding versus recall.

6.2.1.4. Other Errors

Several of the other error types can also be organized around concepts. Parsing errors like "Saw < when expecting an attribute name" and "A slash was not immediately followed by >" indicate problems that students had with the syntax within markup tags instead of between them. The syntax of void elements such as line breaks and horizontal rules, which are comprised of a

single tag instead of a pair, also presented difficulties for the students, resulting in the "Self-closing syntax (/>) used on a non-void HTML element" error message. The occasional error of this type may be a simple typo; however, the persistent recurrence of this type of error may be a red flag, indicating deeper problems grasping the syntax of individual markup tags rather than the coordination of multiple elements.

A second group of errors related to representations of various data, including HTML character references (e.g., ©), colors in hexadecimal notation (e.g., #53A5C5), and URLs (e.g., http://openhtml.org/). Each of these introduces new schemes for properly formatting values, and the opportunity to engage with additional facets of computation.

6.2.2. Resolving Errors

The previous section gives insight into the errors that remained in the final versions of students' assessments. In this section, I use the validator feature as a lens for analyzing coding activity *during* the construction of web pages. Specifically, I analyze the errors present in the students' code for each validation attempt, with the goal of identifying the recurrence and resolution of the errors. This analysis is based on a closer inspection of Assignments 1 and 2 for Spring 2013, enabled by the validator feature and fine-grained logging that were added to openHTML.

6.2.2.1. Validator Usage

Students averaged 12.6 validations (SD=15.0). About half of validations (50.6%) resulted in one or more errors. The extent to which the validators were used varied from student to student, ranging from three students who did not use the validator at all to one student who used it 56 times across their assignments.

Based on observations of coding activity using the openHTML replayer, there seemed to be little correlation between an ability to write syntactically correct code and validator usage. Among students who showed the ability to write error-free code, some rarely validated their code until it was nearly completed while others used it methodically from early on. Similarly, among students who had more substantial difficulties, some relied on the validators heavily to debug their code while others used them rarely or not at all. This is characteristic of the behavioral differences in stoppers, movers, and tinkerers that has been observed among novice programmers [Perkins et al. 1986; Jadud 2006].

6.2.2.2. Recurrence of Errors

Validations generated 582 total error messages, which were narrowed down to 23 error types. Identical errors, determined by error message and location, in consecutive validations were combined into a single episode, which resulted in 268 episodes. These are summarized in Table 6-9.

In addition to frequency and prevalence, recurrence was calculated as the number of consecutive validation attempts in which an error was present. Recurrence indicates how persistent an error is and suggests the degree with which students had trouble resolving it through repeated validation attempts. Errors requiring multiple validation attempts are suggestive of deeper conceptual difficulties as opposed typographical mistakes and other slips.

Table 6-9: Types of errors found during validation. Frequency is the number of instances of an error, prevalence is the number and percentage of students that made an error at least once, recurrence is the median number of validations that an error lasted, and resolution is the number and percentage of instances that were eventually resolved.

Error Categories	Frequency	Prevalence	Recurrence Median	Recurrence Max	Resolution
Element Y is not allowed as child of element X in this context.	73	10 (77%)	1	7	65 (89%)
Consecutive hyphens did not terminate a comment. -- is not permitted inside a comment, but e.g. -- is.	39	3 (23%)	4	6	35 (90%)
Unclosed element X.	23	7 (54%)	1	3	22 (96%)
End tag X.	21	4 (31%)	3	5	21 (100%)
Attribute Y not allowed on element X at this point.	20	1 (8%)	1	1	20 (100%)
Named character reference was not terminated by a semicolon. (Or & should have been escaped as &.)	17	4 (31%)	1	3	17 (100%)
No X element in scope but a X end tag seen.	16	5 (38%)	2	2	16 (100%)
End tag for X seen, but there were unclosed elements.	13	7 (54%)	1	3	12 (92%)
End tag X seen, but there were open elements.	10	4 (31%)	1	1	10 (100%)
Y is not a X value.	4	2 (15%)	1	2	4 (100%)

Stray end tag X.	4	2 (15%)	1	1	4 (100%)
Property X doesn't exist.	4	2 (15%)	1	2	4 (100%)
Parse Error	4	1 (8%)	1	2	4 (100%)
End tag X violates nesting rules.	4	1 (8%)	1	1	4 (100%)
Bad value Z for attribute Y on element X.	4	2 (15%)	1	1	4 (100%)
Garbage after </.	3	1 (8%)	3	5	3 (100%)
Text not allowed in element X in this context.	2	1 (8%)	2	2	2 (100%)
Element X is missing a required child element.	2	2 (15%)	1	2	2 (100%)
Saw = when expecting an attribute name. Probable cause: Attribute name missing.	1	1 (8%)	1	1	1 (100%)
No space between attributes.	1	1 (8%)	1	1	1 (100%)
Character reference was not terminated by a semicolon.	1	1 (8%)	2	2	1 (100%)
A slash was not immediately followed by >.	1	1 (8%)	1	1	1 (100%)
< in attribute name. Probable cause: > missing immediately before.	1	1 (8%)	1	1	1 (100%)

40.7% of errors lasted more than one validation, up to a maximum of 7. The mean recurrence rate was 2.0 validation attempts (SD=1.5) and the median was 1. Since the recurrence of errors was highly skewed, with most lasting only one validation attempt, the median and maximum are provided in the table above. One explanation for this skew is that fixing one validation error commonly resolved several additional errors. Given that most errors lasted only one validation attempt, comparing error types by their average recurrence

rates is not highly instructive. Instead, recurrence seems most useful as a measure for detecting when an individual student is having acute difficulties, expressed as high maximum recurrence rates in the table above.

I note that recurrence rates alone do not fully capture the extent of difficulties students had resolving a particular problem. Inspecting their coding activity through the openHTML replayer revealed that students often engaged in tinkering, toggling the faulty code or eliminating it completely but got no closer to a solution. Especially problematic errors also sometimes led to a cascade of new errors. In these situations, recurrence rates would be low and would not accurately reflect the scope of the problem.

6.2.2.3. Resolution

Despite HTML and CSS syntax errors taking multiple attempts to resolve, students were eventually successful in resolving them in most cases. Among the 268 errors detected in Assignments 1 and 2, only 14 were unresolved in the final submissions — a 94.8% success rate. This was consistent from one error type to the next, all ranging from about 90% to 100%.

When comparing this analysis with the results in Section 6.2.1, I found that in addition to the 14 unresolved errors reported here, 67 were introduced after the final validation attempt or in code that was never validated at all. In other words, 82.7% of the unresolved errors were never brought to the attention of students through validation. It is likely that an equally high number of the unresolved errors in the other term were never detected, given that the

validators were not integrated with openHTML at that point, requiring additional effort on the part of students. By practicing validation more systematically, students might be able to resolve up to 95% of unresolved errors in the other assignments as well.

6.3. Discussion

6.3.1. Mastering Syntax through Practice

In this study, I focused on syntax errors in HTML and CSS, two languages that are fundamental to web development but often overlooked in computing education research. My results highlight that despite the seeming simplicity of these languages, their syntaxes can present many challenges for beginners. On average, students had 16.6 unresolved errors across the six assessments included in the analysis; only 4 students submitted error-free code. These errors not only present obstacles to authoring syntactically valid code, but also compound difficulties with semantics and design.

Accounting for nearly a quarter of the unresolved errors were issues related to parent-child rules. Parent-child relationships follow an extensive system of rules that govern when it is valid for certain types of HTML elements to be nested in one another. In my previous study, I found that errors related to these occurred primarily at the rule-based level of behavior. With the introduction of new elements, parent-child rules and the interactions between them continue to grow. In the study, this was reflected in the types of

errors students made as they progressed, which ballooned with the introduction of lists and sub-lists. I expect that as students learn new, compound elements such as tables and forms, and work with larger, more complex web pages, they will continue to make errors that violate the myriad parent-child rules.

On the other hand, the syntax for nesting tags is relatively simple and consistent from one element to the next, yet it accounted for over one-third of unresolved errors. Although students appeared to grasp the syntax of nesting tags quickly (all of the students demonstrated proper nesting from the first assessment), errors related to nested tags occurred with regularity during all five weeks of the study. These errors often manifested when students were confronted with new, compound elements like lists, and deeper levels of nested tags. This suggests that these nesting errors were attentional in nature — as they grappled with unfamiliar or complex code and their cognitive load was taxed [Chandler and Sweller 1996], an end tag was forgotten or misplaced. Indeed, in my previous study, I found that most errors related to nesting tags occurred at the skill-based level of behavior and were attentional in nature.

What this underscores is that beyond declarative knowledge, practice plays an important role when learning HTML and CSS. The syntax of nesting markup tags may be learned on day one, but the development of skills related to reading and writing nested code — which includes visually parsing delimiters (start and end tags in the case of HTML), and mentally translating

them into a hierarchical structure — is ongoing. Through practice, the deliberate processes of reading and writing nested code can eventually become highly routinized skills [Rasmussen 1983], helping minimize these errors while freeing cognitive load for higher-level concerns. This parallels research on reading text, where reading skills at the letter and word levels have been found to influence higher-level reading comprehension and achievement [Biemiller 1977].

6.3.2. Learning through Validation

The findings show that validation is important, surfacing syntax errors in half of the students' validations attempts. Furthermore, validation is effective, evidenced by the eventual resolution of nearly 95% of the errors detected with validation. In contrast, most of the unresolved errors (83%) were present in code that was never tested. Despite its effectiveness, many students did not to validate their code. This is all the more surprising given that validation was an explicit requirement of the homework assignments, and that in later weeks, openHTML's integrated validator provided added convenience.

One of the challenges of web development is that validation is optional, unlike programming languages like Java that require a compilation step. Complicating matters further, HTML is a highly forgiving language by design, and browser engines attempt to render a web page even in the presence of syntax errors, implicitly modifying the source code if necessary to do so. This results in cases where a web page displays exactly as intended while numerous

syntax errors remain latent in the source. Beyond the quality of the code itself, this lack of feedback can lead to the development of poor habits and faulty mental models that do not equip students with the ability to predict the relationship between input and output in new contexts [duBoulay 1986].

Beyond teaching validation practices specifically and testing more generally, there is an opportunity to encourage validation through the design of web editors. For instance, displaying the validation status of a web page upon saving it would help users to maintain awareness of latent errors in code and motivate users to correct them. Many existing web editors go even further, providing instant feedback of errors detected in the code.

Finally, there are opportunities to improve validator feedback by addressing understanding and suggesting solutions. Although students successfully corrected most of the errors detected during validation, there were cases that required numerous validation attempts. The feedback provided by the validators likely contributed to the difficulties students had in resolving these errors. Many errors generated cryptic feedback; programming language compiler feedback is likewise known to cause novices trouble [Nienaltowski et al. 2007; Marceau et al. 2011; Lee and Ko 2011; Denny et al. 2014]. . This was especially the case for the CSS validator, which returned terse "parse error" or "value error" messages.

One message could be associated with multiple, disparate errors. A common error, "element Y not allowed as child of parent element X in this

context", was made at least once by 77% of students, with 65% leaving one or more of them unresolved. This error occurred in two distinct circumstances: when parent-child rules were violated by valid elements and when invalid elements (e.g., <text> and <stong>) were used at all. Conversely, one error could be associated with multiple error messages. An extraneous end tag could alternately trigger a "stray end tag" or "no element in scope but end tag seen" message depending on the context in which it occurred. Students had little help in understanding the reasons for these distinctions.

When a student is validating their code and reviewing their errors, this is a critical learning opportunity. Rather than mere technical correctness, we see these events as opportunities to provide actionable feedback that helps students learn to author correct code and improve their understanding [Hartmann et al. 2010].

6.3.3. Limitations

Several limitations temper these findings. First, this study focuses on a relatively small sample of students in a single course. Therefore, while the results shed light on the types of syntax errors novices make, they are not likely to generalize to web development students in all contexts. For example, participants in this study were mostly non-CS majors with minimal prior programming experience. A course comprised of CS majors with significant programming experience may commit fewer syntax errors related to nesting, which draws on general skills associated with program composition and

comprehension [Corritore and Wiedenbeck 1991], compared to parent-child rules that are more specific to the domain of web development.

Moreover, this study focused on early assessments that introduced features of HTML and a small amount of CSS. I did not track student activity involving JavaScript and more complex HTML and CSS, which are likely to introduce different kinds of errors and resolution strategies. This was due to the limited viability of openHTML in later weeks of the course. openHTML achieves much of its simplicity by supporting only single web pages. The drawback of this approach is that reusing a CSS stylesheet or other resources between multiple HTML documents becomes cumbersome, requiring users to duplicate their efforts for each webpage. The instructor of the course was able to use openHTML for the two-page site in Assignment 2 through the careful design of the assignment and guidance for the students, but using openHTML beyond this point was simply not feasible. The use of openHTML was also limited by its abstraction of the file management, which allows beginners to focus on the code but prevents them from learning to organize files and use relative links to reference web pages, images, and other resources. This is an important aspect of web development and of computational literacy more generally [Miller et al. 2010].

The log analysis provided a fine-grained view of coding behavior, but I was limited by a lack of contextual clues compared to the previous lab study. Although the assignments provide some guidance on the students' overall

objectives, this lack of context limited our ability to infer their intent with respect to more granular actions. Because of these limitations, I focused the analysis on syntax errors, setting aside difficulties they might have planning the design of a web page or the semantic errors they may have made while creating it. These also comprise a significant portion of their learning experience and have the potential to impact their attitudes and progress in learning web development as much as syntax errors.

Compared to the study described in chapter 5, the log analysis used in this study was limited in its ability to explore the causes of the syntax errors. However, rather than discounting log analysis altogether, the findings suggest how analyzing student activity patterns over time can provide clues to the cause of errors. This study made error messages the unit of analysis. As noted in the results section, multiple errors often tied together as extended episodes of debugging or as symptoms of a deeper conceptual problem. How these data can be effectively interpreted to understand higher-order difficulties or determine the cause of errors is open for future research.

One potential approach is to couple remote log analysis with more direct methods of inquiry. For instance, a post-test might ask students to interpret error messages that they encountered during the course. Students could also be asked to assess the severity of errors and the usefulness of the feedback that the validators provide, whether through follow-up interviews or during the course through a feature implemented in openHTML.

6.4. Summary

In this chapter, I have presented a study of students in an introductory web development course using openHTML to complete their initial assignments. Activity logs collected from openHTML were analyzed to investigate the nature of the syntax errors students made and how they were able to overcome them.

First, with respect to the most common syntax errors that students had, 35.1 percent of the unresolved errors, made up of eight error types, directly related to nesting. An additional 21.5 percent of unresolved errors, made up of three error types, related to parent-child rules. While students demonstrated a familiarity with nesting, they continued to make nesting errors with consistency in the later assignments, particularly when dealing with new elements or more complex structures. On the other hand, errors related to parent-child rules occurred when students encountered new elements or new interactions between elements.

Second, I investigated how well students were able to resolve the errors they made. When validating their code, students were quite successful in overcoming the syntax errors they encountered. In Assignments 1 and 2, the Spring 2013 students found 268 distinct syntax errors in their code during validation. They were able to resolve 94.8 percent of these, taking only 1 or 2 validation attempts to do so in the vast majority of cases. While in the aggregate, measures of error recurrence and resolution showed students were

successful in fixing errors, these measures were also useful in identifying cases where a small number of students did have substantial trouble overcoming errors related to issues like commenting HTML code.

Finally, I explored the effect that the HTML and CSS validators integrated with openHTML had on the students' ability to overcome errors. I found that the validators were instrumental in making students aware of and ultimately resolve errors. Only 5.2 percent of the errors that students detected through validation were unresolved. In fact, the vast majority of unresolved errors, 82.7 percent, occurred after the final validation attempt that a student made or in code that was never validated at all. While students made use of the validators, averaging 12.6 validation attempts, they did not use them consistently. Three students did not use them at all. Despite the value of validation in helping students become aware of syntax errors, most errors were latent, invisible in the ever-present feedback provided by the live preview pane. An indicator for validation status in openHTML is one way to help students maintain awareness of syntax errors in their code and motivate them to correct them without inundating them with error messages. For errors that students did detect but nevertheless had trouble resolving, there is an opportunity to improve validator feedback to not only provide a description of the error, but possible solutions and explanations that strive to improve student understanding.

Chapter 7
Conclusion

In this dissertation, I have investigated the experiences of beginners learning basic web development through the lens of computing education research. I have conducted several studies that examine the difficulties they face learning HTML and CSS, which have informed the design of openHTML. In this final chapter, I discuss the major contributions of my findings and outline avenues for future work.

7.1. Contributions

7.1.1. Learning Barriers in a Web Development Course

This dissertation characterizes the learning experiences of web development in terms of the barriers students encounter in an introductory web development course. In Chapter 3, I conducted a content analysis of the help forums used in a course and identified five broad types of barriers: administration, content, design, coding, and technology.

I determined that 34% of help-seeking instances related to coding. I also discovered that administrative and technological issues were also significant, making up 30% and 25% of help-seeking instances respectively, with technological issues related to configuring the development environment

especially acute in the initial weeks of the course and causing distress for many students.

Although my primary interests lay in the difficulties and opportunities students have when learning HTML and CSS, the non-coding barriers put them in perspective. These findings enrich the literature on the experiences students have in a web development course, which have largely been comprised of case studies, by providing a detailed analysis that is firmly grounded in contemporaneous data.

7.1.2. Common Errors in HTML and CSS

A second outcome of this dissertation is providing one of the first and most detailed investigations of the errors people make when using HTML and CSS. In the computing education literature, errors have frequently been used as a lens for understanding how people learn about programming, though the precise errors identified with HTML and CSS differ from ones previously identified with programming languages — absent are difficulties with variable assignment, loops, or recursion. All three of my main studies contribute to an understanding of these errors. In addition to a detailed description of common errors people make with HTML and CSS, these studies provide two main insights.

First, I have found repeated evidence that beginners can have substantial difficulties with HTML and CSS, despite their relative simplicity and, given the gulf between the number of people who learn these languages and the dearth

of research on the topic, their presumed ease of use. In Chapter 3, I determined that students consistently sought help for HTML for the duration of the course, indicating problems they were unable to resolve on their own. In Chapter 5, participants were tasked with completing basic HTML and CSS coding tasks. Despite possessing various levels of prior experience with web development, including two who self-identified as professional web developers, participants made numerous errors, including several instances where they were not able to complete a task. Finally, I turned back to a web development course in Chapter 6. By analyzing activity logs in openHTML, I identified 38 different syntax error types, and determined many cases where students repeated errors of the same type and had difficulty resolving them. Consistent with my first study, I found that not only did syntax errors with HTML continue weeks into the course, but that they increased in frequency with the introduction of new elements and the growing scope and complexity of the assessments.

Second, I have found support for the value of understanding intent when analyzing these errors. In some cases, errors were due to a lack of familiarity with the extensive syntactic and semantic rules governing how HTML and CSS are used (e.g., parent-child rules), and unanticipated interactions between these rules. In other cases, participants violated well-understood rules (e.g. nesting elements), exhibiting signs of a strain on their cognitive load while dealing with complex or less familiar constructs. In Chapter 5, I reported on a

think-aloud task study that used an intention-based analysis, identifying not only how errors manifest in the code (i.e., the symptoms of student difficulties), but tapping into the goals, plans, and mental models of the participants (i.e., the causes) that explain these errors. In Chapter 6, my ability to determine intent was limited by my use of remote logging methods, but I was able to make limited inferences by relying on my experiences in the previous study and analyzing activity over time (e.g., successfully nesting HTML elements until dealing with more deeply nested code).

Understanding intent is critical for interpreting and addressing programming errors. To illustrate this point, consider a recent study that found enhanced syntax error feedback to be ineffectual for Java students [Denny et al. 2014]. Backing this claim, they report no significant difference in the number of non-compiling submissions and attempts needed to resolve errors between students presented with standard and enhanced feedback. In Chapter 5, I reported that the majority of HTML and CSS errors were typos and other skill-based errors, for which enhanced feedback would be expected to have little effect. On the other hand, a smaller proportion of errors related to unfamiliarity or misunderstanding of specific rules, for which enhanced feedback would be expected to provide a great deal more benefit. Assuming a similar distribution for the Java students, the lack of a statistically significant difference is not surprising. However, taking only the rule-based errors under consideration, a much stronger effect is likely to be observed.

A secondary outcome of this emphasis on intention when studying errors is methodological in nature. In Chapter 5, I outlined methods for the intention-based analysis of coding errors, and heuristics for classifying errors at different levels of activity. Despite the methodological challenges that are raised, the additional effort required to probe the intent of learners in order to understand the cause of errors and design systems that effectively address them is justified.

7.1.3. The Design of a Web Editor for Learners

I have also reported on the design and implementation of openHTML, which strives to minimize non-coding barriers while exposing users to coding as an authentic practice of web developers and a vehicle for introducing computational concepts. The deployment of openHTML demonstrated the efficacy of such a tool in the initial weeks of an introductory web development course and revealed several tradeoffs of this minimal approach. For instance, the single page approach limited opportunities to learn about reuse of stylesheets and other resources and the lack of a file system led to similar issues for learning about source code organization and the use of relative paths.

openHTML also serves as a case study for taking a design-based research approach to supporting web developers. In contrast to most web editors, openHTML was designed with a focus on learners and was informed by multiple rounds of user research. Many aspects of its initial design were

informed by the barriers reported in Chapter 3, including simplified configuration through a web-based implementation, a minimal interface that focuses on HTML and CSS, and immediate feedback through the live preview. By observing usage in an after-school workshop (Chapter 4) and a lab study (Chapter 5), I also identified several usability issues that were addressed through minor tweaks. As described in Chapter 6, several major features were then added to support openHTML's use in formal learning context, including basic administrative features and built-in HTML and CSS validators. Finally, openHTML served as a research instrument, enabling the analysis of student coding behavior through webpage revisions and later, fine-grained activity logging.

This work illustrates a loose, but nonetheless constructive, form of DBR in which multiple empirical studies of novice web developers were used to inform the design of a system, and the system was in turn designed to support research efforts.

7.1.4. Computational Literacy in Basic Web Development

Finally, my studies offer initial evidence for basic web development as a rich context for becoming computationally literate, and characterize the skills and concepts with which people are likely to engage when learning HTML and CSS. Based on the analysis of online help-seeking behavior presented in Chapter 3, I argued that students engage with and have difficulties related to fundamental computational skills and concepts such as notation, hierarchies

and paths, and decomposition. Through the lab-based task study described in Chapter 5, I catalogued numerous skill-based errors such as mistyped constructs and unclosed tags that have parallels with errors commonly made by novice programmers. In the log analysis presented in Chapter 6, I identified two concepts fundamental to HTML, nesting and parent-child rules, and analyzed the syntax errors made by students in terms of them.

In comparison to programming languages like JavaScript, HTML and CSS are a great deal more constrained. Instead of defining one's own properties and methods, HTML and CSS largely expose only ready-made attributes and properties. While this reduced expressiveness can be seen as a disadvantage, it offers some benefits in terms of an introduction to writing code. First, this reduces complexity and cognitive burden by allowing learners to focus on the "what" instead of the "how" as is typical with declarative paradigms. Second, given the domain-specific nature of these languages, their applicability is more apparent. The increased contextualization may confer both motivational and cognitive benefits.

Interestingly, in terms of computational literacy skills and knowledge, where HTML and CSS have most overlap with conventional programming languages are at the very low and high levels. I found that many errors related to low-level skills like enclosing HTML values in quotes, terminating CSS declarations with semicolons, and navigating multiple levels of nested code, which share commonalities with errors that are observed in programming. At

the high level, practices and perspectives such as the precision of computing languages, separation of concerns, modularization, testing, and debugging are shared with programming languages. Where HTML and CSS diverge most are at the level that binds the low with the high, within the syntax and semantics of the particular language constructs.

Identifying computational skills and concepts that students engage with when learning HTML and CSS is a first step, setting the foundation for further research. Therefore, I start the next section on future directions with a discussion on how the learning of such skills and concepts might be measured.

7.2. Future Directions

In this section, I discuss several avenues for future work that build on the research presented in this dissertation.

7.2.1. Learning Effects in Web Development

This dissertation provides initial evidence for basic web development as a context for developing computational literacy, identifying several computational skills and concepts that beginners engage with through HTML and CSS. As yet unknown are to what extent students develop these skills and knowledge, and the effect new approaches to teaching and supporting students might have on them. To pursue this line of inquiry involves developing instruments that measure student learning of the computational skills and concepts that have been identified in this dissertation, and using

these instruments to conduct pre- and post-assessments that can compare the effect of various interventions.

For example, many of the difficulties students had related to reading and writing code at deep levels of nesting. My findings suggest that these errors occur at the skill-based level. That is, students are aware of the syntax for nesting HTML elements within other elements, but forget or misplace the element's end tag when their working memory is overloaded. On the other hand, developing the ability to read deeply nested code frees up working memory to attend to higher-level concerns with the code. Studies of reading have found a similar relationship between low-level skills like reading letters, words, and simple text with higher-order reading achievement [Biemiller 1977].

I am in the early stages of developing an instrument that measures the speed and precision with which students are able to navigate and format hierarchically structure code, loosely inspired by Parson problems [Parsons and Haden 2006]. This instrument, which probes the ability to translate linear text into an abstract, hierarchical model, could be applied to HTML as well as other forms of code such as JavaScript, JSON, and CSS. One potential application is to use it with web development students before and after a course, in order to measure the effect of learning web development on navigating both forms of code taught in the course and its transfer to new unfamiliar formats.

7.2.2. Informal Learning at a Large Scale

The studies presented in this dissertation have primarily examined students in university courses. A formal learning context was chosen as a starting point in order to efficiently study beginners with minimal programming experience learning a common set of topics. However, the literature [Rosson et al. 2004; Dorn and Guzdial 2010a], as well as reports by the participants in the lab-based study, demonstrate the diversity of backgrounds found in web development. Informal, self-directed learning must also be considered when studying how to support beginners of web development.

This informal learning typically occurs online and is sporadic, punctuated by bursts of intense activity, posing a significant challenge for researchers. One way to resolve this is to go where the informal learning happens. For instance, prior studies show that online documentation, tutorials, and question-and-answer forums are popular learning resources. Analyzing how users engage with these resources can give insight into the nature of learning web development. A second option is to take a more interventionist approach by promoting usage of instrumented web development tools and resources. The study described in Chapter 6 highlights the promise of remote logging as an efficient method of studying learning in web development at scale. By instrumenting openHTML, I was able to capture the coding behavior of all of the students whether in the classroom or at home, although my analysis only scratched the surface of making sense of this fine-grained coding data to

understand larger underlying phenomena. Future work would draw from the fields of educational data mining [Baker and Yacef 2009; Romero and Ventura 2010] and learning analytics [Siemens 2012], harnessing "big data" and machine learning techniques to explore questions about learning practices in web development.

A major theme underscoring my research is the criticality of understanding a learner's intention when interpreting their coding behaviors and providing them with guidance. Unfortunately, online activity logs as they have been used in openHTML offer little insight into the user's intentions. One way to address this is to revisit the think-aloud data described in Chapter 5 and analyze the relationships between errors manifested in the code and the underlying cognitive causes. Some errors, such as missing end tags, are likely to have a higher probability of occurring at the skill-based level, while others, such as using class names that begin with a numeral, are likely to occur at the rule-based level. A second, more direct approach is to present students with snippets of code containing various common errors for which their understanding can be gauged.

A final approach is to design openHTML to provide progressive error feedback. At the first level, errors are marked only by their presence and location. If the user is unable to correct the error with the aid of this information, they can activate a second, enhanced level that provides an explanation of the error and potential solutions. A user who successful fixes

an error based on the first level of feedback would indicate a skill-based error, while another user who intentionally accesses the second level of feedback indicates a rule- or knowledge-based error.

7.2.3. Improving Teaching and Learning Tools

openHTML and similar tools offer ongoing opportunities to design and evaluate novel features that support learning HTML and CSS. I will outline three paths forward.

First, as discussed in section 6.3.2, HTML and CSS validation feedback is often cryptic or misleading, rarely addressing learners' conceptions or offering solutions. Additionally, the feature only detected syntax errors, although learners would benefit from feedback on common semantic errors such as unused classes. Future work could be devoted to designing error feedback that is more understandable to beginners and has been developed through an understanding of their current mental models. Findings from the research described in the previous section on relating coding errors to intention could inform this work.

Second, the initial design of openHTML achieved addition by subtraction: I abstracted away many of the non-coding barriers in order to help users focus on the code. Future iterations might be devoted to providing learners with within-tools scaffolding, that is, features that actively model more expert knowledge and practices for beginners, and that can eventually be faded away when no longer needed. For instance, I have developed a tutorial feature that

deeply integrates with code in a web editor. Scaffolding might also take the form of a code snippet library modeling HTML and CSS patterns that users can browse, copy into their own code, and modify.

Finally, the logging feature of openHTML suggests features that track activity and support assessment for both teachers and students. For students, features might be designed to support reflection and metacognition [Azevedo and Hadwin 2005], which are important facets of learning, particularly in cultivating self-regulated and lifelong learners. They might also take the form of a dashboard that transforms data collected by openHTML into visualizations and other information designed to help teachers track and assess student progress.

7.3. Parting Words

Through diverse pathways and motivations, basic web development continues to serve a gateway to computation for countless people. This dissertation has aimed at improving our understanding of these first forays and exploring how to design systems that help beginners make the most of them. It presents some of the first and most substantive studies of beginners learning HTML and CSS, two of the most broadly used computing languages. My motivation for this work is not to train a legion of professional web developers. Rather, it is to leverage these moments to learn important concepts, practices, and perspectives that have ongoing benefits when people interact with software in

all of their pursuits, and to foster attitudes and identities that lead to a deepened and lifelong engagement with computing.

References

ACM, 2010. Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age. *The Association for Computing Machinery & The Computer Science Teachers Association*, pp.1–76.

ADELSON, B., 1981. Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, 9(4), pp.422–433.

AMES, C. AND ARCHER, J., 1988. Achievement goals in the classroom: Students' learning strategies and motivation processes. *Journal of Educational Psychology*, 80(3), pp.260–267.

AN, H., 2007. Designing an effective web-based coding environment for novice learners. *Innovate*, 4(1), pp.1–6.

ANDERSON, J.R. AND JEFFRIES, R., 1985. Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction*, 1(2), pp.107–131.

ANDERSON, J.R., CORBETT, A.T., KOEDINGER, K.R. AND PELLETIER, R., 1995. Cognitive tutors: Lessons learned. *Journal of the Learning Sciences*, 4(2), pp.167–207.

AZEVEDO, R. AND HADWIN, A.F., 2005. Scaffolding self-regulated learning and metacognition: Implications for the design of computer-based scaffolds. *Instructional Science*, 33, pp.367–379.

BAKER, R.S.J.D. AND YACEF, K., 2009. The state of educational data mining in 2009: A review and future visions. *Journal of Educational Data Mining*, 1(1), pp.3–17.

BARAB, S. AND SQUIRE, K., 2004. Design-based research: Putting a stake in the ground. *Journal of the Learning Sciences*, 13(1), pp.1–14.

BAYMAN, P. AND MAYER, R.E., 1983. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM*, 26(9), pp.677–679.

BEAUBOUEF, T. AND MASON, J., 2005. Why the high attrition rate for computer science students: Some thoughts and observations. *ACM SIGCSE Bulletin*, 37(2), pp.1–4.

BEN-ARI, M., 1998. Constructivism in computer science education. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 257–261.

BIEMILLER, A., 1977. Relationships between oral reading rates for letters, words, and simple text in the development of reading achievement. *Reading Research Quarterly*, 13(2), pp.223–253.

BLACKWELL, A.F., 2002. First steps in programming: A rationale for attention investment models. In Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments. pp. 2–10.

BONAR, J. AND SOLOWAY, E., 1985. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, 1, pp.133–161.

BOUSTEDT, J. ET AL., 2007. Threshold concepts in computer science: Do they exist and are they useful? In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 1–5.

BRANDT, J., DONTCHEVA, M., WESKAMP, M. AND KLEMMER, S.R., 2010. 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.

BRANDT, J., GUO, P.J., LEWENSTEIN, J., DONTCHEVA, M. AND KLEMMER, S.R., 2009. 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.

BRAUN, V. AND CLARKE, V., 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2), pp.77–101.

BRENNAN, K., CHUNG, M. AND HAWSON, J., 2011. *Scratch Curriculum Guide*,

BROWN, A.L., 1992. Design experiments: Theoretical and methodological challenges in creating complex interventions in classroom settings. *Journal of the Learning Sciences*, 2(2), pp.141–178.

BROWN, J.S., COLLINS, A. AND DUGUID, P., 1989. Situated cognition and the culture of learning. *Educational Researcher*, 18(1), pp.32–42.

CAMP, T., 1997. The incredible shrinking pipeline. *Communications of the ACM*, 40(10), pp.103–110.

CHANDLER, P. AND SWELLER, J., 1996. Cognitive load while learning to use a computer program. *Applied Cognitive Psychology*, 10, pp.151–170.

CHANG, K.S.-P. AND MYERS, B.A., 2012. WebCrystal: Understanding and reusing examples in web authoring. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 1–10.

CHI, M.T.H., 1997. Quantifying qualitative analyses of verbal data: A practical guide. *Journal of the Learning Sciences*, 6(3), pp.271–315.

COLLINS, A., 1992. Toward a design science of education. In E. Scanlon & T. O'Shea, eds. New Directions in Educational Technology, pp. 15–22.

COLLINS, A., JOSEPH, D. AND BIELACZYC, K., 2004. Design research: Theoretical and methodological issues. *Journal of the Learning Sciences*, 13(1), pp.15–42.

CONGRESS, U.S., 2007. *America COMPETES Act*,

COOPER, S., DANN, W. AND PAUSCH, R., 2000. Alice: A 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5), pp.107–116.

CORBIN, J. AND STRAUSS, A., 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, Sage Publications.

CORRITORE, C.L. AND WIEDENBECK, S., 1991. What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, 3(2), pp.199–222.

DENNING, P.J. AND MCGETTRICK, A., 2005. Recentering computer science. *Communications of the ACM*, 48(11), pp.15–19.

DENNY, P., LUXTON-REILLY, A. AND CARPENTER, D., 2014. Enhancing syntax error messages appears ineffectual. In Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education. ACM Press, pp. 273–278.

DÉSILETS, A., PAQUET, S. AND VINSON, N.G., 2005. Are wikis usable? In WikiSym. pp. 3–15.

DISESSA, A.A., 2001. *Changing Minds: Computers, Learning, and Literacy*, Cambridge, MA: MIT Press.

DORN, B. AND GUZDIAL, M., 2010a. Discovering computing: Perspectives of web designers. In Proceedings of the International Computing Education Research Conference. pp. 23–29.

DORN, B. AND GUZDIAL, M., 2010b. 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.

DUBOULAY, B., 1986. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1), pp.57–73.

ECKERDAL, A., MCCARTNEY, R., MOSTÖM, J.E., RATCLIFFE, M., SANDERS, K. AND ZANDER, C., 2006. Putting threshold concepts in context in computer science education. In Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education. pp. 1–5.

EISENBERG, M. AND PEELLE, H.A., 1983. APL learning bugs. In Proceedings of the International Conference on APL. pp. 11–16.

ERICSSON, K.A. AND SIMON, H.A., 1993. *Protocol Analysis: Verbal Reports as Data*, Cambridge, MA: The MIT Press.

FELKE-MORRIS, T., 2012. *Basics of Web Design: HTML5 & CSS3*, Boston, MA: Addison-Wesley.

FINCHER, S., TENENBERG, J. AND ROBINS, A., 2011. Research design: Necessary bricolage. In Proceedings of the International Computing Education Research Conference. pp. 1–6.

FINDLER, R.B., FLANAGAN, C., FLATT, M., KRISHNAMURTHI, S. AND FELLEISEN, M., 2002. DrScheme: A pedagogic programming environment for Scheme. *Journal of Functional Programming*, 12(2), pp.159–182.

FISCHER, G., 2004. Computational literacy and fluency: Being independent of high-tech scribes. *Strukturieren-Modellieren-Kommunizieren: Leitbild mathematischer und informatischer Aktivitäten*, J. Engel, R. Vogel, & S. Wessolowski (Eds.), pp.217–230.

FISHER, A. AND MARGOLIS, J., 2002. Unlocking the clubhouse: The Carnegie Mellon experience. *ACM SIGCSE Bulletin*, 34(2), pp.79–83.

FLANAGAN, J.C., 1954. The critical incident technique. *Psychological Bulletin*, 51(4), pp.1–33.

FLEURY, A.E., 1991. Parameter passing: The rules the students construct. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 283–286.

FORTE, A. AND GUZDIAL, M., 2005. Motivation and non-majors in CS1: Identifying discrete audiences for introductory computer science. *IEEE Transactions on Education*, 48(2), pp.248–253.

GARNER, S., HADEN, P. AND ROBINS, A., 2005. My program is correct but it doesn"t run: A preliminary investigation of novice programmers" problems. In Australasian Computing Education Conference. pp. 173–180.

GILMORE, D.J., 1990. Methodological issues in the study of programming. In J.-M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore, eds. *Psychology of Programming*. pp. 83–98.

GOLDMAN, K. ET AL., 2008. Identifying important and difficult concepts in introductory computing courses using a Delphi process. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 256–260.

GREEN, T.R.G. AND PETRE, M., 1996. Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. *Journal of Visual Languages and Computing*, 7(2), pp.131–174.

GURWITZ, C., 1998. The Internet as a motivating theme in a math/computer core course for nonmajors. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 68–72.

GUZDIAL, M., 1993. *Deriving software usage patterns from log files*, GIT-GVU.

GUZDIAL, M., 1994. Software-realized scaffolding to facilitate programming for science learning. *Interactive Learning Environments*, 4(1), pp.1–44.

HANSEN, E.J., 1998. Creating teachable moments... and making them last. *Innovative Higher Education*, 23(1), pp.7–26.

HAREL, I. AND PAPERT, S., 1990. Software design as a learning environment. *Interactive Learning Environments*, 1(1), pp.1–32.

HARTMANN, B., MACDOUGALL, D., BRANDT, J. AND KLEMMER, S.R., 2010. What would other programmers do? Suggesting solutions to error messages. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 1019–1028.

HEINONEN, K., HIRVIKOSKI, K., LUUKKAINEN, M. AND VIHAVAINEN, A., 2014. Using CodeBrowser to seek differences between novice programmers. In Proceedings of the ACM Technical Symposium on Computer Science Education. ACM Press, pp. 229–234.

HESTENES, D., WELLS, M. AND SWACKHAMER, G., 1992. Force concept inventory. *The Physics Teacher*, 30, pp.141–158.

HOLDAWAY, D., 1979. *The Foundations of Literacy*, Scholastic.

HOLLAND, S., GRIFFITHS, R. AND WOODMAN, M., 1997. Avoiding object misconceptions. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 1–4.

HRISTOVA, M., MISRA, A., RUTTER, M. AND MERCURI, R., 2003. Identifying and correcting Java programming errors for introductory computer science students. *Proceedings of the ACM Technical Symposium on Computer Science Education*, pp.153–156.

HUFF, C., 2002. Gender, software design, and occupational equity. *ACM SIGCSE Bulletin*, 34(2), pp.112–115.

HUTCHINS, E.L., HOLLAN, J.D. AND NORMAN, D.A., 1985. Direct manipulation interfaces. *Human-Computer Interaction*, 1(4), pp.311–338.

JADUD, M.C., 2005. A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1), pp.25–40.

JADUD, M.C., 2006. Methods and tools for exploring novice compilation behaviour. In Proceedings of the International Computing Education Research Conference. pp. 73–84.

JOHNSON, W.L. AND SOLOWAY, E., 1984. Intention-based diagnosis of programming errors. *Proceedings of the National Conference on Artificial Intelligence*, pp.162–168.

KACZMARCZYK, L.C., PETRICK, E.R., EAST, J.P. AND HERMAN, G.L., 2010. Identifying student misconceptions of programming. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 107–111.

KAPLAN, D.E. AND AN, H., 2005. Facts, procedures, and visual models in novices' learning of coding skills. *Journal of Computing in Higher Education*, 17(1), pp.43–70.

KAY, A. AND GOLDBERG, A., 1977. Personal dynamic media. *Computer*, 10(3), pp.31–41.

KELLEHER, C.L. AND PAUSCH, R., 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2), pp.83–137.

KLASSNER, F., 2000. Can web development courses avoid obsolescence? In Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education. pp. 77–80.

KO, A.J., 2009. Attitudes and self-efficacy in young adults' computing autobiographies. In Proceedings of the Symposium on Visual Languages and Human-Centric Computing. pp. 67–74.

KO, A.J. AND WOBBROCK, J.O., 2010. Cleanroom: Edit-time error detection with the uniqueness heuristic. In Proceedings of the Symposium on Visual Languages and Human-Centric Computing. pp. 7–14.

KO, A.J., MYERS, B.A. AND AUNG, H.H., 2004. Six learning barriers in end-user programming systems. In Proceedings of the Symposium on Visual Languages and Human-Centric Computing. pp. 199–206.

KÖLLING, M., QUIG, B., PATTERSON, A. AND ROSENBERG, J., 2003. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4), pp.249–268.

KRIPPENDORFF, K., 2004. *Content Analysis: An Introduction to Its Methodology*, Thousand Oaks, CA: Sage Publications.

KURLAND, D.M., PEA, R.D., CLEMENT, C. AND MAWBY, R., 1986. A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research*, 2(4), pp.429–458.

LANDIS, J.R. AND KOCH, G.G., 1977. The measurement of observer agreement for categorical data. *Biometrics*, 33(1), pp.159–174.

LAVE, J. AND WENGER, E., 1991. *Situated Learning: Legitimate Peripheral Participation*, Cambridge University Press.

LEE, M.J. AND KO, A.J., 2011. Personifying programming tool feedback improves novice programmers' learning. In Proceedings of the International Computing Education Research Conference. pp. 109–116.

LENHART, A., PURCELL, K., SMITH, A. AND ZICKUHR, K., 2010. Social media & mobile Internet use among teens and young adults. *Pew Internet & American Life Project*, pp.1–51.

LIM, B.B.L., 1998. Teaching web development technologies in CI/IS curricula. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 107–111.

LINN, M.C. AND DALBEY, J., 1985. Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist*, 20(4), pp.191–206.

LISTER, R. ET AL., 2004. A multi-national study of reading and tracing skills in novice programmers. In Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education. Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education, pp. 119–150.

LITECKY, C.R. AND DAVIS, G.B., 1976. A study of errors, error-proneness, and error diagnosis in Cobol. *CACM*, 19(1), pp.33–38.

LOFTUS, C., THOMAS, L. AND ZANDER, C., 2011. Can graduating students design: Revisited. *Proceedings of the ACM Technical Symposium on Computer Science Education*, pp.105–110.

LU, J.J. AND FLETCHER, G.H.L., 2009. Thinking about computational thinking. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 260–264.

MARCEAU, G., FISLER, K. AND KRISHNAMURTHI, S., 2011. Measuring the effectiveness of error messages designed for novice programmers. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 499–504.

MCCRACKEN, W.M. ET AL., 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In SIGCSE Bulletin. Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education, pp. 125–140.

MCKEITHEN, K.B., REITMAN, J.S., RUETER, H.H. AND HIRTLE, S.C., 1981. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13(3), pp.307–325.

MERCURI, R., HERRMANN, N. AND POPYACK, J., 1998. Using HTML and JavaScript in introductory programming courses. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 176–180.

MILLER, C.S., PERKOVIC, L. AND SETTLE, A., 2010. File references, trees, and computational thinking. In Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education. pp. 132–136.

NARDI, B.A., 1993. *A Small Matter of Programming: Perspectives on End-User Computing*, MIT Press.

NARDI, B.A., 1995. Studying Context: A Comparison of Activity Theory, Situated Action Models, and Distributed Cognition. In B. A. Nardi, ed. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. Cambridge, MA: MIT Press, pp. 69–102.

NELSON-LEGALL, S., 1985. Help-seeking behavior in learning. *Review of Research in Education*, 12(1), pp.55–90.

NIENALTOWSKI, M.-H., PEDRONI, M. AND MEYER, B., 2007. Compiler error messages: What can help novices? In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 168–172.

NRC, 1999. *Being Fluent with Information Technology*, National Academies Press.

OWENSBY, J.N. AND KOLODNER, J.L., 2002. Case application suite: Promoting collaborative case application in learning by design classrooms. *International Conference on Computer-Supported Collaborative Learning*, pp.505–506.

PAPERT, S., 1993. *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books.

PAPERT, S. AND HAREL, I., 1991. Situating Constructionism. In S. Papert & I. Harel, eds. *Constructionism*. Ablex Publishing Corporation, pp. 1–13.

PARK, T.H. AND WIEDENBECK, S., 2010. First steps in coding by informal web developers. In Proceedings of the Symposium on Visual Languages and Human-Centric Computing. pp. 79–82.

PARK, T.H. AND WIEDENBECK, S., 2011. Learning web development: Challenges at an earlier stage of computing education. In Proceedings of the International Computing Education Research Conference. pp. 125–132.

PARK, T.H., DORN, B. AND FORTE, A., (in press). An analysis of HTML and CSS syntax errors in a web development course. *ACM Transactions on Computing Education*.

PARK, T.H., MAGEE, R.M., WIEDENBECK, S. AND FORTE, A., 2013a. Children as webmakers: Designing a web editor for beginners. In Proceedings of the Conference on Interaction Design and Children. pp. 419–422.

PARK, T.H., SAXENA, A., JAGANNATH, S., WIEDENBECK, S. AND FORTE, A., 2013b. openHTML: Designing a transitional web editor for novices. In CHI Extended Abstracts. pp. 1863–1868.

PARK, T.H., SAXENA, A., JAGANNATH, S., WIEDENBECK, S. AND FORTE, A., 2013c. Towards a taxonomy of errors in HTML and CSS. In Proceedings of the International Computing Education Research Conference. ACM Press, pp. 75–82.

PARSONS, D. AND HADEN, P., 2006. Parson's Programming Puzzles: A fun and effective learning tool for first programming courses. In Australasian Computing Education Conference. pp. 157–163.

PEA, R.D., 1986. Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, 2(1), pp.25–36.

PEA, R.D. AND KURLAND, D.M., 1984. On the cognitive effects of learning computer programming. *New Ideas in Psychology*, 2(2), pp.137–168.

PEA, R.D., SOLOWAY, E. AND SPOHRER, J.C., 1987. The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics*, 9(1), pp.5–30.

PERKINS, D.N., HANCOCK, C., HOBBS, R., MARTIN, F. AND SIMMONS, R., 1986. Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1), pp.37–55.

PETERSON, R.F., TREAGUST, D.F. AND GARNETT, P., 1994. Development and application of a diagnostic instrument to evaluate grade-11 and -12 students' concepts of covalent bonding and structure following a course of instruction. *Journal of Research in Science Teaching*, 26(4), pp.301–314.

PIAGET, J., 1950. *The Psychology of Intelligence* 2nd ed., Routledge.

POLEY, E., 2010. RUMU Editor: A non-WYSIWYG web editor for non-technical users. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 4357–4362.

POPYACK, J.L. AND HERRMANN, N., 1993. Mail merge as a first programming language. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 136–140.

PUNTAMBEKAR, S. AND KOLODNER, J.L., 2005. Toward implementing distributed scaffolding: Helping students learn science from design. *Journal of Research in Science Teaching*, 42(2), pp.185–217.

PUTNAM, R.T., SLEEMAN, D., BAXTER, J.A. AND KUSPA, L.K., 1986. A summary of misconceptions of high school Basic programmers. *Journal of Educational Computing Research*, 2(4), pp.1–8.

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), pp.257–266.

REASON, J., 1990. *Human Error*, Cambridge University Press.

REED, D., 2001. Rethinking CS0 with JavaScript. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 100–104.

RESNICK, M. ET AL., 2009. Scratch: Programming for all. *Communications of the ACM*, 52(11), p.60.

ROBINS, A., HADEN, P. AND GARNER, S., 2006. Problem distribution in a CS1 course. In Australasian Computing Education Conference. pp. 165–173.

RODE, J.A., TOYE, E.F. AND BLACKWELL, A.F., 2004. The fuzzy felt ethnography—understanding the programming patterns of domestic appliances. *Personal and Ubiquitous Computing*, 8(3), pp.161–176.

RODRIGO, M.M.T. ET AL., 2009. Affective and behavioral predictors of novice programmer achievement. In Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education. pp. 1–5.

ROMERO, C. AND VENTURA, S., 2010. Educational data mining: A review of the state of the art. *IEEE Transactions on Systems, Man, and Cybernetics*, 40(6), pp.601–618.

ROSSON, M.B., BALLIN, J.F. AND NASH, H., 2004. Everyday programming: Challenges and opportunities for informal web development. In Proceedings of the Symposium on Visual Languages and Human-Centric Computing. pp. 123–130.

ROSSON, M.B., BALLIN, J.F. AND RODE, J., 2005. Who, what, and how: A survey of informal and professional web developers. In Proceedings of the Symposium on Visual Languages and Human-Centric Computing. pp. 199–206.

SANDERS, K. AND THOMAS, L., 2007. Checklists for grading object-oriented CS1 programs: Concepts and misconceptions. In Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education. pp. 166–170.

SANDERS, K. ET AL., 2005. A multi-institutional, multinational study of programming concepts using card sort data. *Expert Systems*, 22(3), pp.121–128.

SCARDAMALIA, M. AND BEREITER, C., 1994. Computer support for knowledge-building communities. *Journal of the Learning Sciences*, 3(3), pp.265–283.

SCHULTE, C. AND KNOBELSDORF, M., 2007. Attitudes towards computer science-computing experiences as a starting point and barrier to computer science. In Proceedings of the International Computing Education Research Conference. pp. 38–49.

SCHWILL, A., 1994. Fundamental ideas of computer science. *Bulletin of the European Association for Theoretical Computer Science*, 53, pp.274–295.

SEDIG, K., KLAWE, M. AND WESTROM, M., 2001. Role of interface manipulation style and scaffolding on cognition and concept learning in learnware. *ACM Transactions on Computer-Human Interaction*, 8(1), pp.34–59.

SHAFFER, D.W. AND RESNICK, M., 1999. "Thick" authenticity: New media and authentic learning. *Journal of Interactive Learning Research*, 10(2), pp.195–215.

SHEERAN, L., SASSE, M.A., RIMMER, J. AND WAKEMAN, I., 2002. How Web browsers shape users' understanding of networks. *The Electronic Library*, 20(1), pp.35–42.

SHINNERS-KENNEDY, D. AND FINCHER, S.A., 2013. Identifying threshold concepts: From dead end to a new direction. In Proceedings of the International Computing Education Research Conference. New York, New York, USA: ACM Press, pp. 9–17.

SHNEIDERMAN, B., 1983. Direct manipulation: A step beyond programming languages. *Computer*, 16(8), pp.57–69.

SIEGLER, R.S., 2006. Microgenetic analyses of learning. In *Handbook of Child Psychology*. Wiley, pp. 464–510.

SIEMENS, G., 2012. Learning analytics: Envisioning a research discipline and a domain of practice. In Proceedings of the International Conference on Learning Analytics and Knowledge. pp. 4–8.

SIMON, H.A., 1996. *The Sciences of the Artificial* 3rd ed., MIT Press.

SMITH, J.P., III, DISESSA, A.A. AND ROSCHELLE, J., 1993. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *Journal of the Learning Sciences*, 3(2), pp.115–163.

SOLOWAY, E., GUZDIAL, M. AND HAY, K.E., 1994. Learner-centered design: The challenge for HCI in the 21st century. *Interactions*, 1(2), pp.36–48.

SPOHRER, J.C. AND SOLOWAY, E., 1986a. Alternatives to construct-based programming misconceptions. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 183–191.

SPOHRER, J.C. AND SOLOWAY, E., 1986b. Novice mistakes: Are the folk wisdoms correct? *CACM*, 29(7), pp.624–632.

SRIDHARAN, K., 2004. A course on web languages and web-based applications. *IEEE Transactions on Education*, 47(2), pp.254–260.

STEFIK, A. AND SIEBERT, S., 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education*, 13(4), pp.1–40.

TAUB, R., BEN-ARI, M. AND ARMONI, M., 2009. The effect of CS Unplugged in middle-school students' views of CS. *Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education*, pp.99–103.

TEW, A.E. AND GUZDIAL, M., 2010. Developing a validated assessment of fundamental CS1 concepts. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 97–101.

TREU, K., 2002. To teach the unteachable class: An experimental course in web-based application design. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 201–205.

UNESCO, 2004. The plurality of literacy and its implications for policies and programmes. *UNESCO Education Sector*, pp.1–32.

VORA, P.R., 1998. Designing for the Web: A survey. *Interactions*, 5(3), pp.13–30.

VYGOTSKY, L.S., 1978. *Mind in Society: The Development of Higher Psychological Processes* 14 ed. M. Cole, V. John-Steiner, S. Scribner, & E. Souberman, eds., Cambridge, MA: Harvard University Press.

WALKER, E.L. AND BROWNE, L., 1999. Teaching web development with limited resources. In Proceedings of the ACM Technical Symposium on Computer Science Education. pp. 12–16.

WIEDENBECK, S., 2005. Factors affecting the success of non-majors in learning to program. In Proceedings of the International Computing Education Research Conference. pp. 13–24.

WING, J.M., 2006. Computational thinking. *Communications of the ACM*, 49(3), pp.33–35.

WINSLOW, L.E., 1996. Programming pedagogy -- A Psychological overview. *SIGCSE Bulletin*, 28(3), pp.17–25.

YOUNGS, E.A., 1974. Human errors in programming. *International Journal of Man-Machine Studies*, 6, pp.361–376.