# Avoiding the Turing Tarpit: Learning Conversational Programming by Starting from Code's Purpose


Kathryn Cunningham
University of Michigan
Ann Arbor, Michigan, USA
kicunn@umich.edu

Barbara Ericson
University of Michigan
Ann Arbor, Michigan, USA
barbarer@umich.edu

Rahul Agrawal Bejarano
University of Michigan
Ann Arbor, Michigan, USA
rahulab@umich.edu

Mark Guzdial
University of Michigan
Ann Arbor, Michigan, USA
mjguz@umich.edu

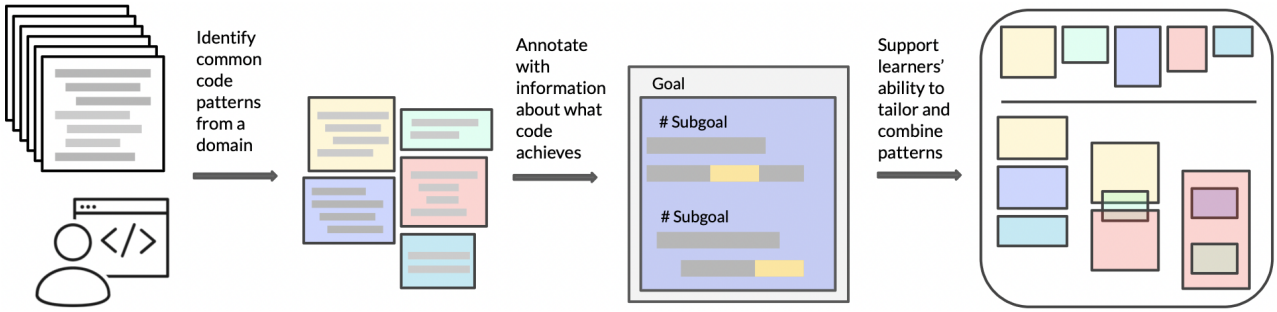




**Figure 1: In purpose-first programming, learners are supported to write and understand code easily with scaffolding based on a small number of common, domain-specific code patterns.**


## ABSTRACT

Conversational programmers want to learn about code primarily to communicate with technical co-workers, not to develop software. However, existing instructional materials don't meet the needs of conversational programmers because they prioritize syntax and semantics over concepts and applications. This mismatch results in feelings of failure and low self-efficacy. To motivate conversational programmers, we propose *purpose-first programming*, a new approach that focuses on learning a handful of domain-specific code patterns and assembling them to create authentic and useful programs. We report on the development of a purpose-first programming prototype that teaches five patterns in the domain of web scraping. We show that learning with purpose-first programming is motivating for conversational programmers because it engenders a feeling of success and aligns with these learners' goals. Purpose-first programming learning enabled novice conversational programmers to complete scaffolded code writing, debugging, and explaining activities after only 30 minutes of instruction.




*CHI '21, May 8–13, 2021, Yokohama, Japan*



ACM ISBN 978-1-4503-8096-6/21/05…$15.00
https://doi.org/10.1145/3411764.3445571



## CCS CONCEPTS

• **Social and professional topics** → **Computing education**.

## KEYWORDS

conversational programmers, programming plans, scaffolding, motivation, computer-supported instruction

**ACM Reference Format:**
Kathryn Cunningham, Barbara Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. 2021. Avoiding the Turing Tarpit: Learning Conversational Programming by Starting from Code's Purpose. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3411764.3445571


## 1 INTRODUCTION

Enrollment in undergraduate computing courses is undergoing exponential growth, the majority of which is driven by students from majors besides computer science [5]. Non-computer science majors often want to learn programming for reasons other than preparation for a career in software development. One goal is to become a "conversational programmer," someone who knows enough about technical topics to communicate with co-workers, but doesn't often program in their work [8, 9]. Conversational programmers can be found in a wide variety of careers, including designers, product managers, executives, and entrepreneurs [8, 68].

However, existing tools do not meet the learning needs of conversational programmers. Wang et al. found that after trying formal and informal methods to learn programming, including courses,

tutorials, and forums, conversational programmers reported a lack of benefit and feelings of failure [68]. In particular, conversational programmers found that these programming learning resources didn't align with their needs: instruction focused too much on syntax and logic, and not enough on how to apply code to solve problems [68]. Indeed, many theories of programming instruction recommend an early focus on a language's semantics, often in the context of "toy" problems, reasoning that such content will prepare learners to have a deep understanding of programs in any domain [35, 59, 71]. For conversational programmers, it appears that this learning approach leads into a "Turing tarpit", where *"everything is possible, but nothing of interest is easy"* [47].

Conversational programmers have learning motivations that are seemingly in conflict. They want to understand code that is authentic to the work of real programmers [8, 68], which is often complex. At the same time, they want to avoid a close study of syntax and execution flow in favor of conceptual and application-focused understanding [68]. Learning approaches designed for future programmers don't work, but neither do approaches designed for non-programmers. These approaches use techniques like programming-by-demonstration (e.g. [7]) or visual programming (e.g. [29]) that differ markedly from the text-based programming of professional programmers and would likely be viewed as inauthentic.

To meet the needs of conversational programmers, Wang et al. recommend that tools for these learners should avoid syntax and logic, demonstrate a clear connection to an application context, require only brief learning time, and minimize code writing [68]. We answer this call by describing and implementing programming learning activities where conversational programmers can quickly and easily create or understand authentic and meaningful code. This approach requires new technical supports that enable conversational programmers to focus on code's *purpose* rather than its semantics during programming learning.

## 1.1 Summary of contributions

*1.1.1 Formative study to understand needs of conversational programmers.* Through a series of focus groups and a survey, we found that conversational programmers appreciate extra support when learning to program, and value general understanding over a focus on detail.

*1.1.2 Introduction of a novel learning approach.* To meet these needs, we introduce **purpose-first programming**, a brief, authentic, and purpose-driven learning approach designed to motivate aspiring conversational programmers. By focusing learning on a small number of common patterns in a domain-specific context, learners are supported in quickly creating and understanding code that reflects expert practice. Scaffolds (assistive mechanisms [31, 67, 70]) provide guidance as learners write, debug, and explain code by emphasizing the purpose of code patterns and ways they can be modified.

*1.1.3 Design of a proof-of-concept system and curriculum.* To investigate the effectiveness of purpose-first programming for motivating conversational programmers, we implemented the approach in a *proof-of-concept* purpose-first programming curriculum that teaches five patterns in the domain of web scraping. This system provides information, structures, and real-time feedback that highlights how code achieves goals in this domain.

*1.1.4 Findings from an in-lab usability study.* We evaluated the curriculum with seven novice conversational programmers. We find that learning with purpose-first programming is motivating for conversational programmers because it engenders a feeling of success and aligns with these learners' goals. Purpose-first programming learning enabled novice conversational programmers to complete scaffolded code writing, debugging, and explaining activities after only 30 minutes of instruction.

# 2 BACKGROUND

## 2.1 Conversational programmers have particular learning needs

Conversational programmers were first named and identified by Chilana et al. in a study of first-year students in a management engineering program [8]. Instead of preparing to perform end-user programming in their chosen field, these students wanted a basic understanding of programming so they could communicate with co-workers in careers such as project management, business leadership, and entrepreneurship. While conversational programmers expressed a low self-efficacy for programming, they were interested in future programming learning and preferred to learn a more challenging industry-level language (Java) than a language designed for non-programmers (Processing [23]), perceiving Java as more useful, practical, and marketable.

Later work by Chilana, Singh, and Guo surveyed non-programmers at a large technology company and found a similar desire for "basic" and "big picture" understanding of programming that can support "high level" conversations with developers and clients [9]. Fifty-four percent of respondents reported taking at least one formal programming course, even though they did not major in computer science or engineering, and 43% had spent time learning programming on the job. In advice for future conversational programmers in the software industry, 48% of respondents recommended that students learn some programming.

Wang et al. interviewed a wide range of conversational programmers who had recently attempted to learn programming with formal or informal methods, and found that most expressed feelings of failure, believing they didn't benefit from their learning experience [68]. In an analysis of the reasons that existing formal and informal programming learning didn't meet the needs of conversational programmers, Wang et al. concluded that instruction typically focused on syntax and logic and didn't provide appropriate conceptual and application-related content. Since conversational programmers have many facets to their careers that don't involve programming, they have limited time to learn about code, resulting in a need to quickly understand accurate and relevant information about what code can do for them.

## 2.2 Programming tools for non-developers avoid industry-standard code

A wide variety of technologies have been developed to make programming more accessible to non-programmers. Such systems often involve block-based or other visual interfaces that make the process of programming easier by reducing the potential for errors.

Block-based languages like Scratch [37, 38] eliminate the need to remember syntax since the jigsaw-like shapes for commands and functions fit together in an intuitive way. Snap! [29] is another example of a block-based programming language, with more features than Scratch, including first class functions and support for multimedia. Helena [7], a block-based language for web scraping tasks, has helped sociologists, engineers, and public policy researchers obtain data from websites.

A community of practice perspective [34] can explain why these tools don't meet the needs of conversational programmers. Lave and Wenger described how desire for learning is motivated by the alignment of learning activities with the tasks performed by people in the communities learners want to join [34]. Sociologists value data collection, and so learning Helena is relevant to the practices of sociologists. Conversational programmers, on the other hand, want to understand tasks that software developers complete [8, 9, 68]. Weintrop showed that even though Snap! allowed learners to use advanced programming concepts like first class procedures, high school students doubted its authenticity since text-based languages are standard in the software industry [69]. Use of industry-standard languages and text-based programming may demonstrate to conversational programmers that what they are learning is authentic to their future careers.

## 2.3 Plans may be a more motivating way for conversational programmers to think about code

Soloway and his students used schema theory to identify *programming plans* in the 1980's [53, 57, 58]. Plans are chunks of code that achieve particular goals, like guarding against erroneous data or summing across a collection. There is evidence that both novice and expert programmers think about code in terms of plans [56]. Student errors can be explained in terms of misunderstanding plans [55, 63] and errors in composing plans [62, 64]. Over time, error rates on plans decrease with practice [61], while error rates on syntactic structures does not [50], which suggests that plans better describe how students learn programming than syntax structures.

According to expectancy-value theory, motivation for an activity is explained by expectancy of success in the activity and subjective value for the activity [15]. Conversational programmers found that existing instructional materials typically focus on lower-level details like syntax and execution flow [68]. Closely tracking code's execution requires a high cognitive load [12], resulting in a low expectancy of success for some learners [11]. When thinking about code in "chunks" [22] of programming plans, the difficulty of understanding code may be reduced, increasing expectation of success.

Activities designed to help learners understand syntax and execution flow typically involve problems intentionally stripped of context, so learners can focus on code semantics "without distraction" [36]. For conversational programmers, such problems have low utility value [16] because the connection to activities in the workplace is unclear [68]. By contrast, programming plans associate a code pattern with a goal relevant to the user, making the purpose of code evident. Domain-specific code plans have an even clearer connection between code and application, potentially resulting in a high value for plan-based learning.

Even approaches to teach non-programmers about "computational thinking" can focus on the structures of a programming language rather than what code can achieve. While definitions of computational thinking are debated, it often involves general-purpose programming ideas, like 'selection' and 'repetition', or practices, like 'debugging' or 'remixing' [4]. Computational thinking is typically described as a general skill that can be applied in any programming situation. However, to meet the needs of conversational programmers as described by Wang et al. [68], a learning approach needs to clearly communicate how content is applicable to a domain area.

Students programming in terms of high-level, domain-specific code plans have successfully solved programming problems that students programming in more traditional languages have not. The Rainfall Problem has been challenging students for over 30 years, with most studies finding less than 20% of students are able to solve it successfully [52]. Fisler found that she could reliably get most of her students successfully solving the Rainfall Problem by using a high-level functional programming language [20]. Fisler explains much of the success due to mapping the high-level functions to domain-specific plans that were easily composed by the students into a successful solution [21].

## 2.4 Other systems have provided plan- or example-based support

The GPCeditor was a programming tool for students to support them in learning Pascal programming through the specification of goals and plans [26]. Students learned plans which they then transferred into a more traditional Pascal IDE. SODA extended the GPCeditor with support for program design and software engineering practices [32]. The GPCeditor and SODA focused on traditional introductory programming learning, rather than supporting programming learning in the context of a domain.

Emile provided adaptable scaffolding for building physics simulations through HyperTalk programs [25]. Students in Emile constructed programs out of plans with *slots* which allowed for specifying a plan for a particular purpose (e.g., to generate accelerated motion for a given velocity and acceleration source for a given object) through a supported process. Students still did not have to learn the language syntax and semantics to be able to achieve their purpose (the construction of physics simulations). Students in Emile did learn physics, which suggests that a plan-based approach to learning programming can lead to learning within the purpose domain.

There is a long history of providing purpose-oriented support in the form of examples or cases. The *minimal manual* approach of Carroll and others [6] is built around providing examples of the processes for common tasks. The examples are indexed by task, like "how I delete text in my document?" or "how do I add a button to my screen?" Evaluations of minimal manuals show that they are successful (in terms of user productivity and satisfaction), and are most successful when steps have to be inferred by the user [2]. Minimal manuals have been used successfully to help non-professional programmers succeed at programming tasks [1].

# 3 FORMATIVE STUDY: UNDERSTANDING CONVERSATIONAL PROGRAMMERS IN AN INFORMATION MAJOR

Prior research profiled conversational programmers in a management engineering major [8], but conversational programmers in other undergraduate majors have not yet been studied. A growing number of universities now offer Information majors to undergraduates, covering content like data science, user interface design, and other topics at the intersection of human activity and technology [13]. Such programs seem likely to attract conversational programmers, who seek careers where they work with both technology and people [8].

To understand the goals and learning preferences of conversational programmers studying Information, we performed a series of focus groups and a follow-up survey with students taking a secondary data-oriented programming course taught by the School of Information at a large university in the Midwestern United States.

## 3.1 Focus group results

Across four focus groups, we talked to eleven students (eight female, three male) about their goals with programming and what learning approaches they felt best prepared them for their future. We provided prototypes of Purpose-first programming activities as probes [3] to gather feedback. The focus groups lasted 30 minutes, and participants received a $25 Amazon gift card. Transcripts of the focus groups were analyzed to identify "personas" [10] representative of students' goals and preferences. We found two personas: the *conversational programmer* and the *analyst*.

*3.1.1 Connie the conversational programmer.* Connie is interested in a career in "user experience design" (P8), "project management" (P11), or "digital strategy" (P9). She wants to be "informed about coding" (P8) in order to "understand what the coders are doing and what they can and can't do" (P9) and "direct what the final product should look like" (P8). But, she "[does]n't want to be the one actually coding" (P9). She wants to pass information "off to my coder" (P10) who will do most of the programming. She also believes that that coding knowledge "sets you apart" (P8) from other majors and helps "establish yourself as a very credible person" (P11) in the workplace.

As far as learning preferences, Connie is "more interested in knowing that I can understand what code is doing than writing it myself" (P9). She wishes there were "more questions where you just look at code and choose the right answer, instead of having to do it yourself all the time" (P10). She wants to learn in a way that helps her understand "overall what the process is like to accomplish certain things" (P9). She appreciates problems that "give you a basic structure to start with" (P11), like mixed-up code problems [46]. When additional help is provided, like guiding comments in the code, Connie is "not as overwhelmed" (P8).

*3.1.2 Alyssa the analyst.* Alyssa wants a career like "data analyst" (P1) or "business analyst" (P7), where she can "use programming to solve problems that people are facing" (P6). She "[does]n't want a job where I have to heavily do programming" (P1), preferring "a mixture of programming and not really programming" (P6).

When learning, Alyssa thinks that "writing code, at least once you know all the basics, is really helpful, because then you can recall from memory" (P7). She feels that "being able to break it down and see what each individual line of code does is really helpful" (P2). She appreciates extra support "prior to starting writing a code just because I think it breaks it down much better and it helps me understand what the code is actually doing" (P2). However, after getting some practice, "writing your own code might be more helpful" (P6).

Alyssa took an introductory programming course from the computer science department, but felt she was "coding the project just to code the project" (P6) and "didn't understand how any of the coding [she] was learning applied to real life situations" (P2).

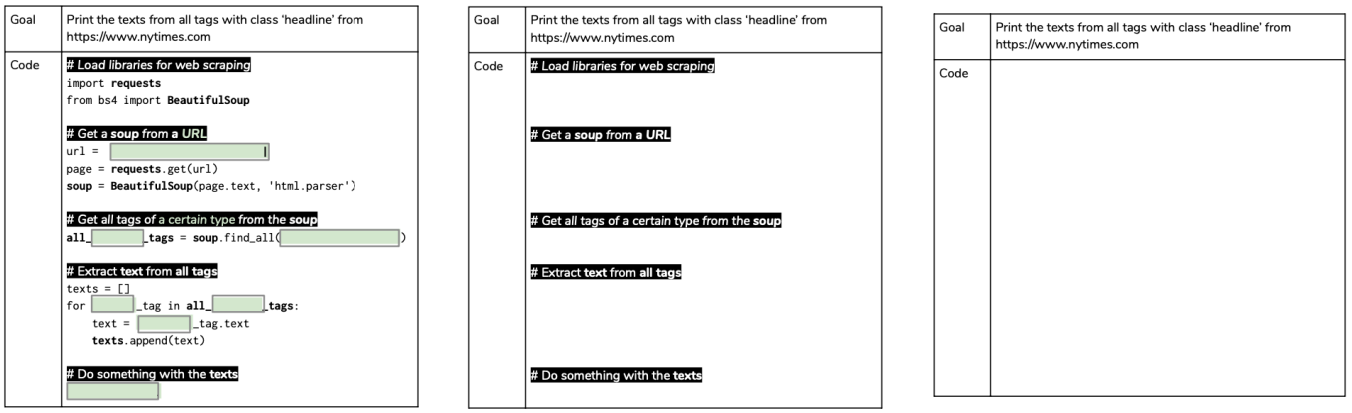
## 3.2 Survey results

We performed a survey of students in the same programming course to understand if our findings from the focus groups generalized. Forty students responded (29% of total enrollment). Participants were offered a 1 in 5 chance to win a $10 Amazon gift card.

Using responses about participants' future career goals and planned job responsibilities, we split respondents into conversational programmers and non-conversational programmers. Participants who planned a future career as a software developer, analyst, or who mentioned programming as a key job responsibility were not considered conversational programmers (n=19). Participants who planned a future career as a manager, designer, or who described another position (e.g. CEO) and did not list programming as a key responsibility were considered conversational programmers (n=18). Thus, we are comparing the comments between participants whom we classify as conversational programmers (like our Connie persona) and those whom we classify as non-conversational programmers (including analysts like our Alyssa persona and also computer science majors). Three respondents did not complete the questions about future goals and were not included.

Participants were asked to rank five programming learning activities (reading code, writing code, testing code, modifying code, and solving mixed-up code problems) by their helpfulness in preparing them for their future. Conversational programmers most often ranked code reading as the most helpful activity (8/16, 50%), while code writing was the activity non-conversational programmers most preferred (9/17, 53%). Non-conversational programmers overwhelmingly found mixed-up code problems (also known as Parsons problems [46]) to be the least helpful (15/17 ranked them last), while conversational programmers were more mixed, with less than half ranking them last (7/16).

Participants were asked to rank the usefulness of different types of code writing activities that provided different levels of support: writing code from scratch (no scaffolding), writing code with guiding comments (some scaffolding), and filling in parts of nearly completed code (high scaffolding) (See Figure 2). Both groups ranked writing code with guiding comments (some scaffolding, Figure 2b) as the most helpful (11/16 of conversational programmers, 14/18 of non-conversational programmers). However, non-conversational programmers overwhelmingly ranked filling in mostly completed code (high scaffolding, Figure 2a) as the least helpful (16/18), while




**(a) High scaffolding (fill-in)** **(b) Some scaffolding (subgoals provided)** **(c) No scaffolding (code from scratch)**

**Figure 2: Survey respondents were asked to rank and reflect on the usefulness of these code writing activities with different levels of scaffolding (assistance). The directions for all activities were: "Complete the code that achieves the goal".**

**Table 1: Difference in attitudes about the highly scaffolded code writing problem (shown in Figure 2a) between conversational programmers and non-conversational programmers.**

| Attitude | p-value | Conversational programmers who agree or strongly agree | Non-conversational programmers who agree or strongly agree |
|---|---|---|---|
| *"This type of problem is less overwhelming because if I had to do it on my own I would have freaked out a little bit trying to remember what syntax to use, and what structure."* | 0.560 | 62.5% (10/16) | 55.6% (10/18) |
| *"I feel like if code is always provided in problems like this, I will just end up forgetting to include things when I'm actually writing my own code."* | 0.036* | 33.3% (5/15) | 72.2% (13/18) |
| *"This type of question helps me understand how certain types of code should be structured."* | 1.0 | 62.5% (10/16) | 83.3% (15/18) |
| *"I think if it's your first time looking at a problem like this, something with this structure would be more useful. But if you've been looking at it and working on it for a little bit, then writing your own code would be more helpful."* | 0.046* | 50.0% (8/16) | 94.4% (17/18) |
| *"I want to solve problems from scratch without being given any hint to what the final solution should be."* | 0.103 | 18.8% (3/16) | 38.9% (7/18) |
| *"I prefer doing practice like this prior to writing a code because I think it breaks it down and it helps me understand what the code is actually doing."* | 0.837 | 68.8% (11/16) | 83.3% (15/18) |

conversational programmers were more mixed about their preferences (4/16 ranked high scaffolding as the most helpful, 6/16 ranked it as the least helpful).

This difference in preference about scaffolding was also evident in survey participants' responses to Likert scale questions about their attitudes towards problems with this high level of scaffolding (see Table 1). We found that agreement with certain attitudes expressed by focus group members was significantly different between the two groups, according to a Mann-Whitney U test. While both groups agreed that a code writing problem with high scaffolding was less overwhelming and helpful preparation for writing code, non-conversational programmers were more likely to be concerned about their ability to write code on their own without additional supports.

## 3.3 Conclusions from the formative study

Conversational programmers and non-conversational programmers' differing goals inform the ways that they want to learn about code. Conversational programmers desire a basic and conceptual understanding of code, while non-conversational programmers want the ability to write code on their own. As a result, these two groups differ in the learning activities they value, and the amount of help they want while completing programming tasks. Conversational programmers prioritize code reading activities, and are welcoming of additional scaffolding during programming learning. Non-conversational programmers are more skeptical about heavy support while learning, as they value being able to write code on their own.

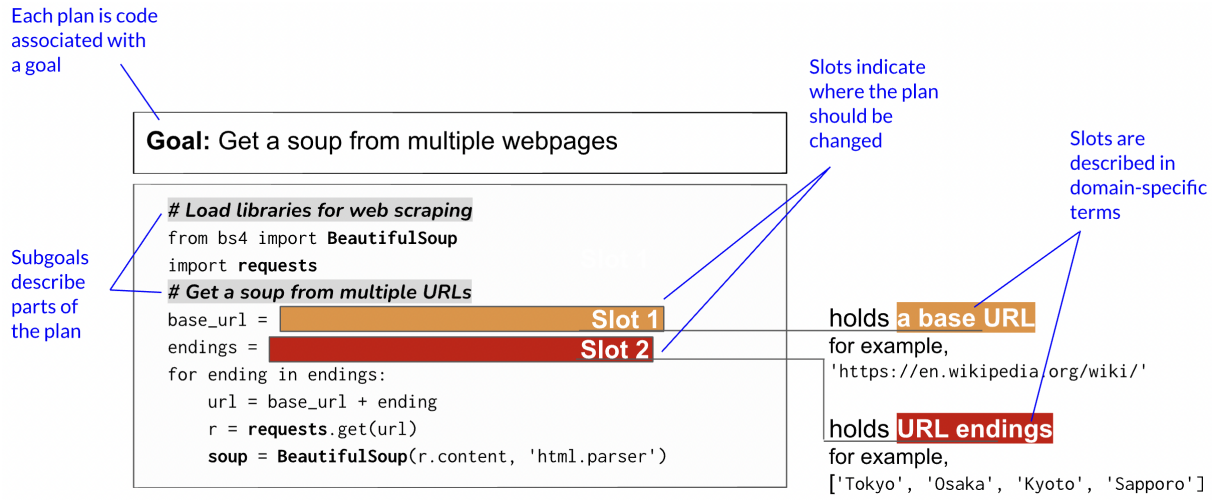



**Figure 3: An example plan from the domain of web scraping. This plan achieves an authentic goal in its domain, and consists of multiple subgoals and slots. The slots define the space of relevant domain knowledge needed to work with this plan.**

## 4 DEFINING PURPOSE-FIRST PROGRAMMING

Through our formative study, we found that conversational programmers appreciate extra support when learning to program, and value general understanding over a focus on details. As P9 said,

> "Even if I'm not necessarily going to remember how to do every little thing, or maybe it takes a lot of help for me to be able to accomplish something in the code, at least what I'm taking away from it more than anything is how a computer works, how overall what the process is like to accomplish certain things."

To meet the needs of conversational programmers, we propose *purpose-first programming*, a new learning approach designed to meet the needs of conversational programmers by emphasizing what code achieves while avoiding details of syntax and semantics. The core units of knowledge in purpose-first programming are *programming plans* [54, 56] drawn from the work of programmers in a particular domain. *Purpose-first scaffolds* help learners learn about and work with these plans so they can understand the purpose of authentic code and write new programs quickly. A purpose-first programming learning experience is driven by three design goals: it should be **brief**, it should prioritize code's **purpose**, and it should be **realistic** with respect to expert practice.

### 4.1 Expanding the definition of a programming plan to serve instructional needs

A programming plan is a commonly used code pattern, associated with the goal the code achieves [54, 56]. Programming plans are a powerful concept that integrates schema theory into the fields of computing education and program comprehension. However, in the literature, the definition of a "programming plan" is varied and sometimes vague.

We propose the following more precise definition of a programming plan to support instruction, assessment, and code writing during purpose-first programming.

*4.1.1 A plan is a frame with slots.* The power of plans is that they can be re-used by programmers over and over again. The choice of what to modify and what to keep the same is crucial to applying a plan successfully. We define a plan as a frame [42, 51] that contains parts that can't be modified, and parts that can. The areas of a plan that can be changed are called "slots". While prior plan editors allowed only numbers or strings to fill a slot [25, 26], we will allow slots to contain not only literal values, but also other code. This allows plans to be nested.

*4.1.2 A plan has subgoals as well as a goal.* Since plans often contain many lines of code, additional *subgoals* can likely facilitate plan tracing, understanding, and integration. Subgoals describe what a small section of code achieves [41]. Research has shown that adding subgoals to code leads to better learning (improved retention and transfer) in less time than without the subgoal labels [40, 41]. Evidence suggests that subgoal labels support self-explanation behavior, in that the labels give students the language to use when explaining the programs to themselves [39, 41, 44]. Subgoal labels are also effective in helping students solve and learn from mixed-up code (aka Parsons) problems [43].

In purpose-first programming, each subgoal uses variables that were defined previously, and/or produces variables to be used in later subgoals. By tracing the input and output to each subgoal, learners can trace purpose-first code at a higher level of abstraction than evaluating each variable assignment and control structure [59]. Subgoals also suggest the way that plans can be combined: code from another plan can be added in a way that satisfies the subgoal label. In purpose-first programming, subgoal labels use language from the domain to describe what the subgoal achieves.

*4.1.3 Slot contents are described with domain-specific concepts.* Where a compiler sees only a string variable, humans understand

that the variable represents a URL, a DNA sequence, or an address. Domain-specific concepts connect code to action in the real world. Similarly, domain knowledge is essential for understanding a plan's goal and how slots can be changed. In purpose-first programming, the content of slots is described in domain-specific terms. As a result, the types of objects that can go into slots provide a list of prerequisite knowledge for using a plan.

### 4.2 Providing "glass-box" scaffolding to support learners as they work with plans

In "purpose-first" programming tasks, learners will always work with authentic code patterns used by practitioners. This "chunking" [22] of meaningful code is a scaffold that allows reasoning about code to occur more easily and aligns with what we know about the ways experts understand code [49, 56]. Instead of writing code from scratch, purpose-first programmers will *tailor* and *combine* plans. Instead of tracing code line-by-line to debug or understand how code works, learners will *infer* code behavior across larger chunks of code.

However, novice programmers don't recognize or use programming plans as readily as experts [56]. They need support to recognize plans, combine plans, and identify which parts of plans should be changed. *Purpose-first scaffolds* support these processes, making understanding code and writing code easier.

A scaffold is an assistive mechanism that helps a learner complete a task they wouldn't be able to do without added support [67, 70]. Designing scaffolding for use by conversational programmers requires balancing competing motivations: (1) these learners have low self-efficacy [8] and desire assistance when coding, but (2) they also want to learn complex content that is authentic to technical workplaces [8].

"Glass-box scaffolds" can help these learners achieve both goals. A glass-box scaffold provides support to help learners achieve tasks, while not obscuring more complex, lower-level processes [31]. Learners can then focus on higher-level learning goals, while viewing information that can help with more complex tasks, if desired. In purpose-first programming, the full code will be available to the learners, demonstrating both authenticity and providing an on-ramp to future learning. At the same time, purpose-first scaffolds will draw attention to key aspects of plans, allowing novices to complete tasks while keeping cognitive load low [65].

## 5 DESIGNING THE PURPOSE-FIRST PROGRAMMING PROOF-OF-CONCEPT CURRICULUM

### 5.1 Identifying authentic, domain-specific programming plans

In prior plan identification work, plans were most often developed in the context of problems typical to an introductory programming course (e.g. [54, 64]). In purpose-first programming, plans are chosen to achieve domain-specific goals authentic to the work of programmers. In order to identify these plans, the corpus from which plans are drawn should reflect the workplace, not the classroom.

*5.1.1 Choosing a domain.* Web scraping, or extracting data from websites, is a common task for data scientists. During the formative study, we found that web scraping tasks also have disciplinary authenticity for students interested in conversational programming careers like user experience design and project management. These learners felt that the coverage of HTML and other web topics was relevant to their goals.

Web scraping requires the use of multiple feature-laden libraries (in our examples, we use the `BeautifulSoup` and `requests` Python libraries). Even basic commands in these libraries involve complex mechanisms, including instantiation of objects, list iteration, and method calls that return custom object types. At the same time, web scraping incorporates similar patterns that are slightly modified to match each page's layout. The fact that web scraping is semantically complex but "planfully" simple makes it ideal for use in the proof-of-concept curriculum.

*5.1.2 Identifying plans.* We collected a corpus of BeautifulSoup web scraping files from a dataset of Github repositories collected in October 2019 [14]. The corpus included Python files containing the BeautifulSoup constructor and at least one instance of the BeautifulSoup method `find()` or `find_all()`. After removing duplicate files and files consisting of only unit tests, 100 files remained. We generated our initial set of plans after an analysis of the first 50 of these files.

We sought feedback on the authenticity of the plans in interviews with two experts who have used the BeautifulSoup web scraping library in their work. Both experts confirmed that all the plans were useful in web scraping, and that they have used the plans in their work. They also felt that useful web scraping tasks could be achieved using only the plans, although they described several tasks that would require use of additional plans, such as web crawling and scraping image files. The experts also provided suggestions for updates to deprecated libraries.

### 5.2 Platform

We developed the purpose-first programming proof-of-concept curriculum using Runestone, an open-source ebook platform with interactive feedback [19]. Runestone is a popular platform for introductory programming learning that currently serves over 25,000 students a day. Instructors can create custom courses on Runestone that incorporate a wide variety of interactive components, such as runnable code and mixed-up code (Parsons [46]) problems.

### 5.3 Designing purpose-first scaffolds

Shifting the focus of instruction from syntax and semantics to programming plans requires new interfaces for learning about code and completing programming tasks. The ways that students interact with code will be different. Instead of writing code by typing in text, purpose-first programming learners will tailor and compose plans. Instead of tracing changes in variables in memory, learners will trace with goals and subgoals. Purpose-first scaffolds should keep code fully visible, but direct attention towards key plan components, like slots, goals, and subgoals.

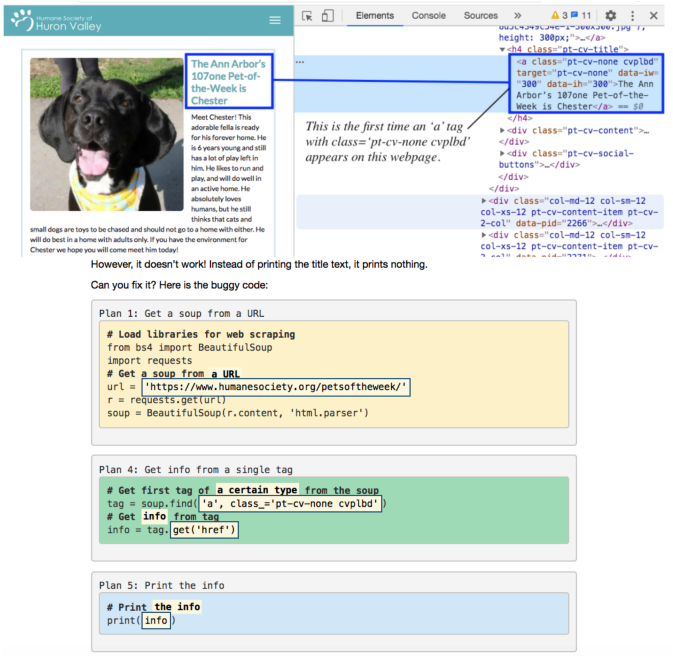
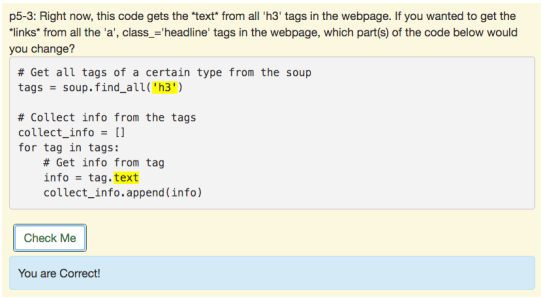



**Figure 4: Slot highlighting, plan goals, and subgoals assist learners as they debug this code.**




**(a) Learners *remember* parts of the plan that should be changed**


**(b) Learners *apply* their knowledge to complete a plan**

**Figure 5: Practice activities contain subgoal label scaffolding, and focus on knowledge about plan slots.**

*5.3.1 Highlighting demarcates plans and slots.* While existing highlighting features emphasize code's *syntactic structures*, we use highlighting to show code's *plan structures* (see Figure 4). Perceptual grouping of related symbols can improve performance in tasks like algebraic calculations [33]. This suggests that highlighting code that is part of the same plan and highlighting slots in a way that associates them with their natural language description in a subgoal may improve problem-solving.

*5.3.2 Practice activities support learners in tailoring plans.* Typical programming learning activities include writing code and predicting the result of code execution [66]. Purpose-first programming suggests new activities: identifying changeable parts of plans (see Figure 5a) and filling in plan slots to achieve a goal (See Figure 5b). These activities are quick to complete and minimize the opportunity for errors while focusing learners' understanding on key areas of code. Instruction and activities describe code's purpose in language similar to that of plan goals and subgoal labels.

*5.3.3 Examples and plan instruction are linked.* To further emphasize code's purpose, all instruction in the proof-of-concept curriculum is situated in examples. Across two examples of complete programs, learners study all five plans. To make the connection between plans and their context of use more clear, examples are interactive: students click on each plan in the example program to visit its instructional page and learn about the plan in detail (see Figure 6).

*5.3.4 Staged code writing supports learners in assembling and tailoring plans.* In a traditional editor, learners type code character by character. In purpose-first programming, learners make use of plan structures to assemble and tailor plans.

Drawing inspiration from mixed-up code (aka Parsons) problems, we developed a code writing activity in three parts (see Figure 7). First, learners pick from a bank of plan goals and drag selected goals into the correct order. Next, learners repeat this task, but with plan code instead of goals. Finally, learners fill in the slots in the code they have assembled.

## 5.4 How this prototype meets the design goals

*5.4.1 Purpose-first programming is brief.* Our prototype curriculum offers a *brief* learning experience where students can learn and apply the basics of web scraping in about an hour. This short timeline is possible because no content is taught unless it helps learners understand and modify one of a small number of plans. For example, the official BeautifulSoup documentation begins by explaining that a soup object is a nested data structure [48], but in our purpose-first programming curriculum it isn't necessary to explain what a soup object is. That level of detail isn't necessary to assemble and tailor plans.

Our proof-of-concept curriculum is also brief because of technical supports. Links to plan information are available when solving problems, so relevant examples are readily accessible. Key information is made salient through annotations and highlighting. Learners don't write code from scratch, and instead tailor and arrange code using the drag-and-drop interface.

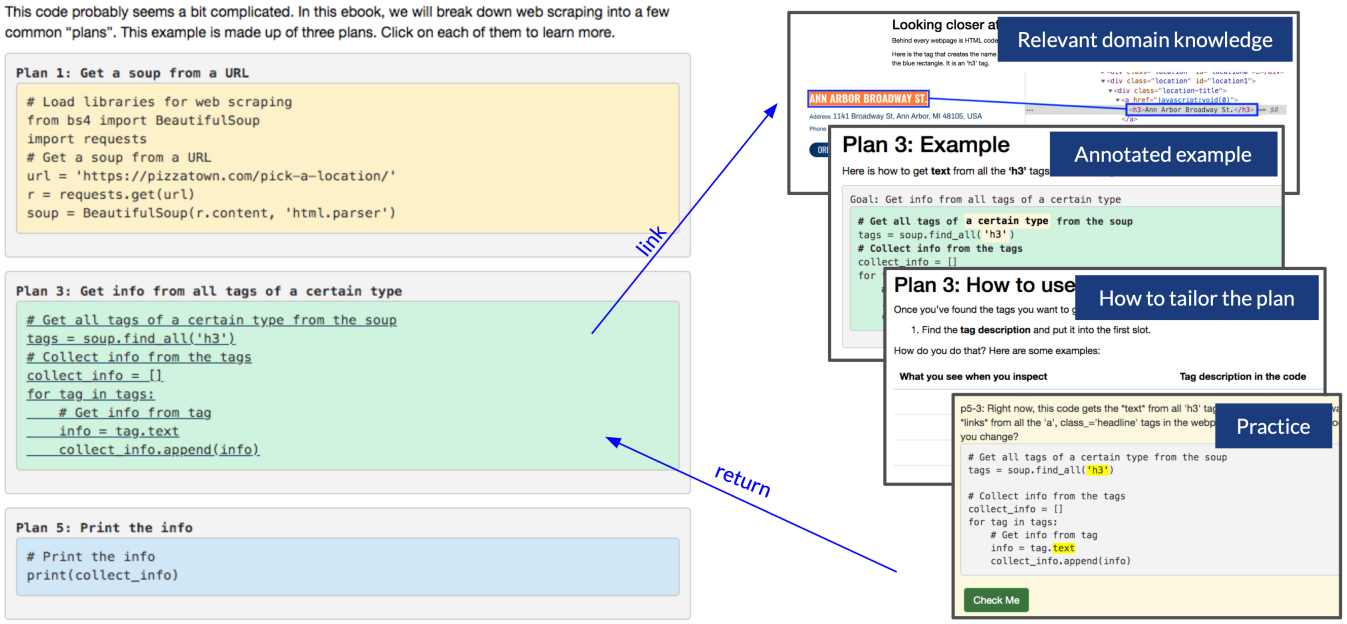



**Figure 6: Instruction is situated in interactive examples. Learners first view and run a complete program example, then learn about how each plan contributes to the full program.**

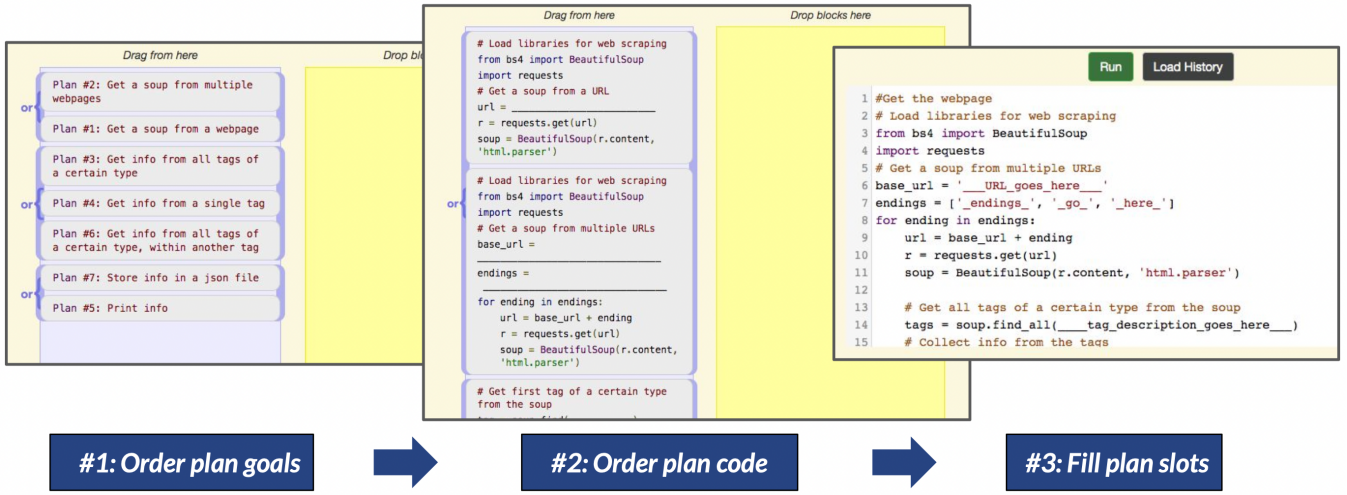



**Figure 7: Writing code takes place in stages. Learners first assemble plans, and then fill in plan slots.**

*5.4.2 Purpose-first programming prioritizes purpose.* Conversational programmers value "high-level", conceptual, and application-oriented information about code [8, 9, 68]. This purpose-first programming curriculum makes code's purpose clear at three different levels.

At the level of entire programs, learners only see examples that achieve a meaningful web scraping purpose. Examples are not "toy" problems, but programs that could potentially be used by data scientists. Examples are runnable, so learners can be assured that programs truly work as intended.

At the level of individual plans, each plan is introduced in the context of a full program, showing its relevance to useful code. By grounding plan instruction in these examples (see Figure 6), learners can see how that plan's goal contributes to the program's goal, and understand the purpose the plan serves in the example.

At the level of the code within plans, annotation with subgoals creates a natural language layer on top of code that explicitly describes code's purpose. Code is available to view, but can be ignored in favor of these descriptions of code's purpose.

*5.4.3 Purpose-first programming is realistic to the work of programmers.* Two features communicated the disciplinary authenticity of the content in the prototype. First, learners were informed that the curricular content was designed based on an analysis of Github files and expert interviews. Second, code examples and activities incorporated runnable code that scraped known websites, such as RateMyProfessors.com, the local humane society, and faculty homepages. As a part of both instruction and assessment, participants ran code and saw results returned.

# 6 EVALUATION

## 6.1 Study design

To evaluate the purpose-first programming proof-of-concept curriculum, we performed an in-lab usability study with students planning to become conversational programmers. First, participants completed instructional content where they learned five new plans. A researcher guided them through the instructional content, reading instructional text aloud and answering questions about the exercises as requested. Next, participants completed writing, debugging, and explaining problems that combined plans in ways not seen in the instructional content. Participants were asked to perform a concurrent thinkaloud while completing these tasks. Finally, participants completed a semi-structured interview to understand each participants' feelings of success during the activity and value for this type of learning.

The study was conducted over video call, and was recorded and transcribed. Each session lasted 90 minutes in total. Each participant received a $50 Amazon gift card. The study took place in the week before the fall semester began.

## 6.2 Recruitment and participants

To target conversational programmers like those profiled in the formative study, we recruited participants enrolled in a secondary data-oriented programming course (the same course studied in the formative study) who didn't plan on becoming software developers or analysts. Participants were eligible if they had no prior experience with the BeautifulSoup web scraping library. Since the study took place before the start of the semester, participants had not yet seen any of the content from the course, and it had often been several months since participants had programmed last.

Seven students participated in the study. All participants were female. Five participants were majoring in Information in the User Experience track, and two were majoring in Business. Six participants had taken one prior programming course (either an introductory data-oriented Python course in the information major (5) or an introductory C++ course in the Computer Science major (1)). One participant had taken both the data-oriented Python course and the C++ course. Participants expressed career goals of user experience designer, product manager, and/or product designer.

## 6.3 Learners were able to complete scaffolded writing, debugging, and code explanation tasks

After completing the instructional content (average time 31.4 minutes), learners attempted scaffolded code writing, debugging, and explanation activities. These activities used different plan combinations than learners had seen in instructional content examples. Participants could reference plan instructional content while solving these problems.

**Table 2: Participants' success and self-reported cognitive load on scaffolded activities (n=7). Cognitive load is measured on a 9-point scale (Very very low mental effort (1) - Very very high mental effort (9) [45]).**

| Activity | Succeeded without assistance | Mean cognitive load |
|---|---|---|
| Writing 1 (order plan goals) | 28.6% | 4.33 |
| Writing 2 (order plan code) | 85.7% | 4.78 |
| Writing 3 (fill plan slots) | 57.1% | 6.67 |
| Debugging | 85.7% | 4.38 |
| Explanation | 42.9% | 6.78 |

*6.3.1 Success in tasks.* Results show that participants were able to apply the plan knowledge they learned in the instructional content to complete various scaffolded programming activities (see Table 2). Participants progressed from no prior experience with the BeautifulSoup library to completing scaffolded coding activities after about half an hour of instruction. This timeline is greatly accelerated over the typical pattern of instruction and coding activities for BeautifulSoup content in the data programming course, where instruction spans approximately five hours of lecture and discussion section time, and assigned coding activities are expected to require at least 30 minutes each.

There did appear to be some start-up difficulty, since learners had the least amount of success without assistance on the first activity they faced: ordering plan goals. This may have occurred because the instructional content didn't include explicit practice about how to combine plans. All participants who didn't complete this activity on the first try were successful on the second try. This is notable because the plan ordering problems included distractors, which have been shown to increase difficulty on mixed-up code problems [28]. "Flailing" is a behavior some learners exhibit when solving "mixed-up-code" problems [18, 30]. "Flailers" re-order blocks randomly until they reach a correct ordering, completing many attempts. We observed only one instance of flailing, by P2 on the second code writing activity.

*6.3.2 Participants used purpose-first scaffolds to complete tasks.* We observed multiple ways that participants used purpose-first scaffolds to complete these activities.

*Looking at plans for help.* When facing difficulty during the completion of plan slots during code writing, participants frequently returned to instruction about plans for help. This approach allowed participants to fix bugs in their code after a careful inspection of prior plan examples.

Participants demonstrated that they could identify and make use of relevant plan information for their problem-solving. Participants typically referenced the plan page most relevant to their current error or code authoring activity. Seventy-eight percent of all visits

to a plan reference page were for a plan were the participant's code contained an error or was incomplete at the time of the page visit.

P6 was able to apply prior plan examples and instructional content to her current problem-solving and successfully complete the code writing problem.

> Okay. I'm going to look at this one. **[P6 references a plan page.]** Yes. This was with the different professors, but it was just like this. (...) Okay. So I guess you do need to keep that in. And then get out and then tag.get('href') or you can do href for something else. I think text maybe. Yes. Okay. (...) But in this example it looks like they did the same sort of thing. And this is finding, yeah, through multiple URLs. So it should be the same class equals. I'm pretty sure this is a class. Maybe it's a that's wrong. If it has to be div, I'm not really sure the difference between a and div. Okay. Is that right?

She visited a prior example that was most relevant to her current task, found similarities, and applied knowledge from the example to complete her code correctly.

*Using goal and subgoal language.* Participants often used subgoal language during their thinkaloud problem-solving, rather than describing code syntax. P2 completed the explain code problem successfully without assistance. Her thinkaloud is below, with subgoal labels bolded.

> Okay. Um... **Load libraries for web scraping**. Oh this is just like... I don't know... the soup thing is like a library that exists and we're just importing that so you can use it to pull stuff from webpages. And then **get a soup from a URL**... So they're listing the URL that they want to pull information from, I don't know what these lines do but they're always there. I'm going to **get all tags of a certain type from the soup**, so this is the class name and this is the tag. Or I guess this whole thing is the tag. Yeah. And then looks like they're making, they're naming this empty list and then they're putting... It looks like they're putting... because this is supposed to get links. Maybe they're getting the specific tag from each link and then putting that into this collect info list. **Get a soup from multiple URLs**. URLs ending... base URL plus ending. I don't know why they would do this, why you... A base URL... Maybe they're trying to get... the links, more info links, from each news page, news story. Or maybe they want the news story... no. **All tags of a certain type**. So yeah. I think they're trying to pull the text that you get when you click each of these more info links and combine them into a list and then print that list. Yeah. If that made sense.

P2 was able to trace through the subgoal labels, using some relevant information from the code, to describe a complex program consisting of five plans combined in a way she hadn't seen before.

## 6.4 Conversational programmers are motivated by purpose-first programming

After completing the curriculum, all participants expressed motivation to continue learning with purpose-first programming approaches. *"I could see myself really liking this curriculum for other topics,"* said P1. P5 said, *"If I could have one of these to do every week, and then in two years I would be so much more well-versed in Python, I would absolutely do it." "If you had any other thing besides web scraping I'd gladly sign up,"* said P7.

How exactly does purpose-first programming motivate conversational programmers? According to expectancy-value theory [17], motivation for a task is influenced by a person's *expectation of success* on the task and their *subjective value* for the task. In the post-curriculum interview, we explored learners' feelings of success and value for purpose-first programming. We analyzed participants' responses using thematic analysis and report major themes below.

*6.4.1 Participants felt success and enjoyment, which came from understanding and completing problems.* Participants largely felt successful on the activities they completed in the curriculum, even though they had moments of challenge and sometimes used resources and hints. *"Even though I had to look back at previous examples I still figured it out in the end, so I feel good,"* said P7. *"I did get frustrated, but I feel like I ended feeling pretty confident, like oh, I think I could do this, I think I could learn this,"* said P3. While most learners expressed a sentiment that they learned a significant amount in the curriculum, P2 felt that solving problems by referencing examples might be *"copying"* more than learning.

Feeling that they understood and accurately completed activities was associated with participants' success. *"I wasn't really sure what to expect, but it walked me through so clearly that I was able to complete the tasks very easily and feel like I actually did it successfully,"* shared P6. Some students expressed surprise at their level of success in the curriculum. *"I honestly feel like I did better than I expected to do,"* said P5.

*6.4.2 Learners perceived the purpose-first programming curriculum as having low cognitive load.* Participants frequently described purpose-first programming as *"separate steps"* (P7) that *"breaks it down really easily"* (P1) so they could think about *"one at a time instead of looking at it all"* (P5). Participants explained that the approach clearly demonstrated how to apply knowledge. *"Plans make it easier to choose what method you should be using for each type of situation,"* P2 said. Plans also allowed learners to focus on less while problem-solving, with choices constrained to a small number. P6 said, *"The concept of web scraping, at first is kind of intimidating, because it's like, how do you get all this information? But the way that the plans are written out, do you have one URL or do you have 2? Are you trying to get links or are you trying to get text? I feel like the way it was explained was very clear."*

This approach was different than most other programming experiences participants had been through. P1 said, *"previously it was never building up to writing the code on my own. It was either really, really easy questions, or really, really hard questions, and there was no transition that really made it stick in my brain."* P4 said that in her previous programming course she struggled with *"having a very*

*general destination and not knowing how to get there. So I thought that [purpose-first programming] was really helpful."*

At the same time, the introduction to new domain knowledge was a challenge that increased cognitive load. Participants had no prior formal training in HTML, and while some had done self-study, most had little HTML knowledge. P4 found so much new jargon overwhelming, saying, *"I feel like by the end of it, I had a better grasp on it. But in the beginning I definitely felt like I had no idea what...it all went over my head."*

*6.4.3 Conversational programmers expect purpose-first programming to work best for people like themselves.* When asked to describe which types of people would learn best with purpose-first programming, participants consistently named themselves, as beginners or people who struggled with code. P3 said, *"I think for someone like me who wants a lot of help, who needs a lot of help to do well, and wants a lot of examples and visuals and documentation, the plans worked really well." "I think plans are pretty useful for beginners like me,"* said P7. *"I have to fail at the code 10 times before I realize what I'm doing, so I guess I sort of like this,"* said P5.

On the other hand, participants felt that purpose-first programming may be constraining for people with significant prior knowledge or who were already succeeding at programming. P3 suggested, *"if everybody had to go through the plans it would feel frustrating or maybe even condescending to someone who was an advanced programmer." "I think it's just more of a beginner concept,"* said P6.

*6.4.4 Some conversational programmers would rather learn with purpose-first programming than code tracing.* Some participants had prior experience with PythonTutor [24], a program visualization tool [60] designed to help novice programmers understand the execution flow of programs. When comparing use of PythonTutor to their experience with purpose-first programming, these learners preferred purpose-first programming. *"I mean I guess in theory [PythonTutor] is useful but sometimes it gets so confusing,"* said P2. *"Having the plans is helpful because it's like if I know what to do then I can do it, but if I don't know what to do then it's like where do I even start?"* P3 said, *"[PythonTutor] is a visual thing which I thought would be helpful but honestly I hated it and never used it. And then just from my experience [PythonTutor] was helpful for people who knew programming really well, so it kind of feels like the plans might be for more beginners."*

On the other hand, P7 found the debugger in Visual Studio Code to be a valuable tool for her deeper learning. *"I think plans helped me in the very beginning, but once I understand the concept, debugging is what really helps me further my learning past the plans into more real life scenarios."*

*6.4.5 Purpose-first programming is valuable because it helps with basic and conceptual understanding.* These conversational programmers desired general knowledge about programming concepts, and felt that purpose-first programming helped them achieve that goal. P2 said, *"maybe I can't explain it perfectly and explain each part in depth but if I have like a general idea of what it's doing I think that's helpful."* P4 concurred, saying, *"having a background in everything as broad as you can is just the most helpful."* P6 said, *"I liked this because you can focus on it more conceptually without worrying about whether every part of your code is right."*

*6.4.6 Purpose-first programming is valuable because it is enjoyable.* Despite some general dislike of programming, most participants said that they enjoyed completing the curriculum. *"I think it was really interesting and fun,"* said P1. P6 said, *"I enjoyed it, considering I'm not a computer science major or anything and I don't really like this type of stuff."* Success often influenced enjoyment. *"When I was frustrated I didn't [enjoy it]. But when I was getting things right like in the debugging problem I was like yeah, I really like this,"* said P3. P2 was the only participant who admitted that she didn't really enjoy completing the curriculum, but success on problems was a bright spot. She said, *"I don't really like coding, I don't really like it but then when I got them right it was nice."*

*6.4.7 Purpose-first programming is valuable because it is realistic.* While these learners appreciated all the supports purpose-first programming provides, they appreciated knowing that what they were doing was "real". *"I like that you have the compiler so you could run code in the page,"* said P5. *"I like that these examples use real websites and [Information department] websites. It was super interesting and you actually got to see that they worked and how they worked,"* said P6. *"I liked using the Rate My Professor websites because they were like real life,"* said P7.

P6 felt her prior experience with block-based programming in high school didn't prepare her well for later text-based programming. However, with purpose-first programming, dragging and dropping blocks of code *was* helpful because it directly prepared her for realistic code authoring later. She said, *"We're not really dependent on the blocks. This is more of just a learning tool, but it's not how we do the code. We still type it."*

## 7 DISCUSSION

We provide preliminary evidence that for conversational programmers, purpose-first programming instruction is less cognitively difficult than typical instructional methods, leads to a feeling of success, and is aligned with goals and needs, leading to value for the activity. Compared to other experiences, like program visualization tools, block-based programming, and traditional programming assignments, conversational programmers preferred learning with purpose-first programming methods.

### 7.1 Understanding how purpose-first programming motivates conversational programmers

Our results can be explained through the lens of expectancy-value theory [17]. Expectancy-value theory states two factors most directly influence the choice to select and complete a task: (1) a person's expectation for success and (2) a person's subjective value for the task. Expectation for success involves a person's task-specific ability and self-concept, while subjective task value involves four factors: (a) attainment value, when the task aligns with the person's self-image; (b) interest-enjoyment value, when the person expects to enjoy the task; (c) utility value, when completing the task helps the person achieve a goal; and (d) relative cost, where value is decreased due to the loss of other potential opportunities. In Eccles' model, a person's identity directly influences both expectancy of success and subjective task value. Identity includes

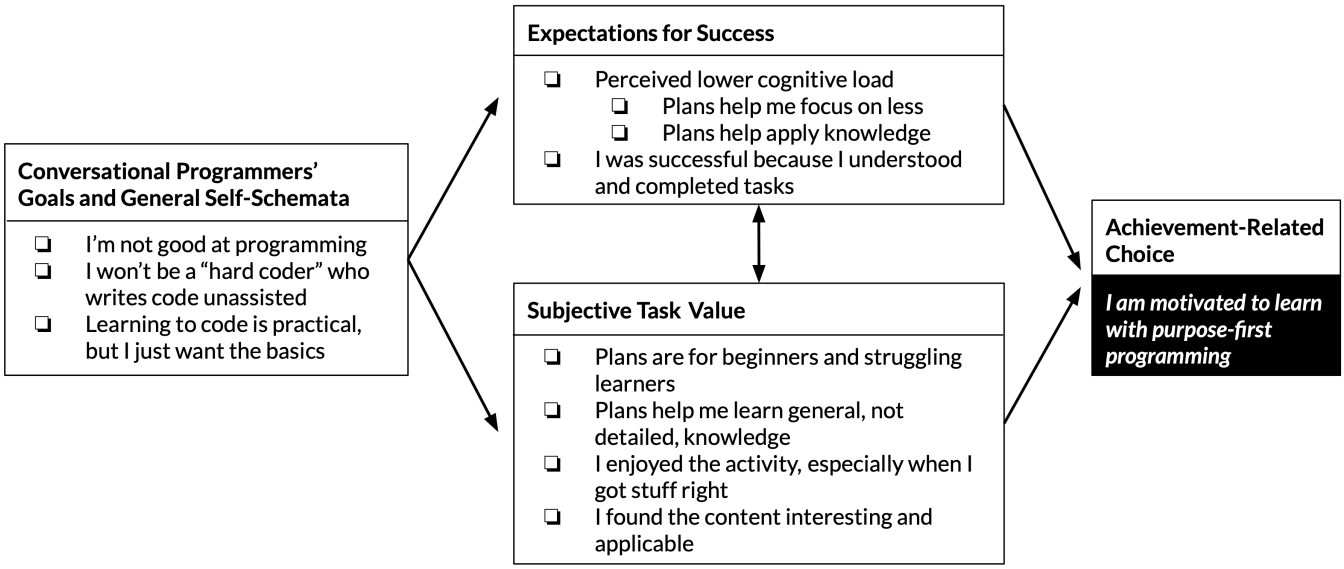



**Figure 8: Expectancy-value theory explains why conversational programmers are motivated by purpose-first programming**

personal and social self-schemata, short and long-term goals, ideal self, and self-concept of general ability.

Multiple factors influenced subjective task value for our participants (see Figure 8). They felt that the focus on basics and concepts was appropriate for their future career goals, and they also felt that this method of learning was a strong fit for their identity as beginner programmers who did not have strong programming skills. In addition, participants found the curriculum realistic because of its use of runnable code and real website, and also largely found it enjoyable, at least as far as learning to code could be enjoyable.

Expectation of success was influenced by the ease of the completion of activities and frequent feedback. While participants did struggle on some activities, they typically felt successful in the end, even when they needed to use help. Participants appreciated how learning with purpose-first programming emphasized how to achieve goals in a series of smaller steps.

## 7.2 Implications for future curriculum design

We provide preliminary evidence for the effectiveness of purpose-first programming learning, but there are many aspects of the design of purpose-first programming learning environment that remain to be studied. Future work should address both the cognitive and motivational factors that can derail programming learning.

Our design was constrained by the abilities of the Runestone ebook platform, but new tools designed specifically for purpose-first programming could provide more efficient support. For example, in our proof-of-concept, code highlighted with plan information was separate from runnable code, which had traditional syntax highlighting. Future purpose-first programming systems could highlight plans, slots, and subgoals *within* an editor, and offer the ability to turn plan highlighting on or off to match the preferences of different learners. In the proof-of-concept, our code writing process was staged across activities, so assembling and tailoring plans took place in separate steps. A dedicated purpose-first programming editor could combine plan assembly and tailoring, offering the ability to drag plan blocks, edit slots, and run code within a single editor. The accessibility of plan information is another area for further design. In the proof-of-concept, learners had to visit other pages to view plan information and examples. In a future system, this information could be made available in the editor itself with approaches like pop-up windows or dynamic sidebars.

Developing domain-specific code plans that communicate authenticity and achieve realistic goals but are also simple enough to teach in a brief learning experience is also a future design challenge. It's unclear how tolerant conversational programmers will be of "imagineering" [27] in their programming learning. Future work should evaluate the trade-off between complexity and realism for this population of learners.

# 8 CONCLUSION

In this paper, we conceptualized, developed, and evaluated *purpose-first programming*, an approach to programming learning for novice programmers who care more about what code can achieve than how a programming language works. After completing a proof-of-concept purpose-first programming curriculum with novel technical supports, novice conversational programmers were motivated to learn future programming topics with purpose-first methods. This motivation stemmed from a feeling of success, because learners could understand and complete problems, and alignment with goals and self-identity, because learners found the content useful and found the level of support appropriate for their needs. These learners were able to utilize purpose-first supports to complete scaffolded programming activities in a new topic area after only a short period of instructional time.

Our findings provide initial support for the effectiveness of purpose-first programming as a motivating starting point for conversational programmers and other novice programmers during programming learning. This work connects cognitive theories to theories of motivation in order to present a new approach to programming learning, designed specifically for novices who care more about the opportunities to use code than the operation of programming languages. As aspiring conversational programmers study programming in greater numbers, the need for a different instructional approach is becoming more apparent. Purpose-first programming can provide a new pathway to programming learning, designed with both the cognition and motivation of these learners in mind. This work opens new avenues to computing, inviting a broader group of learners to engage with programming, particularly those who are less likely to find code semantics motivating.

## ACKNOWLEDGMENTS

This work was supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGC-1148903.

## REFERENCES

[1] Anthony Anderson, Christina L. Knussen, and Michael R. Kibby. 1993. Teaching teachers to use HyperCard: a minimal manual approach. *British Journal of Educational Technology* 24, 2 (1993), 92–101.

[2] John B. Black, John M. Carroll, and Stuart M. McGuigan. 1986. What Kind of Minimal Instruction Manual is the Most Effective. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface* (Toronto, Ontario, Canada) *(CHI '87)*. Association for Computing Machinery, New York, NY, USA, 159–162. https://doi.org/10.1145/29933.275623

[3] Kirsten Boehner, Janet Vertesi, Phoebe Sengers, and Paul Dourish. 2007. How HCI Interprets the Probes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, CA, USA) *(CHI '07)*. ACM, New York, NY, USA, 1077–1086. https://doi.org/10.1145/1240624.1240789

[4] Karen Brennan and Mitchel Resnick. 2012. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association* (Vancouver, Canada), Vol. 1. AERA, Washington D.C., USA, 25.

[5] Tracy Camp, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. Generation CS: The Growth of Computer Science. *ACM Inroads* 8, 2 (May 2017), 44–50. https://doi.org/10.1145/3084362

[6] John M. Carroll, Penny L. Smith-Kerker, James R. Ford, and Sandra A. Mazur-Rimetz. 1987. The Minimal Manual. *Hum.-Comput. Interact.* 3, 2 (June 1987), 123–153. https://doi.org/10.1207/s15327051hci0302_2

[7] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. ACM, New York, NY, USA, 963–975. https://doi.org/10.1145/3242587.3242661

[8] Parmit K. Chilana, Celena Alcock, Shruti Dembla, Anson Ho, Ada Hurst, Brett Armstrong, and Philip J. Guo. 2015. Perceptions of Non-CS Majors in Intro Programming: The Rise of the Conversational Programmer. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, New York, NY, USA, 251–259. https://doi.org/10.1109/VLHCC.2015.7357224

[9] Parmit K. Chilana, Rishabh Singh, and Philip J. Guo. 2016. Understanding Conversational Programmers: A Perspective from the Software Industry. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, CA, USA) *(CHI '16)*. ACM, New York, NY, USA, 1462–1472. https://doi.org/10.1145/2858036.2858323

[10] Alan Cooper. 2004. *The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity* (2 ed.). Sams, Indianapolis, IN, USA.

[11] Kathryn Cunningham, Rahul Agrawal Bejarano, Mark Guzdial, and Barbara Ericson. 2020. "I'm Not a Computer": How Identity Informs Value and Expectancy During a Programming Activity. In *ICLS 2020 Proceedings (The 14th International Conference of the Learning Sciences, Vol. 2)*. International Society of the Learning Sciences (ISLS), 705–708.

[12] Kathryn Cunningham, Shannon Ke, Mark Guzdial, and Barbara Ericson. 2019. Novice Rationales for Sketching and Tracing, and How They Try to Avoid It. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. ACM, New York, NY, USA, 37–43. https://doi.org/10.1145/3304221.3319788

[13] Andrew Dillon. 2012. What It Means to Be an iSchool. *Journal of Education for Library and Information Science* 53, 4 (2012), 267–273.

[14] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, New York, NY, USA, 422–431.

[15] Jacquelynne Eccles. 1983. Expectancies, Values and Academic Behaviors. In *Achievement and Achievement Motives: Psychological and Sociological Approaches*. Freeman, San Francisco, CA, USA, 75–146.

[16] Jacquelynne S. Eccles. 2005. Subjective Task Value and the Eccles et al. Model of Achievement-Related Choices. In *Handbook of competence and motivation*. Guilford Publications, New York, NY, USA, 105–121.

[17] Jacqueline S. Eccles. 2009. Who am I and what am I going to do with my life? Personal and collective identities as motivators of action. *Educational Psychologist* 44, 2 (2009), 78.

[18] Barbara J. Ericson, Mark J. Guzdial, and Briana B. Morrison. 2015. Analysis of Interactive Features Designed to Enhance Learning in an Ebook. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/2787622.2787731

[19] Barbara J. Ericson, Kantwon Rogers, Miranda Parker, Briana Morrison, and Mark Guzdial. 2016. Identifying Design Principles for CS Teacher Ebooks Through Design-Based Research. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. ACM, New York, NY, USA, 191–200. https://doi.org/10.1145/2960310.2960335

[20] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery, New York, NY, USA, 35–42. https://doi.org/10.1145/2632320.2632346

[21] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. ACM, New York, NY, USA, 211–216. https://doi.org/10.1145/2839509.2844556

[22] Fernand Gobet, Peter CR Lane, Steve Croker, Peter CH Cheng, Gary Jones, Iain Oliver, and Julian M Pine. 2001. Chunking Mechanisms in Human Learning. *Trends in cognitive sciences* 5, 6 (2001), 236–243.

[23] Ira Greenberg. 2007. *Processing: Creative Coding and Computational Art*. Apress, Berkeley, CA, USA.

[24] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. ACM, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[25] Mark Guzdial. 1995. Software-realized scaffolding to facilitate programming for science learning. *Interactive Learning Environments* 4, 1 (1995), 1–44.

[26] Mark Guzdial, Michael Konneman, Christopher Walton, Luke Hohmann, and Elliot Soloway. 1998. Layering scaffolding and CAD on an integrated workbench: An effective design approach for project-based learning support. *Interactive Learning Environments* 6, 1/2 (1998), 143–179.

[27] Mark Guzdial and Allison Elliott Tew. 2006. Imagineering Inauthentic Legitimate Peripheral Participation: An Instructional Design Approach for Motivating Computing Education. In *Proceedings of the Second International Workshop on Computing Education Research* (Canterbury, United Kingdom) *(ICER '06)*. ACM, New York, NY, USA, 51–58. https://doi.org/10.1145/1151588.1151597

[28] Kyle James Harms, Jason Chen, and Caitlin L. Kelleher. 2016. Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/2960310.2960314

[29] Brian Harvey and Jens Mönig. 2010. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists. In *Proceedings of the 2010 Constructionism Conference* (Paris, France). 1–10.

[30] Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. 2012. How Do Students Solve Parsons Programming Problems? An Analysis of Interaction Traces. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (Auckland, New Zealand) *(ICER '12)*. ACM, New York, NY, USA, 119–126. https://doi.org/10.1145/2361276.2361300

[31] Cindy E. Hmelo and Mark Guzdial. 1996. Of Black and Glass Boxes: Scaffolding for Doing and Learning. In *Proceedings of the 1996 International Conference on Learning Sciences* (Evanston, Illinois) *(ICLS '96)*. International Society of the Learning Sciences (ISLS), 128–134.

[32] Luke Hohmann, Mark Guzdial, and Elliot Soloway. 1992. SODA: A Computer Aided Design Environment for the Doing and Learning of Software Design. In *Proceedings of the 4th International Conference on Computer Assisted Learning (ICCAL '92)*. Springer-Verlag, Berlin, Heidelberg, 307–319.

[33] David Landy and Robert L. Goldstone. 2007. How Abstract Is Symbolic Thought? *Journal of Experimental Psychology: Learning, Memory, and Cognition* 33, 4 (2007), 720.

[34] Jean Lave and Etienne Wenger. 1991. *Situated Learning: Legitimate Peripheral Participation.* Cambridge University Press, Cambridge, UK.

[35] Raymond Lister. 2011. Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114* (Perth, Australia) *(ACE '11)*. Australian Computer Society, Inc., AUS, 9–18.

[36] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 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* (Leeds, United Kingdom) *(ITiCSE-WGR '04)*. ACM, New York, NY, USA, 119–150. https://doi.org/10.1145/1044550.1041673

[37] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 16 (Nov. 2010), 15 pages. https://doi.org/10.1145/1868358.1868363

[38] John H. Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. 2008. Programming by choice: urban youth learning programming with scratch. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education* (Portland, OR, USA). ACM, New York, NY, USA, 367–371. https://doi.org/10.1145/1352135.1352260

[39] Lauren Margulieux and Richard Catrambone. 2017. Using Learners' Self-Explanations of Subgoals to Guide Initial Problem Solving in App Inventor. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 21–29. https://doi.org/10.1145/3105726.3106168

[40] Lauren E. Margulieux, Richard Catrambone, and Mark Guzdial. 2013. Subgoal labeled worked examples improve K-12 teacher performance in computer programming training. In *Cooperative Minds: Social Interaction and Group Dynamics Proceedings of the 35th Annual Conference of the Cognitive Science Society* (Berlin, Germany), M. Knauff, M. Pauen, N. Sebanz, and I. Wachsmuth (Eds.). Cognitive Science Society, Austin, TX, USA, 978–983.

[41] Lauren E. Margulieux, Mark Guzdial, and Richard Catrambone. 2012. Subgoal-labeled Instructional Material Improves Performance and Transfer in Learning to Develop Mobile Applications. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (Auckland, New Zealand) *(ICER '12)*. ACM, New York, NY, USA, 71–78. https://doi.org/10.1145/2361276.2361291

[42] Marvin Minsky. 1974. *A Framework for Representing Knowledge.* Technical Report. Massachusetts Institute of Technology-AI Laboratory.

[43] Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial. 2016. Subgoals Help Students Solve Parsons Problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. ACM, New York, NY, USA, 42–47. https://doi.org/10.1145/2839509.2844617

[44] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. 2015. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. ACM, New York, NY, USA, 21–29. https://doi.org/10.1145/2787622.2787733

[45] Fred G. Paas. 1992. Training Strategies for Attaining Transfer of Problem-Solving Skill in Statistics: A Cognitive-Load Approach. *Journal of Educational Psychology* 84, 4 (1992), 429.

[46] Dale Parsons and Patricia Haden. 2006. Parsons Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 157–163. http://dl.acm.org/citation.cfm?id=1151869.1151890

[47] Alan J. Perlis. 1982. Special Feature: Epigrams on Programming. *ACM SIGPLAN Notices* 17, 9 (1982), 7–13.

[48] Leonard Richardson. 2020. *Beautiful Soup Documentation.* Beautiful Soup 4.9.0. Accessed: 2020-09-15.

[49] Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* 13, 3 (1989), 389–414.

[50] Kelly Rivers, Erik Harpstead, and Ken Koedinger. 2016. Learning Curve Analysis for Programming: Which Concepts Do Students Struggle With?. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. ACM, New York, NY, USA, 143–151. https://doi.org/10.1145/2960310.2960333

[51] Roger C. Schank and Robert P. Abelson. 1977. *Scripts, Plans, Goals, and Understanding: An Inquiry into Human Knowledge Structures.* Lawrence Erlbaum, Hillsdale, New Jersey, USA.

[52] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do We Know How Difficult the Rainfall Problem Is?. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '15)*. Association for Computing Machinery, New York, NY, USA, 87–96. https://doi.org/10.1145/2828959.2828963

[53] Elliot Soloway. 1985. From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research* 1, 2 (1985), 157–172.

[54] Elliot Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (1986), 850–858.

[55] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM* 26, 11 (1983), 853–860.

[56] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (Sept. 1984), 595–609.

[57] Elliot Soloway, Kate Ehrlich, Jeffrey Bonar, and J. Greenspan. 1982. What do novices know about programming? In *Directions in Human-Computer Interaction*, Andre Badre and Ben Schneiderman (Eds.). Ablex Publishing, Norwood, New Jersey, USA, 87–122.

[58] Elliot M. Soloway and Beverly Woolf. 1980. Problems, Plans, and Programs. In *Proceedings of the Eleventh SIGCSE Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '80)*. ACM, New York, NY, USA, 16–24. https://doi.org/10.1145/800140.804605

[59] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Transactions on Computing Education* 13, 2, Article 8 (July 2013), 31 pages. https://doi.org/10.1145/2483710.2483713

[60] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education* 13, 4 (2013), 15.1– 15.64. https://doi.org/10.1145/2490822

[61] James Clinton Spohrer. 1989. *MARCEL: A Generate-Test-and-Debug (GTD) Impasse/Repair Model of Student Programmers.* Ablex Publishing, Norwood, New Jersey, USA.

[62] James C. Spohrer and Elliot Soloway. 1985. Putting it All Together is Hard for Novice Programmers. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. Vol. March. IEEE, New York, New York, USA.

[63] James C. Spohrer and Elliot Soloway. 1986. Analyzing the high frequency bugs in novice programs. In *Empirical Studies of Programmers Workshop*, Elliot Soloway and S. Iyengar (Eds.). Ablex Publishing, Norwood, New Jersey, USA, 230–251.

[64] James C. Spohrer, Elliot Soloway, and Edgar Pope. 1985. A Goal/Plan Analysis of Buggy Pascal Programs. *Human–Computer Interaction* 1, 2 (June 1985), 163–207. https://doi.org/10.1207/s15327051hci0102_4

[65] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive Science* 12 (1988), 257–285.

[66] Anne Venables, Grace Tan, and Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (Berkeley, CA, USA) *(ICER '09)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1584322.1584336

[67] Lev S. Vygotsky. 1978. *Socio-cultural theory.* Harvard University Press, Cambridge, MA.

[68] April Y. Wang, Ryan Mitts, Philip J. Guo, and Parmit K. Chilana. 2018. Mismatch of Expectations: How Modern Learning Resources Fail Conversational Programmers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3173574.3174085

[69] David Weintrop and Uri Wilensky. 2015. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-Based Programming. In *Proceedings of the 14th International Conference on Interaction Design and Children* (Boston, Massachusetts) *(IDC '15)*. ACM, New York, NY, USA, 199–208. https://doi.org/10.1145/2771839.2771860

[70] David Wood, Jerome S Bruner, and Gail Ross. 1976. The Role of Tutoring in Problem Solving. *Journal of child psychology and psychiatry* 17, 2 (1976), 89–100.

[71] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J Ko. 2019. A Theory of Instruction for Introductory Programming Skills. *Computer Science Education* 29, 2-3 (2019), 205–253.