What Is a Programming Language, Really?

Amy J. Ko

The Information School
University of Washington, Seattle, WA, USA
ajko@uw.edu

Abstract

In computing, we usually take a technical view of programming languages (PL), defining them as formal means of specifying a computer behavior. This view shapes much of the research that we do on PL, determining the questions we ask about them, the improvements we make to them, and how we teach people to use them. But to many people, PL are not purely technical things, but *socio*-technical things. This paper describes several alternative views of PL and how these views can reshape how we design, evolve, and use programming languages in research and practice.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms Human Factors

Keywords *Definitions, human-computer interaction*

1. What Is a Programming Language?

Casually, the answer to this question is an obvious one: programming languages (PL) are notations for telling computers what to do in the future. For the purposes of conversation, this definition appears conceptually valid and it portrays the experience of programming. It's also kind of boring.

And yet, subtle aspects of definitions can play an outsized role in shaping our views. For example, in the definition above I used the verb "tell". This word implies *human* activities in communication, such as listening, turn-taking and miscommunication. It's imperative tone even connotes a relationship of dominance, with the programmer commanding a computer, and the computer complying. It is these ideas that make this definition feel right: we *do* control computers, there *are* miscommunications, and it frequently feels like computers *are not* listening, despite us being in charge.

However, it also these ideas that make the communication metaphor useful. By thinking of PL through this lens, we can re-envision PL. For example, if programming is about "telling," perhaps IDEs are like human translators, facilitating dialog between two agents that do not speak the same language. Human translators have to listen vigilantly, editorializing for expediency. Perhaps IDEs need to do the same.

PLATEAU'16, November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4638-2/16/11...
http://dx.doi.org/10.1145/3001878.3001880

What other definitions of PL exist and what research do they demand? Clearly, the dominant academic view of PL is a formal one. I asked several PL researchers for their definitions and most provided definitions like "formal expressions of computation." Others went further, making stronger statements such as the Curry-Howard correspondence, which describes programs as proofs and proofs as programs. These conceptions of PL engender mathematical questions, such as: *What is a correct program? How do we prove a program correct? What notations exist to express computation?*

These formal definitions are productive, contributing to many advances in program correctness. That said, this definition is not necessarily the "correct" definition. For example, another view of PL is as a kind of **natural language**. This is, after all, the historical reason why we call PL "languages," and why we use words such as "syntax", "grammar", and "semantics" to specify their rules. Devanbu et al. extended this metaphor through a statistical lens, finding, for example, that defective code is often improbable based on patterns in how a PL is used [6]. These perspectives are also related to linguistic ideas such as the Sapir-Whorf hypothesis, which suggests that language determines thought, or at least influences it. This leads to questions such as: *How do PL abstractions shape the computation that programmers write? What capacity do PL have for novel computational expression? How does language design bias the type of computation that programs do?* By answering these questions, we can begin to see programs as speech, with regularity, but also capacity for syntactic and semantic surprise.

It is also possible to view PL as a kind of **documentation**. After all, it is not source code that we usually execute, but a compiled translation of source, suggesting that the true role of a language is to help programmers *reason* about how a program computes. If we take this view, a good PL should streamline human comprehension, begging questions such as: *What affects language readability? How does notation affect comprehension? What makes a notation defect prone?* Stefik et al. take this view [5], exploring the cognitive properties of PL and their effects on code comprehension.

A more cynical view of PL is as little more than the **glue** between APIs, frameworks, libraries, platforms, and services. This view is often the dominant view amongst modern web developers and other engineers tasked with building large systems inside of large software ecosystems via function calls [2]. It provokes questions such as: *What makes good computational glue? Do PL matter at all, when most modern programs are just function calls?* This view suggests that languages are simply interchangeable connectors between functionality.

Others have viewed PL as an **interface** between a person and a computer. In fact, from a historical perspective, PL were the *first* user interface to computers. From this view, languages have properties that any user interface has, including usability, learnability, feedback, error-proneness, user efficiency, and so on, suggesting questions such as: *How can we measure PL usability? What makes a PL learnable? What aspects of PL design affect its error-proneness?* Stefik [5] investigates these questions too, building upon HCI perspectives from Pane and Myers in their design of HANDS [4], which sought to design a language based on the children's descriptions of the rules of digital games.

A related view frames PL as **notations for describing the future**. For example, Blackwell defined programming as an activity in which someone shapes multiple possible future states in time and space with the aid of notation and abstraction [1]. From this view, we might ask: *What kinds of future states of the world can be described? What are the limits of people's ability to reason about future states? How do notations enable people to reason about classes of future states?* These questions are psychological, inquiring how humans use notation to model the behaviour of not only computers, but people and other entities that can take instruction.

Alternatively, there are many that view PL as **expressive media**. Resnick, for example, views Scratch [7] as media for self-expression, and Shapiro views Blocky Talky [3] as supporting play. These perspectives frame PL as like any other creative medium, but with opportunities for interactivity and storytelling. This view suggests questions about the expressive capacity of a PL, such as: *How can we measure the breadth of a language's expressive capacity? How does a notation's black box abstractions limit expression? What can a PL uniquely express that other PL cannot?* Ideas such as McLuhan's "the medium is the message" extends these questions further, suggesting that languages might shape perceptions of what computation is and what it is for.

Some view programs as policy, expressing an agreement between designers and users about what computation will and will not be possible. In fact, in many software engineering contexts, programs are expressions of contractually obligated requirements. In this view, PL are the medium through which a **contract** is satisfied. If we extend this to a view of programs as *social* contracts (ala Rousseau [8]), we might ask questions such as: *What rights do users have in shaping these contracts? Do PL have to be readable by all in order to preserve human agency? What power do PL designers have in shaping policy?* By thinking of languages as media in which laws, regulations, contracts, policies, and rules are expressed, we can productively and provocatively explore the relationship between programs and civil rights.

My colleagues in information science often led me to wonder whether it is *information* or *computation* that is the more dominant phenomenon in computing. After all, when you have a pioneer like Bill Gates claiming "content is king," it suggests that computation, algorithms, and data structures might just be **conduits for the flow of information** through social systems. In this view, we wonder: *How do PL bias and privilege information? How do languages warp the meaning in information through their abstractions? As conduits for information, what benefits and trade-offs do PL offer relative to other types of information containers such as writing and speech?* These questions view computing as (imperfectly) modelling information.

Because knowledge of PL is still limited to a small minority of humanity but are used to shape an increasing proportion of humanity's experiences in life, PL can also be defined as a form of **power**. Knowing a language means control over a computer, but it also means control over others computers, and by extension, designing and deciding how people experience the extent of their lives. In the same way that reading literacy is a form of power, PL provokes questions such as: *With great programming power, should great responsibility come as well? What kind of social power do programming languages provide? Who has rights to the social power of programming languages?*

To those who do not know a PL, PL might also be viewed as a **path to prosperity**. Knowing a PL means having access to the highest paying jobs in our global economy. In many places in the world, they represent stability and wealth. This view of programming languages asks: *Who should have access to programming language knowledge? Can everyone learn a programming language? What are the economic consequences of everyone knowing a little bit about PL?* As the world begins to shift education policy to bring computing education to everyone with initiatives such as the White House's CS for All and other countries' efforts to implement K-12 CS education, it is up to researchers to carefully consider the answers to these questions.

2. What's Next?

Each of these views contains a research agenda. Some of these agendas are already deeply explored—we understand what PL are formally and we increasingly understand what they are as tools. Other agendas, particular those that probe the human, social, societal, and ethical dimensions of PL, are hardly explored at all. As researchers, we should be concerned with every view, and see each as an opportunity to understand holistically how PL can shape our world.

References

[1] Blackwell, A.F. (2002). First steps in programming: A rationale for Attention Investment models. *IEEE VL/HCC*, 2-10.

[2] Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *ACM CHI*, 1589-1598.

[3] Deitrick, E., Shapiro, R. B., Ahrens, M. P., Fiebrink, R., Lehrman, P. D., & Farooq, S. (2015). Using distributed cognition theory to analyze collaborative computer science learning. *ACM ICER*, 51-60.

[4] Pane, J.F., Myers, B.A., & Miller, L.B. (2002). Using HCI techniques to design a more usable programming system. *IEEE VL/HCC*, 198-206.

[5] Stefik, A. & Hanenberg, S. (2014). The programming language wars: questions and responsibilities for the programming language community. *ACM Onward!*, 283-299.

[6] Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., & Devanbu, P. (2016). On the naturalness of buggy code. *ICSE*, 428-439.

[7] Resnick, M., & Silverman, B. (2005). Some reflections on designing construction kits for kids. *Interaction Design and Children*, 117-122.

[8] Rousseau, J.J. (1913). *Social contract & discourses*. New York: E.P. Dutton & Co.