

**Collaborations between two communities have unearthed a sweet spot for future programming efforts.**

BY SARAH E. CHASINS, ELENA L. GLASSMAN, AND JOSHUA SUNSHINE

## PL and HCI: Better Together

IN THE LAST 10 years, the computer science (CS) community has developed novel programming systems that are transforming our world. Data journalists are wielding new programming tools to enrich many major media outlets with interactive visualizations. Microsoft Excel, the primary data programming environment for hundreds of millions of people, now comes with a program synthesis tool that helps users clean and transform their data, sparing them from writing painful spreadsheet formulas. These projects share an important common factor: they succeed because they make programming easier. They demonstrate the power of combining human-computer interaction (HCI) and programming languages (PL). We organized the PLATEAU workshop, part of a growing community that tackles work at this intersection. Here are the research problems that led us to this hybrid field:

**PL → HCI** Josh Sunshine began his career as a PL researcher working on the design of the Plaid language. He was drawn to HCI techniques when he tried to run a user study of Plaid. He found that users failed to complete even simple tasks—the language was just too difficult. His language design work since then has relied heavily on formative HCI methods like contextual inquiry and natural programming elicitation.<sup>24</sup> The end result is usable languages and successful users.

**HCI → PL** Elena Glassman was working toward her Ph.D. in HCI when she developed a tool for visualizing student code to help teachers see where student solutions overlap and where they differ. For each new programming assignment, she had to build a new analyzer, which was tedious, time-consuming, and required her expertise as the tool's designer. Later, a colleague introduced her to program synthesis (PL). She realized that she could equip her tool with an example-based synthesizer so that teachers could author custom analyzers for their own assignments.

**HCI ↔ PL** In talking to social scientists about their technical challenges, Sarah Chasins learned that Web scraping was a big obstacle to obtaining data for their research. She began iteratively developing a Programming-By-Demonstration (PBD) Web automation tool with its own custom language to meet the social scientists' needs. Over the course of the work, each individual subproblem demanded both PL and HCI. For example, combined PL and HCI approaches put parallel scraping in reach. To make her new parallelization construct usable, Sarah phrased the problem in terms of a familiar task (HCI); to implement it, she compiled to parallel programming primitives (PL).

The rising tide of PL+HCI research arrives as we observe a few key trends. First, advances in language engineering support make it easier for anyone to develop new languages. Second, methodological and theoretical innovations in HCI make it easier than ever to



study humans doing rich and complex computing tasks like programming, which lets us apply HCI techniques to language development. Third, broad and diverse new audiences are seeking automation.

► On the basis of these trends and our own knowledge of the field, we have identified a few key directions, summarized in the accompanying figure, that HCI and PL experts should explore to take full advantage of the combined power of HCI and PL:

► HCI practitioners can benefit from new tools that make it easy to build domain-specific and general-purpose programming languages. However, users need help writing safe and correct programs, and PL techniques can help. Finally, users may not always want to write code directly; to balance ease-of-use with the power and flexibility of

programming, our interfaces should give users multiple ways to express their intent.

► PL practitioners can use need-finding techniques to identify high-impact problem domains or programmers' current and future pain points. They can make better design decisions via cognitive and behavioral theory where those theories are available. Where theory is not available, they can make better design decisions by leveraging iterative design cycles

that incorporate user feedback.

The remainder of this article describes misconceptions that have inhibited work at the intersection of these two subfields, how each subfield can benefit from the other, and the kinds of dramatic research successes that result from successful PL+HCI unions. Finally, we discuss the directions for future work and how they will deliver important new languages and interfaces. Our key takeaways are summarized in Table 1.

**Table 1. Key directions for HCI experts looking to integrate PL practices and PL experts looking to integrate HCI practices.**

| To interface designers:            | To language designers:                               |
|------------------------------------|--|
| Give users PLs                     | Pick good problems,                                  |
| But help them use PLs responsibly, | Develop theories of human capabilities and behavior, |
| And don't expect code alone.       | And get frequent user feedback when you lack theory. |



### What Are We Talking About Here?

HCI is concerned with creating new ways of interacting with computers, using computers to enhance human-to-human interaction, and studying how existing systems affect individuals and society. PL is concerned with the theory, design, and implementation of programming languages, program analyses, and program transformations. This article is devoted to work that combines PL and HCI techniques to advance the goals of either field. However, many other fields and subfields consider how we can use programming languages to serve humans. We provide pointers to a few of the most relevant fields here:

**Software engineering.** PL, HCI, and software engineering (SE) have a key overlapping interest: getting computers to do what we want. Modern PL-HCI research often ends up at SE venues as the closest fits in today's conference landscape, but much of the work at the PL-HCI border does not fit naturally within SE's scope of interest. In particular, SE primarily focuses on professional software engineers, and many PL-HCI works are aimed at other audiences.

**Psychology of programming.** The psychology of programming (PoP) community has a long history of studying everything from the cognitive work of individual programmers to how they deal with large codebases to how they work in engineering teams. (See Blackwell et al.<sup>4</sup> for an overview of how the field evolved from the late 1960s into the present.) This critically important work has unfortunately had limited impact on mainstream PL.<sup>14,29</sup> This article advocates for more work that crosses the boundaries between PL and HCI, but we hope readers will recognize that many of the same arguments apply for crossing the disciplinary boundaries between PL and PoP.

**Computer science education.** CS education (CSEd) research, because it focuses on interventions that make it easier for novices to learn CS, often involves forays into programming languages and tools. For example, consider the Alice<sup>9</sup> and Scratch<sup>32</sup> projects, which set off the modern interest in block-based editors and structure editors. This style of CSEd work often advances HCI goals, but like SE it empha-

sizes a particular audience—novice programmers who want to learn CS—and a particular set of goals.

**End-user programming.** This subfield has a long, rich history and, like SE and CSEd, an emphasis on a particular subset of users. In this case, the target audience excludes professional software developers and includes users in other domains who need computational support for their goals. The body of work in end-user programming (EUP) extends back to “*A Small Matter of Programming*,”<sup>26</sup> and it remains an active domain.<sup>18,21</sup>

Some of the work in the intersection of PL and HCI fits neatly into these related communities, and some of it does not. Certainly, the new collaborations between PL and HCI researchers are not the first efforts to tackle the goals laid out in this article—see for example Kay,<sup>17</sup> Myers et al.,<sup>24</sup> and Pane et al.,<sup>28</sup> in addition to the works cited earlier. However, this article highlights the work we can do when we bring HCI and PL techniques together at the same table that we cannot do in isolation. Substantive collaboration across these fields—and with SE, PoP, CSEd, and EUP!—offers a promising route toward usable languages and powerful interfaces. We are excited to see what these subfields can do together as they begin a fresh wave of cross-disciplinary collaborations.

### Common PL+HCI Misconceptions

We begin by addressing a few of the misconceptions that sometimes stand in the way of PL-curious HCI researchers and HCI-curious PL researchers.

**Misconception: PL doesn't care about people.** This common misconception reflects the idea that PL researchers only care about logic and proofs, or only about compiler performance—not about people. In fact, much of the field's work on language features and developer tools has been driven by an interest in the user experience. Although work that brings HCI techniques to bear on PL problems is still fairly young, the interest in making programming languages and tools more usable is longstanding. For instance, the entire program synthesis community sprang up around the idea that some programming tasks are easier for machines than for humans and

should be offloaded to specialized program generation tools.

**Misconception: PL just makes new general-purpose languages.** Another common misconception we hear about PL research is that it is all about creating new general-purpose programming languages. Since the most popular languages are at least 20 years old, they ask, has the programming languages community had any impact? Some even argue that *PL research is stagnant*. In fact, only a tiny fraction of papers at programming languages conferences (<1%) discuss new general-purpose language designs. Most research investigates novel implementation techniques, program analyses, verification and synthesis techniques, tools to support language engineers, and new language features. Popular languages are taking advantage of that work as they evolve. For example, the tremendous performance improvements in JavaScript engines were built on just-in-time compilation techniques developed by PL researchers.

**Misconception: PL can't benefit from human factors research.** Some researchers contend that HCI methods are not applicable to programming languages because they are complex learned artifacts. The benefit of new language constructs may only come after substantial education and experience, and they believe HCI methods are limited to tools for end users and novices. In fact, HCI methods have been used to study everything from nuclear power plant control systems to augmented reality and flight control systems. Another misconception we hear from PL practitioners is that HCI methods are only useful for surface concerns like fonts, colors, and layout. HCI is not, and has never been, restricted to purely surface-level or visual features. It can encompass everything from the user's mental models as they learn a new tool to the class of information passed between user and tool to the set of abstractions that lets them express their needs.

**Misconception: HCI is all about evaluation.** Another common one: HCI is just about evaluating interfaces via users studies. HCI has never had a narrow focus on purely evaluative work. Over the course of its 40-year history, the HCI community has de-

veloped methods for engaging users in the entire iterative design cycle. At the beginning of the design process, need-finding and formative studies offer low-cost ways to identify existing pain points and anticipate usability problems early. Throughout the design process, a vast space of HCI methods—for example, heuristic evaluation, cognitive walkthroughs, “Wizard of Oz” studies, rapid prototyping, think-aloud studies, natural program elicitation—can give developers more information to make better-informed design decisions.

**Misconception: HCI is just implementing what users say they want.** Another misconception, about formative studies in particular, is that using HCI during the design process means simply implementing what users say they want. This is the Steve Jobs, Henry Ford “If I had asked people what they wanted, they would have said faster horses” concern. Conducting formative studies does not have to mean asking users what they want and then delivering what they request. Some need-finding research involves listening to user requests; but a great deal is focused on observing users’ behavior in a given context, even testing hypotheses about their behavior. Via iterative design of prototypes, researchers can expose potential users to multiple hypothetical futures they would never have requested and solicit feedback. These strategies empower potential users to shape technologies that have never existed before, putting those technologies on track to be useful and usable.

**Misconception: Doing HCI is too hard.** This misconception usually revolves around either the idea that user studies need to include dozens of people to be valid or the idea that the IRB approval process is grueling. In fact, even studies with small numbers of participants can contribute important insights and evaluations. The key is to include enough participants to provide evidence of the claims we want to make. If we build a tool for rare domain experts, we may run a study with five people that focuses on qualitative insights. Or, if we expect our tool to have a large effect on outcomes, enrolling 10 participants in a within-subjects study may be enough to show meaningful differences between their

experiences using the experimental and control interfaces. If the class of potential participants is large, we may run a medium-sized lab study of 20–25 people or a large online study that focuses on quantitative insights. Finally, if we want a lightweight way to check our ideas during a design process, it can be enough to watch over a friend’s shoulder and hear them talk through using our prototype; this informal,  $n=1$  ‘study’ can be enough to reveal critical design flaws or spark new ideas!

IRB processes vary by institution, but most have official low-risk (‘exempt’) submission categories for which the approval process is lightweight and fast—and a vast majority of PL+HCI studies fall into these categories. Colleagues who do exempt human subjects research are a great resource for institution-specific advice about IRB processes.

### HCI and PL: A Two-Way Street

While PL and HCI have had relatively little cross-over in terms of collaborations and shared literature, each community has developed techniques that can help researchers in the other field. Here we describe a few concrete ways that HCI concepts and techniques can improve PL outcomes; PL concepts and techniques can improve HCI outcomes; and PL and HCI researchers can integrate their complementary expertise to advance goals that matter in both communities.

**PL → HCI: The power of PL-backed interfaces.** Languages are powerful interfaces for communicating with computers. Unlike typical menu- and button-based interfaces, languages are compositional: they provide a set of primitives and a means of combination, empowering users to create new primitives out of existing ones. If they are Turing-complete, they can describe any computable function. Even a non-Turing-complete language can express an infinite space of functions. While both GUIs and languages are often designed around making it easy for users to say common things, a language empowers users to say uncommon things too. Users can even interact with standard interface elements instead of code and still wield the power of a programming language, if the interface automatically generates programs

for the user (for example, via program synthesis). These PL-backed interfaces can help us realize the vision for powerful interfaces advanced by Shneiderman in his seminal “Direct Manipulation.”<sup>34</sup> In particular, the expressive power of programming languages can elicit the “desire to explore more powerful aspects of the system” that is often lacking from GUIs.

**Building PLs can be easy.** Language engineering has become easier with the development of new, easier language implementation support tools like language workbenches and parser generators. Designing task-relevant abstractions and instantiating them in a domain-specific language now takes only minimal training. For HCI work that benefits from the power and flexibility of a language, these new PL tools can support interface and system designers in making new languages, abstractions, and domain-specific languages.

**Using PLs can be easy.** PL advances like synthesis and modern retargeting let us ask users for a little work and get a lot in return. With techniques like programming by demonstration and programming by example, users can provide non-code specifications (for example, input-output pairs) and get a program in return. With retargeting approaches, we can take programs originally intended for one purpose and reuse them to create new artifacts.

**Using PLs correctly.** PL, like all other areas of computer science, brings technical capabilities to the table that can help address HCI concerns. For example, the PL community has developed verification techniques to the point that they can check more than simple, low-level properties; they can verify functional correctness, safety, security, accessibility, even adherence to social norms,<sup>30</sup> and other properties that matter to the HCI community. Many PL techniques, such as program analysis, bug fixing, and verification, can offload tasks to the machine, reducing the cognitive load of human operators and designers. These sophisticated techniques are already being applied in professional programming environments. As more end users begin automating tasks, we see opportunities to apply these same techniques to their computer and robot interactions.

**HCI → PL: Iterative, user-centered design.** User-centered design focuses us on assessing the usefulness and usability of our languages and tools throughout the design process—not just in a final evaluative step. Need-finding studies let designers identify key needs, stumbling blocks, and challenges before the design process even begins. When we tackle needs that have already been validated via need-finding studies, we have good reason to believe our languages or tools can solve real users' problems. Formative studies throughout the design process let us progress steadily toward usability during the language or tool building process. Soliciting feedback from users at multiple points in the design process means we are less likely to end up with user-antagonistic tools at the end, when we have already sunk years of time, energy, and engineering into them.

**Theories of human cognition and behavior, design heuristics.** Making every design decision based on direct user observation would be expensive, time-consuming, and impractical. Design heuristics, for example, Green et al.<sup>12</sup> describe elements of interactive systems that designers have found

over and over are critical to usability, like the visibility of the system's status or the ability to 'undo' an action. Theories of human cognition and behavior make predictions about what users will, will not, and cannot do in any system we construct for them. Like design heuristics, theory predictions are guidelines to narrow our design space and form expectations that may or may not be violated when the user and the system ultimately interact. Some programming tools are already designed on the basis of programming-specific behavioral theory, for example: understanding how programmers backtrack enabled researchers to develop selective undo in Integrated Developer Environments (IDEs).<sup>37</sup> Developing more and deeper theory can pay dividends for the entire programming languages community.

**Evaluation: Beyond user studies.** Many programming systems developers are interested in making claims about their advantages for users. HCI has developed many methods for evaluating these claims. These include traditional user studies in the lab but also low-cost heuristic methods, deeper long-term case studies,<sup>35</sup> and rigorous analysis of field data like user logs. To back up the strongest and most exciting claims, we may need multiple evaluation methods—for example, user logs to acquire large-scale data and a lab study to understand the otherwise contextless log statistics. Readers interested in learning more about the diverse set of human-factors evaluations we can apply to programming interactions are encouraged to read Myers et al.'s excellent essay, "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools."<sup>25</sup>

**HCI ↔ PL:** In a few domains, both HCI and PL currently advance the state of the art, although these advances are not always shared across the disciplinary divide. In these domains, we hope to engender a richer culture of cross pollination, in the belief that both communities can benefit from the findings of the other.

**Abstraction design.** Each subfield has its own culture and design goals. They both contribute to features that matter to users, but often to different sets of features. The PL community has deep expertise in developing modular,

reusable abstractions. The HCI community has deep expertise in developing abstractions that are easy to learn or match the existing mental models of their target users. With rich histories of abstraction design across both fields, a union of these forms of expertise holds the promise of delivering useful, usable, and powerful abstractions.

**Interactive and non-interactive environments.** Programming environments that demand a mix of interactive and non-interactive modes are common in the real world. For example, programmers draft code in a relatively non-interactive text window, then refactor the same code via an interaction with their editor of choice. HCI has developed rich theories of interactive computing environments, while PL has long studied how to shape languages to produce good experiences for non-interactive programming settings. For modern programming systems that demand both modes, it is the combination of both PL and HCI expertise that offers the guidance we need (also, see the sidebar "Can We Simply Stage HCI and PL Expertise?").

### What It Looks Like When It Goes Well

Bringing HCI and PL expertise together at the same table lets us meet challenges that neither field can accomplish alone. This section highlights how the union of these fields equips us to bring programming to new audiences, improve the programming experience for novices and experts alike, and fine-tune the division of labor between human and machine.

**PL+HCI brings the power of programming to new audiences.** Non-coders want programs. They want programs that collect, analyze, and visualize data; programs to control their own phones, computers, and other devices; programs to eliminate boring, repetitive tasks. But for now, there is still a gap between the programming skills of the average adult and the skills required to write the programs they want or need.

The union of PL and HCI techniques can close that gap by dramatically reducing the programming skills required to automate important tasks. Modern domain-specific languages put simple but useful programs in

## Can We Simply Stage HCI and PL Expertise?


Design choices interact. We cannot ask the PL expert to design the abstractions, deliver them to the HCI expert for a second pass, and expect the optimal design as a result. These choices interact. Design decisions that we make to improve learnability have implications for how we achieve modularity, and vice versa.

To achieve tools and languages that meet the goals of both subfields, we need HCI and PL expertise at the same table. It is not enough to know what users want unless we can make a language, synthesizer, or programming environment that delivers it. Likewise, making a new language, synthesizer, or environment will not advance our goals unless the new artifact meets real user needs.


reach in domains like building websites<sup>36</sup> or automating smart home actions (IFTTT). With HCI, we can learn the kinds of inputs users are willing and able to provide; with PL, we can invent techniques that turn those inputs into the programs users need. Already, modern program synthesis empowers non-coders to build new voice assistant skills via a conversation with their phone;<sup>20</sup> write feedback about one student program to propagate feedback to many students' programs;<sup>15</sup> scrape large datasets from the Web by demonstrating how to scrape one row;<sup>7</sup> transform and clean data by giving examples of a few transformed items or cells;<sup>13,19</sup> and visualize or model a dataset by providing just the dataset.<sup>6,23</sup>

**PL+HCI lets us use formal reasoning to create richer programming experiences.** Some work that starts as advances to programming language theory or implementation ultimately invents novel programming interaction techniques. The simplicity of Smalltalk's object model enabled the language designers to develop many novel programming conveniences that we now take for granted—for example, an integrated development environment, reflection, and unit testing frameworks.<sup>17</sup> Work that starts as an effort to make incomplete programs well-typed can ultimately let us build programming environments that work just as well for partial programs as complete programs.<sup>27</sup> Work that starts as an effort to create bidirectional mappings between program inputs and outputs can let us build programming environments in which users can program by editing code or by tweaking a diagram.<sup>16</sup> By deeply considering formal models of programming, we can ultimately produce richer interactive programming systems.

**PL+HCI produces better decisions about the division of labor between the human and the machine.** Computers are better at some tasks than humans, and vice versa, and this landscape shifts as computing advances and education evolves. In the classical model of programming, the programmer instructs the machine, and the machine follows the instructions. Modern programming tools can divide programming tasks between human and machine in new and creative



## The union of PL and HCI techniques can dramatically reduce the programming skills required to automate important tasks.



ways. For example, reasoning about whether a robot upholds human social norms (for example, maintaining eye contact) is usually left to the human programmer, but new human-robot interaction work offloads this task to a verifier,<sup>30</sup> effectively erecting guardrails that keep programmers from violating their own design goals. Synthesis tools let users offer input-output examples and other non-code specifications when those specifications are easier to provide than the code itself. Rather than requiring humans to hand-write low-level image processing pipelines, the Halide project<sup>31</sup> allows programmers to write in a high-level language, delegating the low-level scheduling details to the computer. This new generation of tools leverages a diverse array of techniques, everything from program synthesis and machine learning to domain-specific languages and program verification.

### Obstacles

There are two key obstacles to accomplishing our vision of united PL and HCI. First, most PL and HCI researchers lack knowledge of each other's tools and methods. The prerequisites for work in these fields are disjoint. Many, even most, researchers in PL or HCI enter one of these subfields without learning even the basics of the other.

Second, the demands for rigor in the PL and HCI communities do not always compose. We need knowledge of both communities to selectively apply the standards of each community as appropriate. Not all tasks should be neatly evaluated in a one-hour controlled user study. Not all claims require mathematical proofs. We are in danger of smothering exciting new research if we ask authors to check boxes that make sense for the single-subfield contributions we have seen before but not for the new contributions they are offering.

We believe that learning about both fields and their intersection is the best remedy to both these obstacles. We hope this article's glimpse into PL+HCI research inspires readers to learn more. Readers who want a preview of the kinds of contributions that will push this field forward—the kinds



**Table 2. What can PL practitioners borrow from HCI? This table summarizes three classes of PL contribution that we can produce by drawing on HCI techniques.**

| Contribution         | Key Message   | Elaboration  |
|----------------------|---|--|
| Need-Finding         | Pick good problems                                  | If we identify real needs before we begin designing, we have a better chance of contributing useful, high-impact programming languages and tools.  |
| Behavioral Theory    | Develop theories of human capabilities and behavior | Given that user evaluation is time-consuming and expensive, we can make better design decisions more quickly if our field builds up theories that predict user behavior.   |
| Iterative Refinement | And get frequent user feedback when you lack theory | PL innovation constantly uncovers new design questions. We can apply user-centered design—a feedback loop between builders and users, a cycle of evaluations and redesigns—to inform our decisions in addition to any applicable theory. |

of papers we should be accepting into our favorite venues—should read on to the next section for a taste of how we advance to the vision of productively integrated HCI and PL.

### Where Do We Go Next?

In this section, we present a vision of where this hybrid field should go now. We highlight a few types of contributions that harness PL and HCI's combined strengths. Readers who want to participate in the PL+HCI field can read on for a guide on how to contribute. We give specific recommendations for those with HCI backgrounds and those with PL backgrounds.

**PL practitioners: Consider the following contribution types.** We believe a few key contribution types have the potential to dramatically improve PL practice. By borrowing techniques from HCI, PL practitioners can produce higher-impact languages and tools, make their languages more usable by novices and experts, and make it easier for future language designers to produce usable languages.

*Need-finding studies.* Need-finding studies help us produce a rich understanding of the needs of a target population and ultimately identify the problems that real users need to solve. Contextual inquiry, interviews, surveys, analyses of log data, analyses of forums and StackOverflow, exploratory user studies—all of these can reveal important user needs. Need-finding has played an important role in shaping successful PL projects ranging from D3<sup>5</sup> and Vega<sup>33</sup> to FlashFill.<sup>13</sup> At their best, need-finding studies produce needs analyses that are useful not just for motivating a single project but for the research community as a whole.

The HCI and SE communities al-

ready publish standalone need-finding papers for a variety of user populations. The PL community serves different populations, with different problems, using different techniques. We have started to see excellent need-finding papers that address these populations, problems, and techniques, but we have just scratched the surface.<sup>22</sup>

**Contribution:** Standalone need-finding studies for populations, settings, and tasks that could be particularly well-served by novel programming language research.

*Cognitive and behavioral theory transfer and development.* Psychology, cognitive science, linguistics, and many other fields study the characteristics of human cognition. Their work offers theories about the classes of reasoning that humans find easy, hard, and impossible, with and without training. In the domain of PL design, a scientific understanding of how programmers write programs could guide us to better language and tool designs. In the 1970s, the PoP field started building the foundation for this direction, and their work points us to methods we can reuse for learning about modern high-level languages. Critical work in this field continues, drawing on work in software engineering, psychology, CS education, and HCI. Although language design rarely motivates current work in this area, we see a huge opportunity to design experiments to generate language-relevant theory.

One way to bootstrap theory development is to borrow or adapt theories from other disciplines. Social sciences ranging from psychology and economics to cognitive science, organizational behavior, and learning science offer behavioral theory that may ap-

ply to programming. It is common in other disciplines to write papers that adapt or transfer theory from one domain to another. We know of no examples in programming languages literature, but there are many such papers in HCI<sup>1</sup> and software engineering.<sup>2</sup> As we establish or adapt behavioral theories of programming, language designers can base design decisions on predicted user behavior, rather than direct experimentation with target users, to quickly make languages more useful and usable.

**Contribution:** Theory development and theory transfer, for predicting human cognition and behavior during interaction with programming systems.

*Iterative refinement.* We can learn from users throughout our PL design processes. Formative studies enable designers to learn from users before implementing a complete system. For instance, we can conduct formative user studies with incomplete prototypes, learning where users stumble and what features help them. Such studies can help us ensure our language designs are usable, learnable, and not error-prone before substantial effort is put into developing formalisms, proofs, compilers, and other high-effort artifacts. Methods for soliciting user feedback during the design process include surveys, interviews, focus groups, natural programming elicitation, think-aloud studies, “Wizard of Oz” studies in which a human plays the role of the compiler, and studies with other low-cost prototypes. We can evaluate some of the same questions without even recruiting users, for example, via cognitive walkthroughs or heuristic evaluation. HCI venues often publish papers describing such formative studies and the resulting designs. Programming language designs that have been iteratively refined via formative methods should similarly find a place in the literature.<sup>8</sup> As designers, we should get input from users early and often.

**Contribution:** Language and tool designs guided by user-centered, iterative design processes.

*HCI → PL summary.* With a new, broader, and more diverse audience interested in computing, we face an exciting time in our field's history. As we develop more of the contribution types

described in this work, we are poised to offer useful, usable programming languages and tools that tackle high-impact problems. Taken together, these three contribution types, summarized in Table 2, tell us: Pick good problems, develop theories of human capabilities and behavior, and get frequent user feedback when you lack theory.

**HCI practitioners: Consider the following contribution types.** Going forward, we hope to see a few contributions become more common in the HCI community, as HCI increasingly draws on advances in PL. With new PL techniques, HCI practitioners can deliver even more powerful and flexible interfaces, help users avoid important classes of failures, and offer accessible new pathways into the world of computing.

*PL-backed interfaces.* Where you might typically design a GUI, consider giving your users a language—either a domain-specific programming language or a graphical UI that offers the key components of a language: primitives and means of composition. From here on, we will refer to the class of interfaces with primitives and means of composition, whether they are textual or graphical, as PL-backed interfaces.

PL-backed interfaces offer power and flexibility, enabling new interactions that other UI types cannot support. Consider the success stories of languages like D3<sup>5</sup> and Vega,<sup>33</sup> which could have been encapsulated within authoring tools, but not without sacrificing some expressiveness and user control. With advances in tools for language design and implementation—including support for domain-specific languages, language extensions, embedded languages—it is now much easier to offer PL-backed interfaces.<sup>10</sup>

**Contribution:** Developing or studying languages as UIs.

*Guardrails to make PL-backed UIs safer.* Giving users powerful, flexible PL-backed interfaces can empower them, but it can also empower them to make new mistakes. We can address this by building guardrails into our UIs, tools that prevent or catch errors, bugs, and bad outcomes. For this goal too, PL offers a wealth of techniques for aiding programmers, everything from verification (for example, verifying that a robot control program makes the robot

compliant with human social norms<sup>30</sup>) to program analysis (for example, a spreadsheet extension that identifies likely spreadsheet errors based on discrepancies with other nearby formulas<sup>3</sup> or generates spreadsheet tests automatically<sup>11</sup>).

**Contribution:** Developing or studying techniques for enforcing or encouraging correct use of PL-backed interfaces.

*Non-code inputs to PL-backed UIs.* Although providing a PL-backed interface can put powerful new computing experiences in reach, it often takes more than a well-designed language to help users unlock a PL's full potential. We can help users author complex programs via PL tools that write code based on non-code specifications. Program synthesis paradigms like programming by demonstration, programming by example, and programming by manipulation offer users alternative ways to express their intent. For example, a Helena<sup>7</sup> user demonstrates how to collect the first row of their target dataset in a standard Web browser, and Helena synthesizes a program that traverses thousands or millions of webpages to collect the full dataset. Programming by demonstration thus enables social scientists and other domain experts to collect the data they need from the Web. Leveraging new PL techniques lets us design new interfaces for programming and ultimately brings the power of programming to new audiences.

**Contribution:** Developing or studying techniques for creating code from non-code specifications.

*PL → HCI summary.* We are excited for the potential of PL-backed interfaces in the future of HCI. As our users

face increasingly complex new computing tasks, now is the time to put human-centered languages in the hands of more users. Together these three contribution types, summarized in Table 3, offer a simple takeaway message: Give users PLs, but help them use PLs responsibly, and don't expect code alone.

**Fostering HCI+PL research.** We believe the six contribution types discussed previously—need-finding, behavioral theory, iterative refinement, PL-backed interfaces, guardrails, and creating code from non-code—can advance both human-computer interaction and programming languages, that they represent important new frontiers for both subfields. We want to invest in these contribution types. What actions should we take, as individuals and as a community, to produce more work like this?

► **For new or aspiring PL+HCI researchers:** For new researchers, this article describes classes of work that represent important but under-explored contributions. Are you bringing expertise that would help you write a theory transfer paper? A “guardrails” paper? As our community is opening to these topics, now is a great time to consider these directions. If you're looking to test-drive this path, start attending talks. Take a PL class if you are more familiar with HCI, take an HCI class if you are more familiar with PL, or take one of the new crop of courses at the HCI-PL intersection. Start collaborations across the boundary. Find ways to publish the work in multiple but substantial coherent pieces, if necessary, to reach both fields.

► **For reviewers:** This article provides an overview of why these contribution

**Table 3. What can HCI practitioners borrow from PL? This table summarizes three classes of HCI contribution that we can produce by drawing on PL techniques.**

| Contribution | Key Message                        | Elaboration   |
|--------------|------------------------------------|---|
| PL as UI     | Give users PLs,                    | Giving your users a PL gives them a powerful tool that offers flexibility, expressiveness, and control.   |
| Guardrails   | But help them use PLs responsibly, | Both fields offer methods for building in guardrails—checks in the languages, tools, and environments that make errors less likely when we give users languages as interfaces. For example, static analysis, verification, and type systems can all offer important guardrails.   |
| Beyond Code  | And don't expect code alone.       | Don't expect code alone to be enough to put the target programs in reach, especially for complex domains. Sometimes users need further aids, some of which can come from PL—for example, program synthesis, programming by demonstration, anything that makes code out of non-code, new editors, new programming experiences. |



types are necessary, why they hold the promise of enriching both fields. We encourage you to read more on these topics, but we hope this article is reason enough to think twice before dismissing these contributions, even if the papers strike you as unusual or unprecedented at first.

► **For advisors and mentors:** Increasingly, we find researchers are succeeding not despite but because of their cross-disciplinary research. Students considering work at this intersection are not sacrificing job prospects. And as reviewers in both communities are becoming more open to work that combines contributions in both PL and HCI, there is less and less reason to limit your students to a single domain.

► **For the research community as a whole:** Venues like VL/HCC, PLATEAU, PPIG, and LIVE have long track records of recognizing and evaluating work at the intersection of PL and HCI. However, the work needs to appear at flagship conferences to thrive. These flagship conferences should invite reviewers with PL+HCI expertise and evaluate the work rigorously based on appropriate evidence standards.

► **For the industrial and practitioner community as a whole:** We want to see powerful PL-backed interfaces and usable programming languages reaching real users. We should be pouring resources and engineering effort into making it easier for humans to control computers. Few companies have engineering teams working on language design and language usability questions jointly. If you sell a product—cloud computing resources, data analysis suites—that people use via programming, or that people may want to automate, spin up an engineering team that joins PL and HCI expertise.

## Conclusion

Computers have given us services, scientific results, and communication modes that we would not have achieved without them—but many modern interactions with computers feel constrained. Many users feel as if they work in service of the machine rather than the other way around. Even expert programmers still spend a surprising amount of time wrestling

with the command line or tackling painful sysadmin tasks. If we are successful in this PL+HCI effort, it will be easier for us—programming experts, novices, and previously unreached users alike—to communicate our intent quickly and accurately to computers. It will be easier for us to rally computers to our billions of exciting and diverse human goals. **C**

## References

- Alkhatib, A. and Bernstein, M. Street-level algorithms: A theory at the gaps between policy and decisions. In *Proceedings of the 2019 Conf. Human Factors in Computing Systems*.
- Barik, T., Ford, D., Murphy-Hill, E., and Parnin, C. How should compilers explain problems to developers? In *Proc. Joint Meeting of the Euro. Software Engineering Conf. and Symp. Foundations of Software Engineering*, 2018.
- Barowy, D., Berger, E., and Zorn, B. Excelint: Automatically finding spreadsheet formula errors. In *Proceedings of the ACM on Programming Languages* 2 (2018), 1–26.
- Blackwell, A., Petre, M., and Church, L. Fifty years of the psychology of programming. *Intern. J. Human-Computer Studies* 131 (Nov. 2019), 52–63; <http://oro.open.ac.uk/62027/>.
- Bostock, M., Ogievetsky, V., and Heer, J. D3 data-driven documents. *IEEE Trans. Visualization and Computer Graphics* 17, 12 (2011), 2301–2309.
- Chasins, S. and Phothilimthana, P. Data-driven synthesis of full probabilistic programs. In *Proceedings of the 2017 Computer-Aided Verification*. Springer International Publishing.
- Chasins, S., Mueller, M., and Bodik, R. Rousillon: Scraping distributed hierarchical web data. In *Proceedings of the 2018 Symp. User Interface Software and Technology*.
- Coblentz, M., Aldrich, J., Myers, B., and Sunshine, J. Interdisciplinary programming language design. In *Proceedings of the 2018 Intern. Symp. New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 133–146.
- Cooper, S., Dann, W., and Pausch, R. Alice: A 3-d tool for introductory programming concepts. *J. Comput. Sci. Coll.* 15, 5 (Apr. 2000), 107–116.
- Erdweg, S. et al. The state of the art in language workbenches. In *Proceedings of the 2013 Intern. Conf. Software Language Engineering*.
- Fisher, M., Rothermel, G., Brown, D., Cao, M., Cook, C., and Burnett, M. Integrating automated test generation into the WYSIWYT spreadsheet testing methodology. *ACM Trans. Softw. Eng. Methodol.* 15, 2 (Apr. 2006), 150–194; <https://doi.org/10.1145/1131421.1131423>.
- Green, T. and Petre, M. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Visual Languages & Computing* 7, 2 (1996), 131–174.
- Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the Symp. Principles of Programming Languages*, 2011.
- Hansen, M., Lumsdaine, A., and Goldstone, R. Cognitive architectures: A way forward for the psychology of programming. In *Proceedings of the ACM Intern. Symp. New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2012*, 27–38. ACM, New York, NY, USA; <https://doi.org/10.1145/2384592.2384596>.
- Head, A., Glassman, E., Soares, G., Suzuki, R., Figueredo, L., D'Antoni, L., and Hartmann, B. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the 4th ACM Conf. Learning @ Scale*, 2017.
- Hempel, B., Lubin, J., Lu, G., and Chugh, R. Deuce: A lightweight user interface for structured editing. In *Proceedings of the 2018 Intern. Conf. Software Engineering*, 2018.
- Kay, A. The early history of Smalltalk. *History of Programming Languages—II*, 1996, 511–598.
- Ko, A., et al. The state of the art in end-user software engineering. *ACM Comput. Surv.* 43, 3 (Apr. 2011); <https://doi.org/10.1145/1922649.1922658>.
- Le, V. and Gulwani, S. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 2014 Conf. Programming Language Design and Implementation*.
- Li, T., Radensky, M., Jia, J., Singarajah, K., Mitchell, T., and Myers, B. Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of the 2019 Symp. User Interface Software and Technology*.
- Lieberman, H., Paternò, F., and Wulf, V. *End User Development (Human-Computer Interaction Series)*. Springer Verlag, Berlin, Heidelberg, 2006.
- Ma'ayan, D., Ni, W., Ye, K., Kulkarni, C., and Sunshine, J. How domain experts create conceptual diagrams and implications for tool design. In *Proceedings of the 2020 Conf. Human Factors in Computing Systems*.
- Moritz, D., Wang, C., Nelson, G., Lin, H., Smith, A., Howe, B., and Heer, J. Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. *IEEE Trans. Visualization and Computer Graphics* 25, 1 (2018), 438–448.
- Myers, B., Pane, J., and Ko, A. Natural programming languages and environments. *Commun. ACM* 47, 9 (Sept. 2004), 47–52; <https://doi.org/10.1145/1015864.1015888>.
- Myers, B., Ko, A., LaToza, T., and Yoon, Y. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016).
- Nardi, B. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, USA, 1993.
- Omar, C., Voysey, I., Hilton, M., Aldrich, J., and Hammer, M. Hazelnut: A bidirectionally typed structure editor calculus. In *Proceedings of the 2017 Symp. Principles of Programming Languages*.
- Pane, J. and Myers, B. Usability issues in the design of novice programming systems. Project status report, 08, 1996.
- Pane, J. and Myers, B. The influence of the psychology of programming on a language design: Project status report, 06, 2000.
- Porfiro, D., Saupé, A., Albarghouthi, A., and Mutlu, B. Authoring and verifying human-robot interactions. In *Proceedings of the 2018 Symp. User Interface Software and Technology*.
- Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of Programming Language Design and Implementation*, 2013.
- Resnick, M., et al. Scratch: Programming for all. *Commun. ACM* 52, 11 (Nov. 2009), 60–67; <https://doi.org/10.1145/1592761.1592779>.
- Satyanarayan, A., Moritz, D., Wongsuphasawat, K., and Heer, J. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Visualization and Computer Graphics* 23, 1 (2016), 341–350.
- Shneiderman, B. Direct manipulation: A step beyond programming languages. *Computer* 16, 8 (1983), 57–69.
- Shneiderman, B. and Plaisant, C. Strategies for evaluating information visualization tools: multi-dimensional in-depth long-term case studies. In *Proceedings of the 2006 AVI Workshop on Beyond Time and Errors: Novel evaluation methods for information visualization*, 1–7.
- Verou, L., Zhang, A., and Karger, D. Mavo: Creating interactive data-driven web applications by authoring html. In *Proceedings of the 2016 Symp. User Interface Software and Technology*.
- Yoon, Y. and Myers, B. Supporting selective undo in a code editor. In *Proceedings of the 2015 Intern. Conf. Software Engineering*.

**Sarah E. Chasins** (schasins@cs.berkeley.edu) is an assistant professor of Electrical Engineering and Computer Science at University of California, Berkeley, CA, USA.

**Elena L. Glassman** (glassman@seas.harvard.edu) is an assistant professor of computer science and the Stanley A. Marks and William H. Marks Assistant Professor at the Radcliffe Institute for Advanced Study at Harvard University, Cambridge, MA, USA.

**Joshua Sunshine** (sunshine@cs.cmu.edu) is a senior Research Fellow in the School of Computer Science at Carnegie Mellon University, Pittsburgh, PA, USA.



This work is licensed under a <http://creativecommons.org/licenses/by/4.0/>